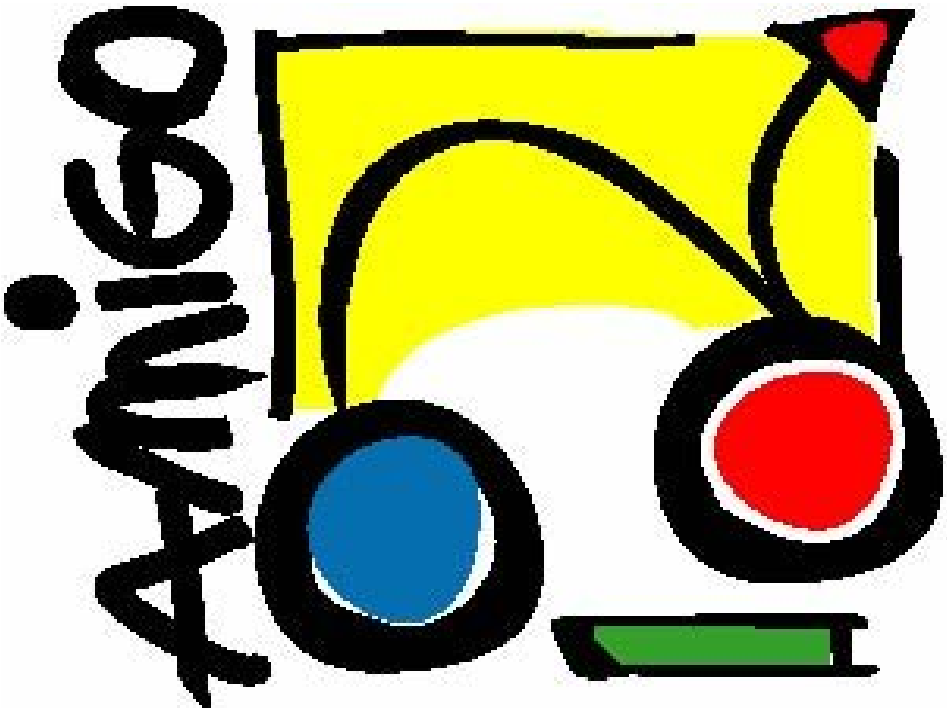


IST Amigo Project
Deliverable D3.2

Amigo Middleware Core: Prototype Implementation & Documentation

IST-2004-004182
Public

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

Deliverable Number	:	D3.2
Title of Deliverable	:	Amigo Middleware Core: Prototype Implementation & Documentation
Nature of Deliverable	:	Public
Internal Document Number	:	amigo-d3.2-final
Contractual Delivery Date	:	February 28, 2006
Actual Delivery Date	:	March 31, 2006
Contributing WPs	:	WP3
Editor	:	**INRIA:** Nikolaos Georgantas
Author(s)	:	**INRIA:** Sonia Ben Mokhtar, Yérom-David Bromberg, Nikolaos Georgantas, Noha Ibrahim, Valérie Issarny, Anupam Kaul, Frédéric Le Mouël, Daniele Sacchetti, Ferda Tartanoglu **FT:** Anne Gerodolle, Mathieu Vallée **ICCS-NTUA:** Miltiades Anagnostou, Ioannis Papaioannou, Ioanna Roussaki, Dimitris Tsesmetzis **IKER:** Jorge Parra **IMS:** Marco Ahler **Microsoft:** Ron Mevissen, Daniel Schaffrath, Stephan Tobies **TELIN:** Henk Eertink, Pravin Pawar, Remco Poortinga, Andrew Tokmakoff **TID:** Jordi García, José María Miranda, Marc Planagumà, Álvaro Ramos, Sergi Sorribas **VTT:** Jarmo Kalaoja, Julia Kantorovitch

Abstract

D3.2 is the first deliverable on the prototype implementation and associated documentation of essential Amigo middleware components, while it also reports on ongoing conceptual and design work for other Amigo middleware components. D3.2 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 – in a restricted way – on the Amigo OSS Repository - Public Web site

(http://amigo.gforge.inria.fr/home/index.html). D3.2 addresses the Amigo programming and deployment framework, service description vocabulary and language, aspects of service discovery, service discovery and interaction interoperability, domotic infrastructure, security, content delivery, and data store.

Keyword list

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, middleware, service discovery protocols, service interaction protocols, programming and deployment framework, context, quality of service, multimedia streaming, content distribution, security, privacy, data storage

Table of Contents

Figures

Tables

1 Introduction

The present Deliverable D3.2 is the first one officially concerning the implementation of the Amigo Base Middleware (or simply middleware), even if a first prototype implementation of essential middleware functionalities was already detailed in Deliverable D3.1b [Amigo-D3.1b]. More specifically, in D3.1b, middleware core functionalities were developed and incorporated in an Integrated Prototype, which was demonstrated at the first Project Review. That integrated prototype provided a first, proof-of-concept integration of several interoperability mechanisms across the Amigo domains, i.e., the PC, mobile, domotic and CE domains. For the same review, a second demonstrator was realized of security-related mechanisms, which were detailed in Deliverable D3.1c [Amigo-D3.1c].

By title, the present deliverable concerns the prototype implementation and documentation of the Amigo middleware core; however, we deliver and document herein implementation of both middleware core functionalities and upper middleware functionalities. Further, we report on middleware functionalities for which conceptual and design work is still being carried out. Thus, the level of presentation of different middleware functionalities differs depending on their current stage of progress.

More specifically, for ongoing conceptual and design work, which elaborates further on the bases set by the previous deliverables [Amigo-D2.1, Amigo-D3.1a,b,c], we follow the conventional way of reporting, already employed in the previous deliverables.

On the other hand, for implementation work, we follow component-oriented delivering and reporting. An overview of all the Amigo middleware components – all of them open-source – currently under development or planned to be developed has been provided in the Intermediate Amigo OSS Report [Amigo-OSSReport]. For most of these components, detailed design has been elaborated in previous deliverables [Amigo-D3.1a,b,c]. These components are currently under development, which is supported by the Amigo OSS Repository - Source Code Management (SCM) [Amigo-OSS-SCM], accessible only internally by the Amigo Consortium (see [Amigo-D9.5]). In the present document, we provide an updated or extended overview (with respect to [Amigo-OSSReport]) for each component under development. For each such component, we further deliver as part of D3.2:

- Source code of the current prototype version, if one is already available;
- User's guide and developer's guide documents, if already available;
- Javadoc[1] (or equivalent for C#) documentation, if already available.

Following the same for the service description vocabulary and service description language, we deliver for each one of them besides the present document:

- 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.

All material besides the present document that makes part of D3.2 is sufficiently referenced herein and is accessible – currently in a restricted way – on the Amigo OSS Repository - Public Web Site [Amigo-OSS-Pub] (see [Amigo-D9.5]). Certainly, online documentation for OSS components and ontologies is inherently living documentation, which is constantly evolving along with the evolution of the components or ontologies. Currently, the online

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

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

documentation that we provide is at an early stage and will take a form closer to complete when the first public versions of components and ontologies will be available.

The specific Amigo middleware functionalities, components, and ontologies reported in the chapters of the present document (and, for the latter two, further, when already available, delivered in source along with online documentation) are:

- *Programming and deployment framework*, enabling software development and deployment not only for the Middleware components (WP3) but also for the Intelligent User Services (WP4) and Application (WP5, WP6, WP7) components. Global principles and techniques have been developed and applied on two software platforms, OSGi and .NET, providing two alternative solutions already used in practice by Amigo developers. We present conceptual work, and deliver the current implementation of essential components realizing the framework along with additional documentation (Chapter 2).
- *Service description vocabulary ontologies*, enabling systematic, formal representation of concepts in the Amigo environment towards establishing common understanding based on semantics among interacting entities, as the dynamics and openness of the environment make impossible the enforcement of a single syntactic standard. The main goal of such concept representation is the employment of represented concepts in the description of Amigo services. We present the conceptual approach to designing the vocabulary ontologies, and deliver the OWL specification of the produced ontologies along with additional documentation (Chapter 3).
- *Service description language*, enabling systematic, formal description of Amigo services by further employing the vocabulary ontologies. Any service of any service technology may be described functionally along with its underlying middleware, as well as non-functionally in terms of context and QoS. We present the conceptual approach to designing the language, and deliver the OWL specification of the produced language along with additional documentation. Further, we report on conceptual work as well as early implementation work for performance evaluation on a number of aspects of service discovery; we carry this out at a generic level not yet connected with the language (Chapter 4).
- *Service discovery and service interaction interoperability (SDI and SII)*, enabling integration of heterogeneous devices and hosted services in the networked home environment. Based on previous – detailed for SDI / early for SII – design and implementation work, we elaborate a detailed design for SII and provide an early implementation for evaluating its performance (Chapter 5).
- *Domotic infrastructure*, enabling exposing (for discovery and interaction) heterogeneous domotic devices as unified software services using standard service technologies. We provide an overview of the infrastructure and the components currently under development realizing this infrastructure (Chapter 6).
- *Security and Privacy*, enabling the Amigo security framework, where user and device access to the home can be controlled based on authentication and on a role-based authorization scheme. We deliver the current implementation of the components realizing the security framework along with additional documentation (Chapter 7).
- *Content Delivery*, enabling making available, distributing and adapting content in the Amigo home for Amigo services and applications. We provide an overview of the content delivery infrastructure and the components currently under development realizing this infrastructure, and deliver early additional documentation (Chapter 8).
- *Data Store*, offering a generic storage capability to other components and applications inside an Amigo System. We provide an overview of the Data Store component currently under development, and deliver early additional documentation (Chapter 9).

Finally, we conclude with a short discussion on the principal points and progress of the present deliverable (Chapter 10).

2 Programming and deployment framework

2.1 Objectives

This chapter proposes a component model that allows clear separation of development and deployment issues. As shown in Section 2.3, the expected result is an “Amigo bundle repository” where components will be available for downloading and installation.

We propose general principles for using a platform like OSGi or .Net, and provide guidelines to developers of functional blocks so that their work can be packed into components that can be further (at deployment time) composed in an arbitrary manner with other components.

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 SDP, communication protocols and (when necessary) interoperability methods.

This chapter is organized as follows: Firstly, we define some vocabulary that we use in the rest of the chapter (Section 2.2), and indicate the results expected of this effort (Section 2.3). We then present the .Net Amigo programming environment (Section 2.4) and, finally, present the OSGi Amigo programming (Section 2.5) and deployment environment (Section 2.6). For both of these environments, this document itemizes a list of components that are either already available or under development. A complete set of component documentation (including a user’s guide, a developer’s guide, and tutorials) is available separately as online documentation referenced at [Amigo-OSS-Pub].

2.2 Vocabulary

Terms like services, interfaces, and components are strongly overloaded in computer science, and – for example – .Net and OSGi use different words to refer to similar concepts, or even the same word to refer to the same concept. In this document, we generally adopt the vocabulary used in OSGi specifications.

In the following:

A *software node* runs on a physical node. It may be a .Net platform, an OSGi platform or any process.

More specifically, a *platform* is a software node where software components can be deployed.

A *functional block* is identified as such in the abstract Amigo architecture: For example, Context management is a functional block; Security is a functional block, etc.

A *software component* is some part of a functional block that can run on a software node. It communicates with other components of the same functional blocks using some protocol stack.

A *bundle* is a software component that can be deployed on an OSGi platform. In this section, the word *bundle* is also used to refer to deployable .Net components.

An *interface* defines a set of operations or methods. This word may refer to a programming language related abstraction (a Java or C# interface can be implemented by a Java or C# class) or to the interface of a remote service, described in an interface description language (IDL). For example, the interface of a web service is described in WSDL.

A *service* is an artifact provided by a component that offers an interface. It may be

- A local service: accessible from inside the same software node, in the form of an object implementing a given interface (C# or Java).
- A remote service: accessible from a remote client using a communication protocol through a “service api”, or SAPI.

2.3 Expected results

The expected result of this effort is a repository of bundles, which may be called the “Amigo Bundle Repository”. Two versions of the repository will be provided, one for .Net bundles, the other for OSGi bundles. Services packed as applications could also be made available on a repository. These repositories will be accessible through http. They will provide for each functional block: the list of available bundles, and for each bundle, the documentation of the bundle, the source code of the bundle, and the deployable bundle itself.

The first version of the Amigo OSGi repository is publicly accessible at http://amigo.gforge.inria.fr/obr/v0 (also referenced at [Amigo-OSS-Pub]).

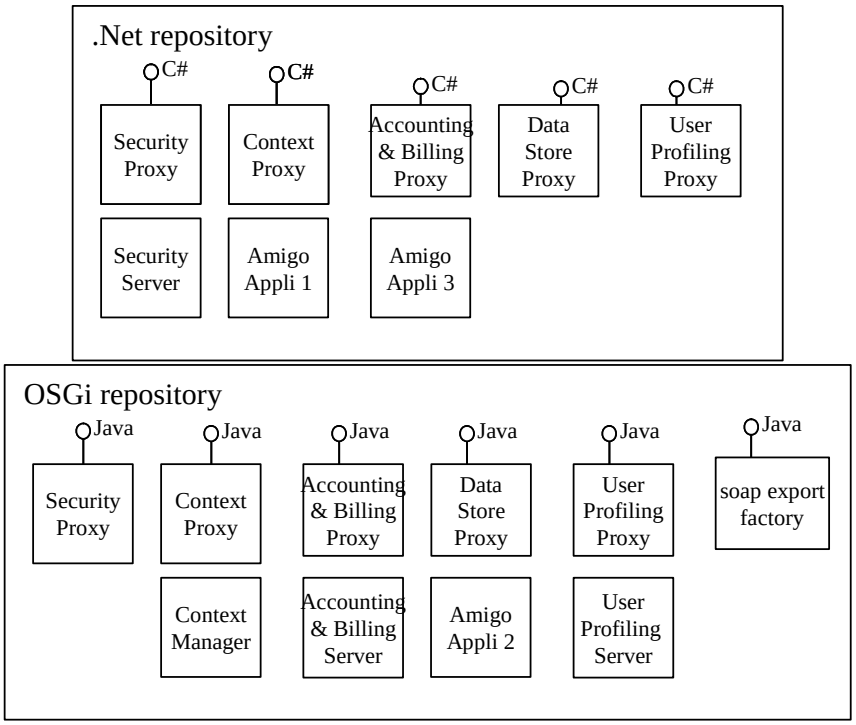


Figure 2-1: The Amigo Bundle Repository contains a set of bundles that can be deployed on a platform (OSGi or .Net). Some bundles provide a Java or C# api that other bundles deployed on the same platform can use. Note: This figure is only illustrative and does not indicate the Amigo bundle repository's final state.

Each functional block may provide one or more bundles that correspond to different parts of the functional block. We can distinguish between three main types of bundles: “server bundles” (e.g. the security server bundle), “proxy bundles” (e.g. the security proxy bundle), and “local bundles” (e.g. the soap export factory).

- Server bundles will be deployed on only one or a few nodes of a network; they will be in general developed for only one of the targeted platforms. Access to services provided by the server bundles is done through a remote interaction protocol. The word “server” here does not refer to a “client-server” model but simply to the fact that a remote service is offered.
- Proxy bundles should be available for both programming frameworks. They will allow reusability of code among developers of components using a given functional block. Proxy bundles do not provide remote services but rather a local API that gives access to the distributed functional block by means either of simple “stubs” (local representative of remote services working in a client/server model) or of "smart proxies" that offer a simplified view of a distributed system by possibly handling complex interaction. Proxy interfaces allow developers to use a functional block without knowing (at development time) the distributed architecture of the block. In some cases, several implementations of a proxy may be available, and the choice of which implementation to use could be made at deployment time. This is particularly useful when using a complex functional block, where the functionalities offered are clearly identified, but the distribution of these functionalities over the network will depend on the network configuration and the capabilities of the nodes.
- Local bundles are not linked to any functional block. They provide services such as logging, protocol adapters, etc.

Figure 2-1 shows how the Amigo bundle repository could look like: if we take the example of the security functional block, on the .Net repository a “server bundle” and a “proxy bundle” may be available, whereas on the OSGi repository only a “proxy bundle” is available. The security proxies offer a simple API to interact with the security functional block. They hide the details of discovering the security server, managing the security protocol, possibly reconnecting to a new security server in case the current security server becomes unavailable, etc.

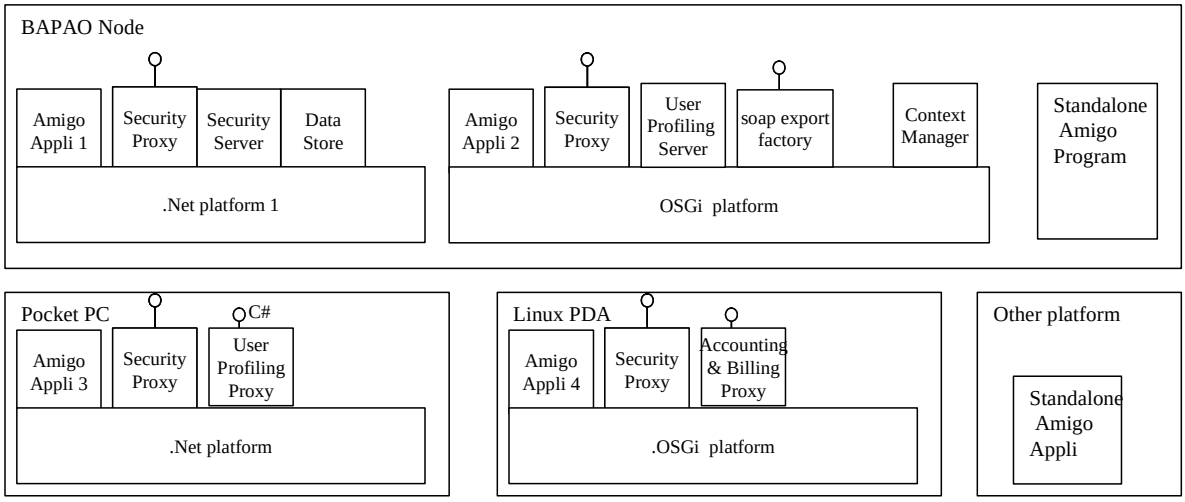


Figure 2-2: An Amigo Network with 4 physical nodes and 6 Amigo software nodes with different configurations. The security proxies discover the security server using WS-discovery and interact with it using SOAP.

Figure 2-2 shows an example of use of the Amigo Bundle Repository shown in Figure 2-1: all server bundles are deployed on a PC (called BAPAO, for “Base Amigo Peripheral that is Always On”), together with some proxy bundles. The security proxy is also deployed on the OSGi platform, for local use by the user profile server proxy. Applications deployed on a

Pocket PC and Linux PDA also access the security manager thanks to the locally deployed security proxy. The standalone Amigo applications (SN3 and SN6) may also use the security manager, but they have to manage the complete interaction protocol.

2.4 Amigo .Net programming framework

Provider

Microsoft

Introduction

The programming framework is considered an essential part for an Amigo System since it will be used by nearly all application/component developers as a base.

The goal of our programming framework is to support these developers by enabling them to write their application or component software in a short timeframe by relieving them of time consuming and complex tasks. In this way, developers can concentrate on their core business logic and are not distracted/bothered by complex technologies like remote communication or discovery protocol details.

The programming framework provides developers with a platform on top of the .Net platform that abstracts communication and discovery details from their software. It is almost as if the developer does not need to be concerned about these issues; writes his software and in the end incorporates it seamlessly into the programming framework to benefit from its functionalities.

The programming framework will be further extended with common functionalities like logging, configuration, versioning, remote management and software replication mechanisms that are related to deployment. Remote interfaces like those used for configuration and management will be aligned between the programming framework on .Net and the OSGI-based programming framework.

Development status

Development will start in Q1 2006.

Intended audience

The programming framework is intended for component as well as to application developers.

License

See EMIC license (Annex A).

Language

C#

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

Hardware: PC/Laptop/PDA/Smartphone

OS: Windows XP / Windows Server 2003 / PocketPC 2003 / SmartPhone 2003

Software: .Net for Windows / .NetCF for Windows

Platform

Microsoft .Net 2.0 / Microsoft .NetCF 2.0

Tools

Generic .Net tools

Visual Studio 2005

Files

See [Amigo-OSS-Pub]

Documents

Developer's guide: See [Amigo-OSS-Pub]

User's guide: See [Amigo-OSS-Pub]

Tasks

There will be an initial release in M24. Full release in M30.

Bugs

Not yet available

Patches

Not yet available

2.5 Amigo OSGi programming framework

2.5.1 Context

An OSGi platform allows deployable elements, called "bundles", to be remotely installed from any URL, e.g., from http servers. A bundle is a jar file containing Java code, a special manifest describing the bundle's capabilities and possibly other resources. When started, a bundle can provide "services". In OSGi terminology, a service may be any Java object. OSGi platforms provide a service registry which allows:

- Registering an object as a (local) service, which means associating this object with a list of properties described in an LDAP syntax, among which is the provided Java interface(s).
- Look up services matching target criteria.

Additionally, the OSGi framework takes care of the life-cycle of services and automatically suppresses the references of services registered by a bundle when this bundle is stopped. As

any Java object can be registered as an OSGi service, Amigo APIs developed in Java can easily be provided as OSGi services and packed in OSGi bundles.

Many useful OSGi bundles are already available on the Web. Here, we briefly introduce some open source bundles that the Amigo OSGi programming and deployment framework uses:

- The Oscar Bundle Repository[3] bundle allows accessing a set of OSGi bundles on a repository accessible through http. When installing a new bundle, the OBR bundle will take care of dependencies and install (if necessary) bundles that provide packages needed by this bundle.
- The Service binder[4] provides an XML language to declare services offered and required by a Java component. Service binder is now standardized in OSGi R4[5] as "declarative services".
- Oscar[6] provides an implementation of the standard OSGi HTTP service, which allows servlet deployment on an OSGi platform. This will be the base to provide Amigo services as web services.
- The domoware[7] UPnP base driver implements the UPnP base driver specification standard defined by OSGi.
- Knopflerfish[8] has packed the Axis[9] servlet into a bundle. When the Axis bundle is running, objects registered to the OSGi lookup with property "SOAP.service.name" set are automatically made available as Web services.

2.5.2 Description of work

The Amigo OSGi programming framework includes standard or legacy OSGi bundles, as those described above. The work related to OSGi in Task 3.4 will consist of:

- Maintaining the Amigo OSGi Bundle Repository – help partners to pack Java components in the form of an OSGi bundle and make them available on the repository.
- Provide additional bundles to ease the development of distributed services. This is described in the following sections. Section 2.5.3 introduces the main principles; Section 2.5.4 gives an example of code using this environment and Section 2.5.5 details the subcomponents that are already available or are planned.
- Provide enhanced tools that ease the deployment of Amigo bundles according to semantic criteria. This is described in Section 2.6.

2.5.3 Components aimed to ease the development of distributed services

We further rely on the fundamental concepts of "export factories" and "binding factories". An "export factory" is a service that makes a Java object remotely available. For this purpose, an export factory provides a method (called "export"). The result of "Exporting a service" is an "Amigo reference" that can be serialized and published using a discovery protocol. This

[3] http://oscar-osgi.sourceforge.net/

[4] http://gravity.sourceforge.net/servicebinder/

[5] "OSGi Service Platform, Release 4 CORE" , http://www.osgi.org/osgi_technology/

[6] http://oscar.objectweb.org/

[7] http://domoware.isti.cnr.it/

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

[9] http://ws.apache.org/axis/

"Amigo reference" contains all useful information to allow a client to access the service, such as the host name and port number where the service can be found, the communication protocols that can be used, etc... Exporting a service may or not involve the construction of some dedicated objects on the server. Symmetrically, a "binding factory" is used on the client to access a given service, given an "Amigo reference". A binding factory provides a method that takes an Amigo service description as parameter and returns a "stub". This stub can then be used by the client to communicate with the remote object. Export factories, binding factories and SDP implementations are packaged in OSGi bundles as follows:

- The Amigo core bundle provides Java interfaces representing the export factory, binding factory, and lookup abstractions, together with basic mechanisms which allow clients to export an object by using (in a transparent way) the currently deployed export factory, or to build a stub to connect to a remote service.
- Specialized bundles (e.g. the kSOAP export bundle, the SLP bundle) provide implementations of these interfaces based on a specific protocol and a specific technology.

Programmers of Amigo-aware bundles that use this framework only need to know the interfaces defined in the Amigo core bundle. The choice of the underlying protocol that an Amigo-aware bundle uses is done at deployment time and depends on the specialized bundles that are deployed together with this Amigo-aware bundle.

A subset of these bundles will be installed on every OSGi node, depending on which type of application bundles it will host and the capacities of the hardware platform. It may be desirable to limit the memory footprint on embedded devices. Furthermore, a specific protocol may be preferred depending on the network configuration: in some circumstances, http protocol may be preferred because of firewall problems, whereas for communication between Java nodes JRMP (Java Remote Method Protocol) may be preferred for performance reasons. Therefore, an OSGi platform running on a PDA and hosting only client applications could host only binding bundles, and be limited to a single binding technology (e.g., kSOAP) whereas a platform running on a PC and hosting a variety of server and client applications would host several export and binding factories, so as to maximize interoperability with other nodes.

The proposed approach facilitates the introduction of new protocols, as this involves only writing the corresponding export and binding factories and packing those as OSGi bundles that register the factories as services. These bundles can then be installed on already existing OSGi nodes, and provide the possibility for already installed applications to export their services or access services using this new protocol. This method makes it possible to expose a service through several protocols, keeping the overhead for the service programmer as lightweight as possible. Exposing the same service according to various protocols reduces the need for translation services and increases communication efficiency. A "client" can access a service running on a remote OSGi platform, provided there is a binding factory running on the client's OSGi platform that is compatible with one of the export factories used on the server's OSGi platform. However, in the case of incompatible binding/export factories (e.g., an embedded server that would provide only a SOAP export service and an embedded client that would contain only a RMI binding service), interoperability methods developed inside Amigo Task 3.3 will be used. Note that interoperability methods may themselves be packed as bundles and deployed on an OSGi platform.

2.5.4 Writing an Amigo service and an Amigo client

The code presented hereafter is for illustration purposes only, as the interfaces in this document may become obsolete with further development. An updated tutorial can be found at http://amigo.gforge.inria.fr/obr/tutorial/ (also referenced at [Amigo-OSS-Pub]).

2.5.4.1 Writing an Amigo service

We suppose here that a developer writes a Java class (HelloImpl) that implements some Java interface (Hello). This developer wishes to make the object available on the network as a *service*, using the default Amigo communication and discovery protocols.

Hereafter is the code of this component.

```
1    Public class HelloImpl implements Hello{
2         //implement the Hello interface
3         Public String sayHello(String argument){
4              …..
5         }
6         // define fields that reference the middleware Amigo
     components
7         AmigoLdapLookup lookup;
8         ServiceExporter serviceExporter;
9         // define methods that set these fields
10        public void setLookup(AmigoLdapLookup lookup){
11             this.lookup=lookup;
12        }
13        public void unsetLookup(AmigoLdapLookup lookup){
14             if (lookup==this.lookup) lookup=null;
15        }
16        public void setServiceExporter(ServiceExporter
     serviceExporter){
17             this.serviceExporter=serviceExporter;
18        }
19        public void unsetServiceExporter(ServiceExporter
     serviceExporter){
20             if (serviceExporter==this.serviceExporter)
     serviceExporter=null;
21        }
22
23        public void activate(){
24             // 1- create an instance of “AmigoService” that
     describes this object
25             AmigoService service =
     serviceExporter.createService(server);
26             // 2- create an “exported reference” so that this object
     is accessible through a remote protocol (e.g. SOAP)
27             service.exportMethods(AmigoReference.DEFAULT,
     “test.Hello”);
28             // 3- advertise this reference as a “Hello” service with
     some additional property called nodeName
29             service.addProperty(“serviceType”,”Hello”);
30             String nodeName = System.getProperty(“nodeName”);
31             service.addProperty(“nodeName”,nodeName);
32             lookup.register(service);
33        }
34   }
```

In this example, the HelloImpl class uses the service binder to find instances implementing the ServiceExporter and AmigoLdapLookup interface. To that purpose, the developer has defined 2 fields (lines 7 and 8) and written methods that set this field (lines 10 to 21). He or she defines some metadata (shown below) that describe the dependencies of HelloImpl (it needs a ServiceExporter and Lookup instance to work properly), then packs this class into an OSGi bundle together with a service binder activator.

```
<?xml version="1.0" encoding="UTF-8"?>
<bundle>
  <component class="com.francetelecom.amigo.hello.HelloImpl">
    <requires service="com.francetelecom.amigo.core.AmigoLdapLookup"
      filter=""
      cardinality="1"
      policy="dynamic"
      bind-method="setLookup"
      unbind-method="unsetLookup"
    />
    <requires service="com.francetelecom.amigo.core.ServiceExporter"
      filter=""
      cardinality="1..n"
      policy="dynamic"
      bind-method="setServiceExporter"
      unbind-method="unsetServiceExporter"
    />
  </component>
</bundle>
```

This metadata describes that the component needs an instance of AmigoLdapLookup and at least one instance of ServiceExporter.

When deployed, the service binder will create an instance of HelloImpl and create an instance manager for this component. This instance manager is in charge of calling the setLookup and setServiceExporter method when the lookup and service exporter will be available. Once both dependencies are resolved the activate method is called. The service is exported (lines 25-27), i.e., a reference allowing to access this object remotely, e.g., a SOAP URL, is created) and then registered with SDP with two properties, serviceType and nodeName (lines 29-32).

2.5.4.2 Discovering and using a service

The developer now wants to write a service that needs to access an instance of Hello service. For that purpose, (s)he writes a HelloUser class.

```
Public class HelloUser {
   // defines a field that references the Amigo Lookup
   AmigoLdapLookup lookup;
   // define methods that set these fields
   public void setLookup(AmigoLdapLookup lookup){
      this.lookup=lookup;
   }
   public void unsetLookup(AmigoLdapLookup lookup){
      if (lookup==this.lookup) lookup=null;
   }

   public void activate(){
      // 1- find which “Hello” services are available on the network
      String request = “serviceType=Hello”;
      AmigoService[] services = lookup.lookup(request);
      // 2- invoke all Hello services
      for (int i=0;i<services.length;i++){
           // print out the location of the service
           System.out.println(“found a hello service at location”+
              Services[i].getProperty(“serviceLocation”);
           try{
              // get a stub
              Stub stub=services[i].getSpecificStub(“test.Hello”);
              String result = stub.sayHello”(“World”);
              System.out.println(“this service answers “+result);
              }catch(AmigoException ex){
                   System.err.println(“impossible to create a stub for
reference “+services(i).getReference());
              }
      )
   }
}
```

The HelloUser class also relies on the service binder to discover the instance of the lookup middleware component. Hereafter is the metadata of this component.

```
<?xml version="1.0" encoding="UTF-8"?>
<bundle>
  <component class="com.francetelecom.amigo.hello.HelloUser">
    <requires service="com.francetelecom.amigo.core.AmigoLdapLookup"
      filter=""
      cardinality="1"
      policy="dynamic"
      bind-method="setLookup"
      unbind-method="unsetLookup"
    />
  </component>
</bundle>
```

2.5.4.3 Deploying the HelloImpl and HelloUser components

The HelloImpl and HelloUser components may be deployed on any node of the network that runs an OSGi platform. When activated, HelloUser will discover all instances of HelloImpl and call the sayHello method. The only conditions are that HelloImpl is deployed together with (at least) a bundle providing an ExportFactory service and a bundle providing an AmigoLdapLookup service, and HelloUser is deployed together with (at least) a bundle providing an AmigoLdapLookup service and a bundle providing a BindingFactory service able to handle the references created by the export factory used by HelloImpl.

2.5.5 List of Amigo OSGi bundles

As stated before, the OSGi-based programming & deployment framework is composed of a series of OSGi bundles. A subset of these bundles or all these bundles may be installed on each OSGi platform of an Amigo system. The set of bundles installed on an OSGi platform determines the Amigo profile of this platform.

The OSGi bundles presented herein may belong to different categories:

- Encapsulation of OSS libraries developed in another open source project. Then, the license terms should be the same license terms as those of the original project.
- Original Amigo bundles. The license chosen by France Telecom for these bundles is LGPL. These bundles include an Amigo core bundle (which provides interfaces and basic mechanisms) and specialized bundles that provide adaptation to different protocols.

These bundles will be available on the Amigo OSGi bundle repository, which will contain: (i) binary bundles ready for deployment on an OSGi platform, (ii) documentation associated to these bundles, and (iii) source code corresponding to the binary release. The source code will also be available on the Amigo SVN repository.

The Amigo OSGi bundle repository is available at http://amigo.gforge.inria.fr/obr/ (also referenced at [Amigo-OSS-Pub]).

Table 2-1 lists sub-components of the OSGi framework and their dependencies.

Component	Type	Depends on	License	Availability
log4j	Ext. library		Apache	Month 20
kSOAP	Ext. Library		BSD	Month 20
Amigo core	Amigo	Log4j	LGPL	Month 20
Amigo kSOAP export factory	Amigo	kSOAP, Amigo core, OSGi HTTP service	LGPL	Month 20
Amigo kSOAP binding factory	Amigo	kSOAP, Amigo core	LGPL	Month 20
Amigo Axis export factory	Amigo	Amigo core, Axis, OSGi HTTP service	LGPL	Month 22
Amigo Axis binding factory	Amigo	Amigo core, Axis,	LGPL	Month 22
Amigo SLP adapter	Amigo	Log4j, Amigo core	LGPL	Month 20
Amigo UPnP adapter	Amigo	Log4j, Amigo core, OSGi UPnP base driver	LGPL	Month 24
Amigo WS-discovery adapter	Amigo	Log4j, Amigo core, kSOAP	LGPL	Month 24
Amigo Service Binder	Amigo	Log4j, Amigo core	LGPL	Month 27
Amigo Semantic adaptation bundles	Amigo	Log4j, Amigo core	LGPL	Month 30

Table 2-1: Sub-components of the OSGi framework

2.5.5.1 log4j Bundle (Library Bundle)

Provider

Library provided by Apache / OSGi encapsulation by France Telecom

Introduction

This bundle encapsulates the log4j library, an open flexible logging system for Java applications. log4j is developed within the Apache project (http://logging.apache.org/log4j).

Development status

Done: encapsulation of log4j 1.2.13

To be provided M21: encapsulation of log4j mini (to be used on constrained devices)

Intended audience

This is general purpose software for any Java developer.

License

Apache license

Language

Java

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

Hardware / OS: Any OS supporting Java personal profile or J2SE

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

- log4j.jar contains the log4j bundle
- test-log4j.jar contains a bundle that uses the log4j bundle

Documents

For general documentation see http://logging.apache.org. The OSGi bundle will be provided with an example of use.

Tasks

First release Month 18 for Amigo partners

Public release Month 20

Bugs

None so far

Patches

None so far

2.5.5.2 kSOAP Bundle (Library Bundle)

Provider

Libraries come from kObject and kXML projects. OSGi encapsulation by France Telecom

Introduction

This bundle encapsulates the kSOAP2 and kxml2 libraries, which allow writing XML- or SOAP-related applications for any Java target (Midp, personal profile, J2SE). kSOAP2 is developed inside the kObject project (http://kobject.sourceforge.net); kXLM2 is developed within the kXML project (http://kxml.sourceforge.net/).

Development status

Done

Intended audience

Java developers that want to place SOAP calls, answer SOAP calls or manipulate XML.

License

BSD

Language

Java

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

Hardware / OS: Any OS supporting Java personal profile or J2SE

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

- ksoap2.jar contains the ksoap2 bundle

Documents

For general documentation see http://kobject.sourceforge.net.

Tasks

First release Month 18 for Amigo partners

Public release Month 20

Bugs

None so far

Patches

None so far

2.5.5.3 Amigo Core OSGi Bundle

Provider

France Telecom

Introduction

This bundle provides the Java interfaces and core classes that form the Amigo programming framework core: interfaces ExportFactory, BindingFactory, AmigoLdapLookup…

Remark: This bundle provides basic mechanisms for communication and service discovery, but is not linked with any protocol. It should be deployed together with implementing bundles related to communication protocols (binding factories and/or export factories) or service discovery protocols.

Development status

First version available

Intended audience

- Java developers that want to expose Java objects as remote services;

- Java developers that want to access to remote services;
- Java developers that want to write an adapter for a given technology.

License

LGPL

Language

Java

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

Hardware / OS: Any OS supporting Java personal profile or J2SE

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

The bundle appears on the bundle repository under a “bundle name” indicated in brackets.

- amigo_core.jar: bundle that provides the core interfaces and classes (bundle name amigo_core);
- hello_server.jar: test bundle that exports a simple Hello service (bundle name amigo_test_hello_server);
- hello_client.jar: test bundle that uses a Hello service using a well-known endpoint (bundle name amigo_test_hello_client);
- hello_lookup_client: test bundle that discovers the available Hello services and uses the first discovered (bundle name amigo_test_hello_lookup_client).
- test_pictureframe_server.jar: test bundle that provides a “picture frame” (bundle name amigo_test_pictureFrame_server) as an amigo service.
- test_pictureFrame_client.jar: test bundle for the amigo test picture frame server: this displays a graphical interface to choose an image from available images on the client’ file system to be displayed by the “picture frame” server (bundle name amigo_test_pictureFrame_client).

Documents

Java documentation, tutorial, developer’s and user’s guide are available on http://amigo.gforge.inria.fr/obr/ (also referenced at [Amigo-OSS-Pub]).

Tasks

Initial version Month 18 for Amigo partners

Public release Month 20

Bugs

None so far

Patches

None so far

2.5.5.4 Amigo kSOAP Export Factory Bundle

Provider

France Telecom

Introduction

This bundle allows making a Java object available through the SOAP protocol. It provides a local OSGi service that implements the ExportFactory interface. The choice of the kSOAP library allows this bundle to be deployed on constrained devices.

Development status

Initial release Month 18

Intended audience

Network administrators who want to use HTTP/SOAP as the base communication protocol should deploy this bundle on every OSGi platform that will provide remote services using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

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

Hardware / OS: Any OS supporting Java personal profile or J2SE

Software: kSOAP bundle, log4j bundle, HTTP service, servlet

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

amigo_ksoap_export.jar

Documents

Java documentation

Tasks

Initial version Month 18 for Amigo partners

Public release Month 20

Enhancements foreseen: WSDL generation

Bugs

None so far

Patches

None so far

2.5.5.5 Amigo kSOAP Binding Factory Bundle

Provider

France Telecom

Introduction

This bundle allows building a stub to a remote object accessible through the SOAP protocol. It provides a local OSGi service that implements the BindingFactory interface.

Development status

Initial release Month 18.

Intended audience

Network administrators who want to use HTTP/SOAP as the base communication protocol should deploy this bundle on every OSGi platform that will access to remote services using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

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

Hardware / OS: Any OS supporting Java personal profile or J2SE

Software: kSOAP bundle, log4j bundle

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

amigo_ksoap_binding.jar

Documents

Java documentation

Tasks

Initial version Month 18 for Amigo partners

Public release Month 20

Bugs

None so far

Patches

None so far

2.5.5.6 Axis Export Factory Bundle

Provider

France Telecom

Introduction

This bundle allows making a Java object available as a Web service. It provides a local OSGi service that implements the ExportFactory interface.

Development status

Under development

Intended audience

Network administrators who want to use HTTP/SOAP as the base communication protocol and provide WSDL service description may deploy this bundle on every OSGi platform that will publish Java object as Web services using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

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

Hardware / OS: Any OS supporting Java J2SE

Software: HTTP service, servlet, axis bundle provided by Knopflerfish

Platform

Java (J2SE), OSGi

Tools

None

Files

amigo_axis_binding.jar

Documents

Java documentation

Tasks

Initial version Month 20 for Amigo partners

Public release Month 22

Bugs

None so far

Patches

None so far

2.5.5.7 Axis Binding Factory Bundle

Provider

France Telecom

Introduction

This bundle allows accessing a Web service. It provides a local OSGi service that implements the BindingFactory interface.

Development status

Under development

Intended audience

Network administrators who want to use SOAP/WSDL as the base protocol for communication/service description may deploy this bundle on every OSGi platform that will access to Web services using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

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

Hardware / OS: Any OS supporting Java J2SE

Software: axis bundle provided by Knopflerfish

Platform

Java (J2SE), OSGi

Tools

None

Files

amigo_axis_export.jar

Documents

Java documentation

Tasks

Initial version Month 20 for Amigo partners

Public release Month 22

Bugs

None so far

Patches

None so far

2.5.5.8 SLP Bundle

Provider

France Telecom

Introduction

This bundle provides an implementation of AmigoLdapLookup based on SLP (Service Location Protocol).

Development status

Initial version based on mesh SLP (Columbia University) under test. Original library can be found at http://mslp.sourceforge.net/ .

Intended audience

Network administrators who want to use SLP as the base protocol for service discovery in Amigo may deploy this bundle on every OSGi platform that will access to SLP using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

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

Hardware / OS: Any OS supporting Java J2SE or personal profile

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

amigo_meshslp.jar

Documents

Java documentation

Tasks

Initial version Month 18 for Amigo partners

Public release Month 20

Bugs

None so far

Patches

None so far

2.5.5.9 UPnP bundle

Provider

France Telecom

Introduction

This bundle provides an implementation of AmigoLdapLookup based on UPnP.

Development status

First release available Month 24

Intended audience

Network administrators who want to use UPnP as the base protocol for service discovery may deploy this bundle on every OSGi platform that will access to UPnP using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

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

Hardware / OS: Any OS supporting Java J2SE or personal profile

Software: any implementation of OSGi UPnP base driver

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

Not yet available

Documents

Not yet available

Tasks

First release available Month 24

Bugs

None so far

Patches

None so far

2.5.5.10 WS-Discovery Bundle

Provider

France Telecom

Introduction

This bundle provides an implementation of AmigoLdapLookup based on WS-Discovery.

Development status

First release Month 24 for Amigo partners

Intended audience

Network administrators who want to use WS-Discovery as the base protocol for service discovery may deploy this bundle on every OSGi platform that will access to WS-Discovery using the Amigo core API.

This bundle is not used at development time.

License

LGPL

Language

Java

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

Hardware / OS: Any OS supporting Java J2SE or personal profile

Software : not yet known

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

Not yet available

Documents

Not yet available

Tasks

First release Month 24

Bugs

None so far

Patches

None so far

2.5.5.11 Amigo Service Binder

Provider

France Telecom

Introduction

The OSGi “declarative services” (formerly, service binder) allows to automatically manage the dependencies between services on the same OSGi platform, by defining a declarative language to describe dependencies and providing a bundle that instantiates service objects and manage dependencies using the OSGi discovery service. The Amigo Service binder will extend this abstraction to distributed services discovered through a Service Discovery Protocol.

Development status

First release Month 24 for Amigo partners

Intended audience

Developers who provide services that depend on other services.

License

LGPL

Language

Java

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

Hardware / OS: Any OS supporting Java J2SE or personal profile

Software : not yet known

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

Not yet available

Documents

Not yet available

Tasks

First release Month 24

Bugs

None so far

Patches

None so far

2.5.5.12 Semantic Adaptation Bundles

Provider

France Telecom

Introduction

This bundle will provide mechanisms for adaptation between a client requiring a service and a server providing a service “close enough” to that required by the client. They will provide the following functionality:

- Dynamic translation of component interfaces based on service matching description;

This bundle will ease the use of enhanced service discovery for OSGi programmers. It will highly depend on the Service Matching Tool and the Enhanced Service Discovery component (see [Amigo-OSSReport]).

Development status

Semantic adaptation bundle will be available at Month 30.

Intended audience

Application service developers that seek to dynamically discover and use heterogeneous services available in the environment.

License

LGPL

Language

Java

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

Hardware / OS: Any OS supporting Java

These bundles will use the service matching tool and the Enhanced Service Discovery service.

Platform

Java (version still to be determined), OSGi

Tools

None

Files

Not yet available

Documents

Not yet available

2.6 Amigo OSGi deployment framework

The goal of the Amigo OSGi Deployment Framework is to provide a Dynamic Service Deployment that takes into account the semantic description of services and the semantic description of the deployment itself to apply a semantically and timely local deployment strategy.

Usually, the deployment of services is decided statically after the development of the application. Before running an application, its components are deployed in an unchanging way. The innovation of our approach is that it provides a dynamic deployment that goes along with the dynamic nature of the environment and a semantic deployment that takes into account the nature of devices/platforms and the nature of the context present at a time being.

The semantic deployment functionality is provided by the Dynamic Service Deployment service (see Figure 2-3). Our service interacts with other high-level services of the middleware such as the Enhanced service Discovery that enables a semantic discovery and the Service Matching Tool that allows communication through semantically heterogeneous services. These high-level services rely on more classical services, the Discovery Service and the Interoperability Service for the discovery and the protocol transformation. Communication interfaces of these services are defined by a SAPI family, which is a set of possible interfaces such as Amigo interfaces or legacy interfaces (UPnP, etc).

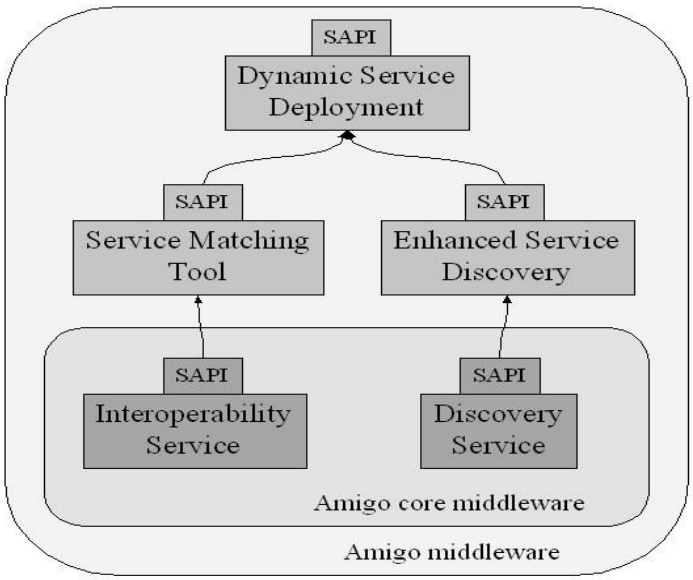


Figure 2-3: Dynamic Service Deployment service in the Amigo Middleware

The internal architecture of the Dynamic Service Deployment service:

- Service Container: the Service Container stores current services executing on the current platform. This container can be filled locally by the current platform or remotely by other Dynamic Service Deployment services.
- Deployment Strategy: the Deployment Strategy is in charge of deciding the deployment target, i.e. a software node on a remote host, and also the duration of the deployment.

2.6.1.1 **Dynamic Service Deployment service**

Provider

INRIA

Introduction

The Amigo bundle repository component offers two functionalities. It can upload a service using its reference or a semantic description from a specific URL or from an environment description. It can also download a service into a context using its reference or semantic description.

Development status

Not yet available. Development started in 2006.

Intended audience

The deployment framework is intended for component as well as to application developers.

Licence

LGPL

Language

Java

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

Hardware / OS: Any OS supporting Java personal profile or J2SE

Platform

Java (personal profile or J2SE), OSGi

Tools

None

Files

amigo_dynamicservicedeployment.jar

Documents

Java documentation

Component-specific documentation not yet available

Tasks

Semantic deployment bundles will be available in M30.

Bugs

None so far

Patches

None so far

3 Service description vocabulary ontologies

3.1 Introduction

This chapter presents a selected set of vocabularies that support semantic description of Amigo services. These vocabularies are based on the domain analysis presented earlier [Amigo-D3.1a].

The architecture and modularization principles for the developed vocabularies are introduced first (Section 3.2). The rest of the chapter presenting the vocabularies is structured based on this architecture (Sections 3.3 - 3.5).

The ontology diagrams presented in this chapter have been exported with the OntoViz plug-in of the Protégé ontology editor. Note that these diagrams only present the high-level structure of ontologies. Also in order to reduce the complexity of figures no data type properties are depicted.

A complete version of ontologies with online documentation is available at [Amigo-OSS-Pub]. The documentation also includes short guidelines for installing required tools and a getting-started document for using and developing the vocabularies. The Amigo OSS Repository - Source Code Management (SCM) [Amigo-OSS-SCM] supports the collaborative development work and releasing of these ontologies. More specific developer and user guidelines will be provided to help working with the vocabularies in the future.

3.2 Architecture and modularization principles of vocabulary ontologies

The main rationale behind the overall architecture is that vocabularies should support maintainability and future evolution of concepts related to the Amigo home. The modularisation of service description vocabularies is based mainly on the specificity of the concepts defined by the vocabulary and can be classified into three levels:

1. The concepts defined by a generic ontology are considered to be generic across many fields. Synonyms for generic ontology are "upper-level" or "top-level" ontology.
2. Core ontologies define concepts which are generic across a set of domains.
3. Domain ontologies express conceptualizations that are specific for a specific universe of discourse. The concepts in domain ontologies are often defined as specializations of concepts in the generic and core ontologies.

The borderline between core and domain ontologies is not clearly defined because core ontologies intend to be generic within a domain. In this classification, the service modelling vocabularies described here can be classified mainly into the last two categories.

The vocabularies have been developed using OWL language. Its import mechanism is used to enable references on concepts from more generic ontology modules by more specialised ontologies. An ontology module can be specialised by subclassing the concepts defined in the imported file. Separate namespaces are used to prevent naming conflicts. The import hierarchy of the Amigo Service Description Vocabularies is presented in Figure 3-1.

The domain vocabularies are extendable modules that will provide detailed information about technologies and features of a particular class or model of a device in Amigo home. New domain vocabularies can be added to cover more device manufacturers when needed. Any concepts from the more high level vocabularies can be specialised. For example, new device types can be defined as subclasses of Core Domain Device concept, and specific models introduced as individuals (instances) of device classes.

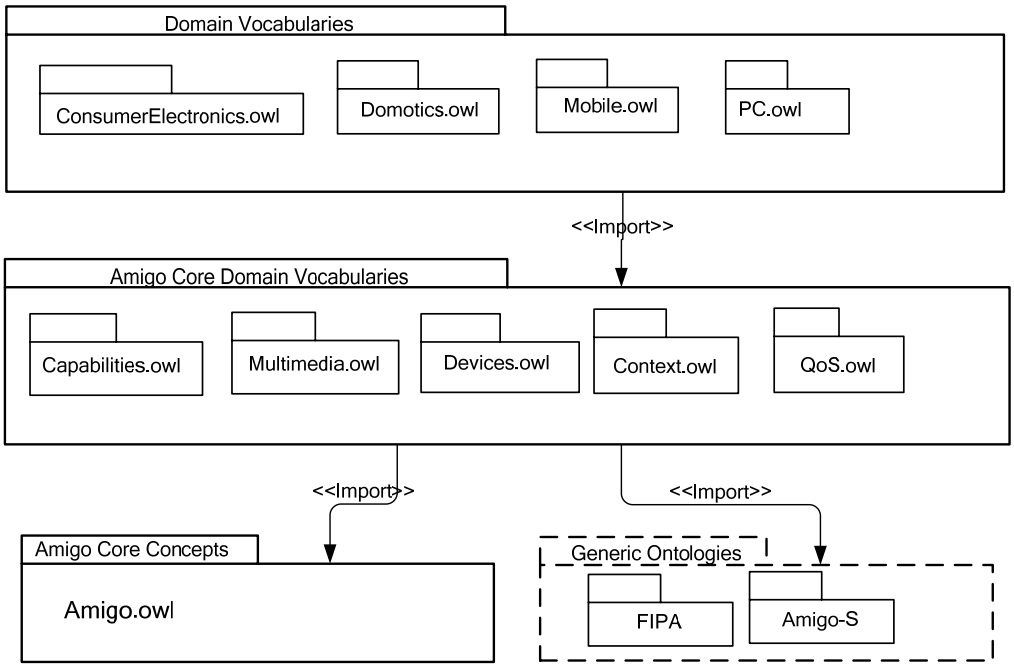


Figure 3-1: Architecture of Service Description Vocabularies

The main part of vocabularies belongs to the Amigo Core Domain vocabularies. These vocabularies may also evolve and commonly used concepts from Domain Vocabularies can be generalised and added into one of these vocabularies. Generic ontologies and vocabularies such as FIPA device ontology for various domains may be imported or adapted by these ontologies. Also the Amigo Service Description language ontology (Amigo-S) (see Chapter 4) may be one of the imported ontologies.

The Amigo Core Concepts provide a classification of concepts selected from Amigo core domain vocabularies that have important cross-domain relations. This reduces the need for core domain vocabularies to import each other improving the maintainability of vocabularies.

This high-level architecture supports the evolution of domain vocabularies with introducing emerging new technologies needed in Amigo or any other home environment. To achieve explicitness and modularity, normalisation criteria have been proposed in the literature [Rec03] for implementations of description logic related to domain ontologies. The so-called primitive skeleton of domain is based on the following criteria:

1. The branches should form trees, i.e., no domain concept should have more than one primitive concept as parent.
2. Each branch should be homogeneous and logical, i.e., the principle of specialisation should be subsumption (i.e., a concept is a specialisation of another).
3. It should clearly distinguish "self standing" concepts from "partitioning" or "refining" concepts.
4. The axioms, range and domain constraints should never imply that any primitive domain concept is subsumed by more than one primitive domain concept.

As opposed to these primitive concepts, there are so-called "defined concepts" defined by "necessary and sufficient" description logic conditions. The subsumption of such defined concepts can be left to the reasoner and should not be defined by the ontology developer.

In order to achieve this kind of normalisation, the analysis of Amigo domain vocabularies has been mainly based on finding the skeleton of primitive concepts or "taxonomy" for the domains [Amigo-D3.1a]. Guidelines for vocabulary developers, which present an analysis of an example domain, are currently used internally by Amigo partners [Amigo-OSS-SCM]. However, in practice it has been difficult to express some defined concepts with sufficient and necessary logical conditions, so manual classification of them has sometimes been provided. Multiple inheritance of concepts defined in the most top-level ontologies has also been used as a mean to provide a common conceptual base for the otherwise unrelated domain and core ontologies. Also the distinction required by criterion 3 has not yet been applied fully. However, these criteria provide good general guidelines for the future modularisation of vocabularies.

3.3 Amigo Core Concepts (Amigo.owl)

Amigo core concepts define the basic vocabulary that helps to tie the other vocabularies together. To keep the vocabularies modular, the vocabularies should avoid unnecessary references and use the core concepts when possible. The vocabularies can use the classes and property types in Amigo core vocabulary either directly or use sub-classing of classes and properties. The sub-classing of property types is the useful way for creating restrictions on domain and range of relations between individuals. The Amigo core concepts are presented in Figure 3-2.

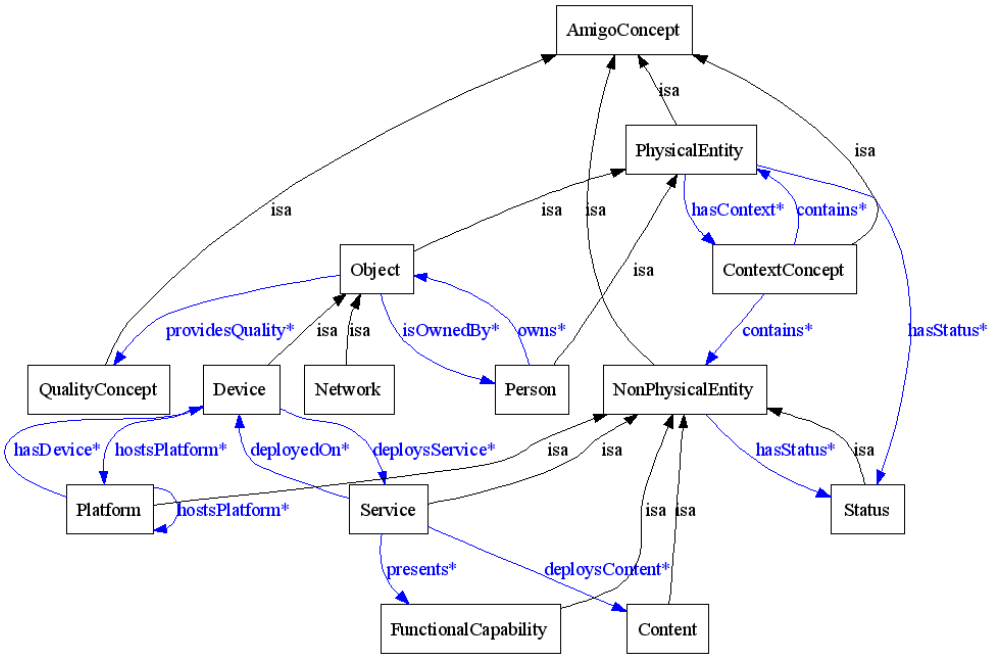


Figure 3-2: Amigo main core concepts

The PhysicalEntity class is the class that incorporates all the living and non-living entities inside the Amigo Home environment. The class is defined within the base Amigo vocabularies, as it is considered to be one of the system's core entities. Person is representing each physical person as part of the physical world. A Person object can either be a User of the Amigo system or simply a non-user that has entered the Amigo home environment. The Object class aims to represent all the living and non-living entities of the Amigo home environment that do not belong to the Person class.

The NonPhysicalEntity class incorporates abstract software related entities such as content, services and platforms hosted by devices. The Amigo Service is a focal entity in the Amigo system for service modeling. The language for service description is presented in Chapter 4.

A high-level context classification is also presented to facilitate specialization of context concepts for the requirements of different Amigo domains. Extending the approach in [Amigo-D2.1], six core context domains have been identified that may potentially be used to build a complete information model of the conceptual and physical world. These context domains are briefly described below (see also Figure 3-3):

- User Context Domain. User is the class incorporating the context attributes related to the Amigo user. It includes the physical characteristics of persons, their personal information, their preferences, etc.
- Physical Context Domain. The classes of this domain aim to represent the physical parameters of the Amigo environment. These parameters include physical living and non-living objects, environmental, spatial and temporal properties, and have been introduced in order to cover the plethora of physical context information. Such parameters model, among others, the physical location of the user and the device, their temporal settings and the environment they exist in.
- Device Context Domain. This domain includes the Amigo devices that operate inside the Amigo home. It represents devices from the home automation, the consumer electronics, the mobile communications and the personal computing fields.
- Service Context Domain. The classes of this domain model the services that are provided in the Amigo home environment.
- Network Context Domain. This domain represents all network related information. It is important as almost all devices of the Amigo environment are depending on extended networking and interconnection capabilities.
- Social Context Domain. The classes of this domain represent the social relationships of the various Persons inside the Amigo home environment. Examples of such relationships are: husband-wife, parent-child, student-professor, employer-employee, friends, colleagues, etc.

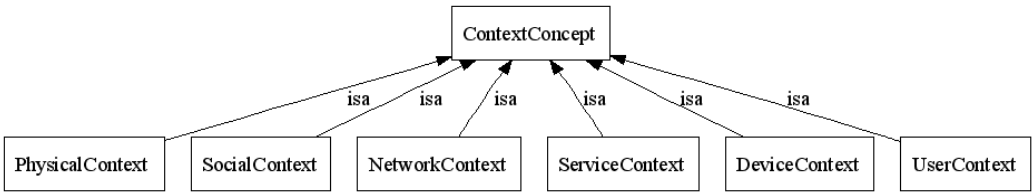


Figure 3-3: The high-level classification of context-related concepts

Functional capabilities (see Figure 3-4) model abstract capabilities provided by a software service for another service or a human. A basic classification for functional capabilities is given based on the Amigo application domains. This kind of high level classification is usable mainly for service management purposes to identify the services available in Amigo home (this classification is extended and the link with service description languages discussed in the Capabilities vocabulary module).

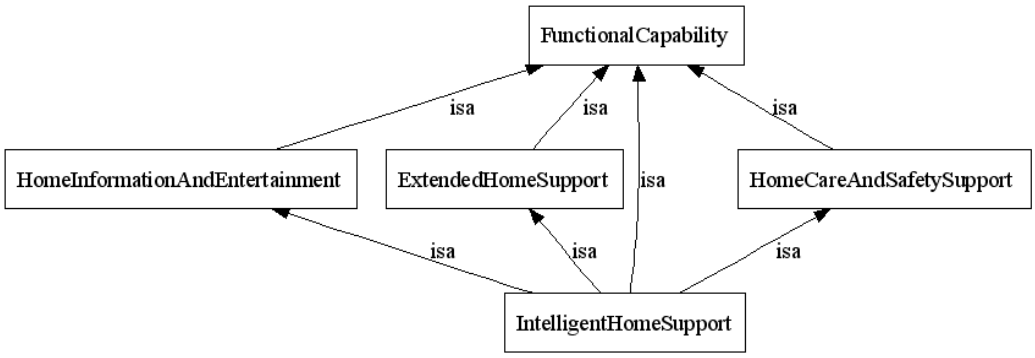


Figure 3-4: Classification of functional capabilities based on Amigo application domains.

A simple content classification vocabulary (see Figure 3-5) is also provided at core level for different types of content. The main concepts in this classification are ContentClass, ContentMetadata and ContentResource.

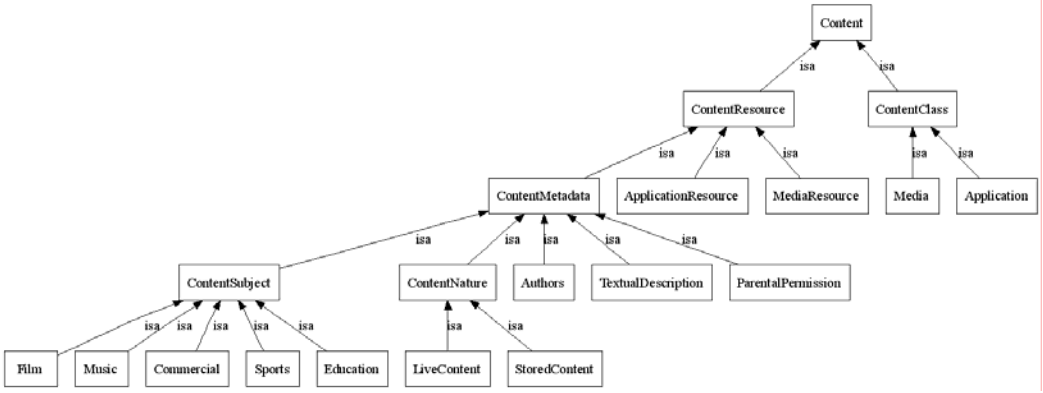


Figure 3-5: Content vocabulary

3.4 Amigo Core Domain Vocabularies

3.4.1 Devices and platforms (Devices.owl)

The main classification of generic platforms and devices are combined into the platform and device vocabularies (see Figure 3-6 and Figure 3-7).

The device vocabularies provide a classification on platforms hosted by devices and generic classification of device types and their states. Note that the hasDeviceStatus is a subproperty of hasStatus property defined in Amigo core concepts.

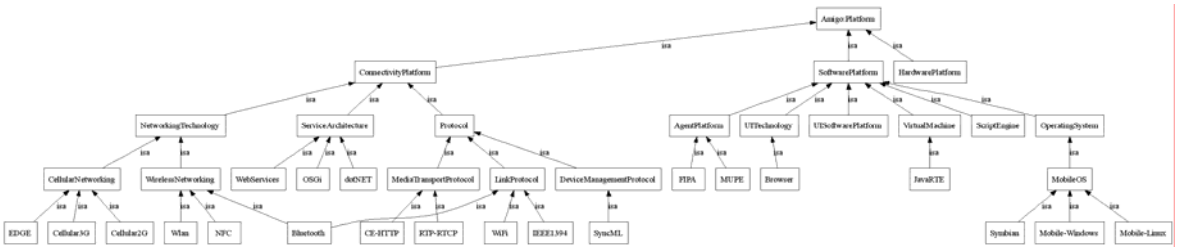


Figure 3-6: The platform vocabulary

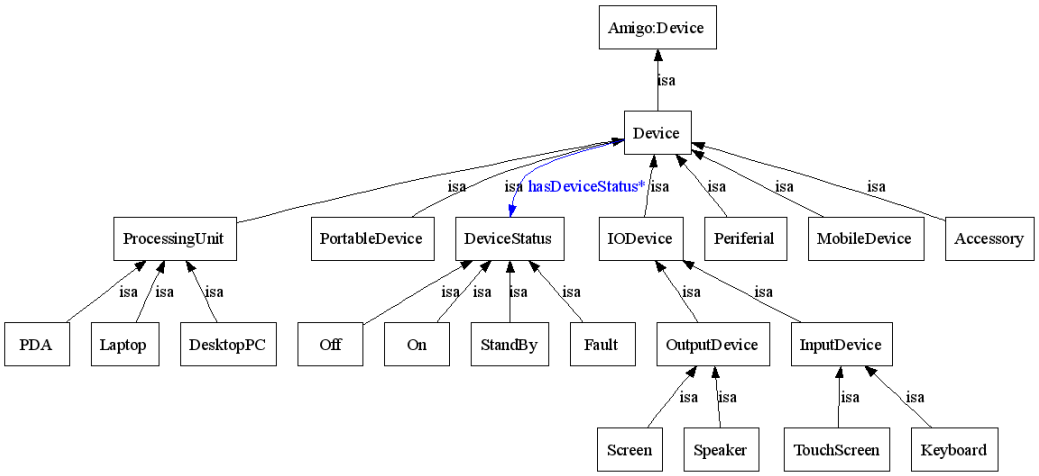


Figure 3-7: The device vocabulary

It must be noted that although these concepts can be used to characterize the device context, these vocabularies are still very generic. The integration of descriptor structures from more specific ontology languages such as FIPA device profiles can be done for example by defining the FIPA compatible device related quality concepts as subclass of Amigo:QualityConcept (Figure 3-8). However, details of this are still under consideration in collaboration with the development of QoS, Multimedia and Consumer Electronics vocabularies.

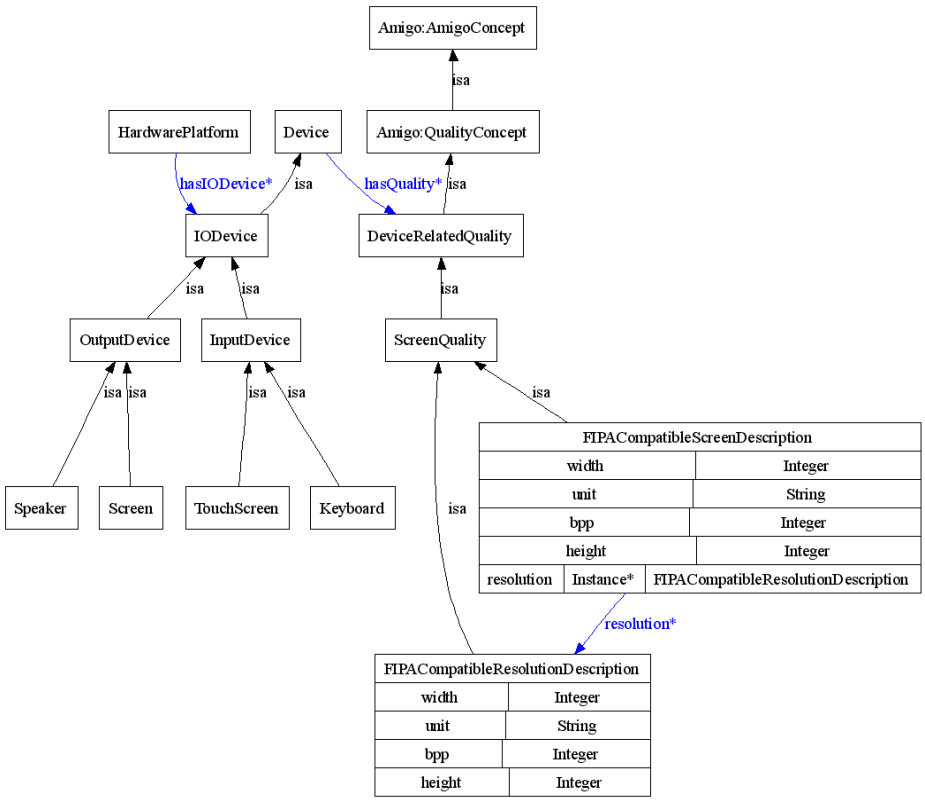


Figure 3-8: An example of how to integrate FIPA compatible screen descriptors to the Amigo device vocabulary

3.4.2 Functional capabilities (Capabilities.owl)

These vocabularies refine the functional capability concept defined in Amigo Core Concepts. Functional capabilities can be classified using the classification provided by the Amigo core vocabulary.

To help linking functional capabilities with semantic software service description languages this classification is extended by two new concepts: ServiceProfileCapability and ServiceCapability. These concepts classify the roles that a functional capability can have when presented by a software service. This classification enables reasoning on service profile hierarchy and capabilities provided and required by the service profiles during the semantic service discovery.

- The hierarchy under ServiceProfileCapability (see Figure 3-9) classifies the functional capabilities that can be used to classify profiles presented by software services. This includes the set intelligent services provided by the upper layer of middleware.

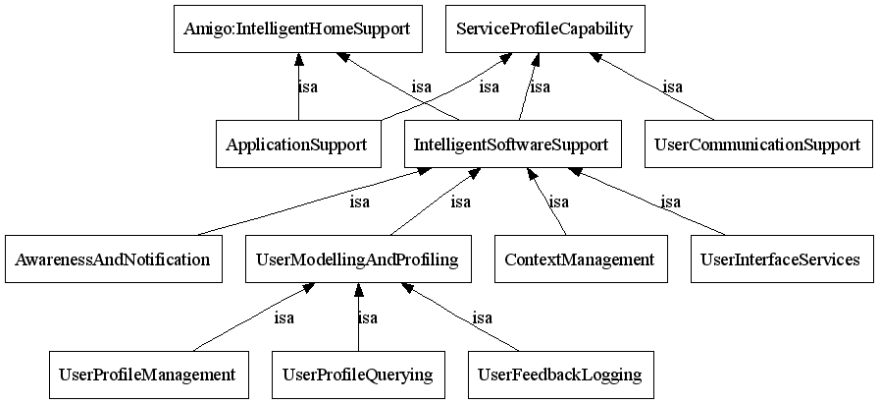


Figure 3-9: Part of ProfileCapability hierarchy

- The hierarchy under ServiceCapability (see Figure 3-10) classifies the functional capabilities provided or required by a software service and presented as part of its profile.

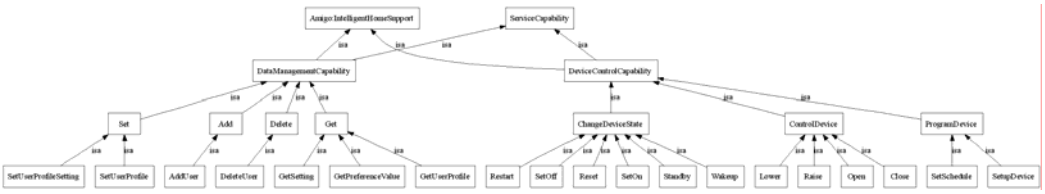


Figure 3-10: Part of ServiceCapability hierarchy

An example of how these two concept hierarchies could be linked in a service description language is given in the following Figure 3-11.

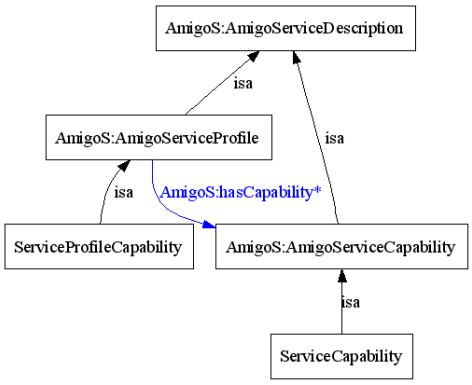


Figure 3-11: An example of simple semantic service description language

3.4.3 Quality of Service (QoS.owl)

This section presents the Quality of Service vocabulary refined ontology. It is based on the work described in [Amigo-D3.1a], and has been slightly updated in order to address the special requirements of the Amigo scenarios and applications. The QoS language ontology introduced in [Amigo-D3.1b] has not been updated and is also briefly discussed in Chapter 4 of this document.

The Quality of Service concept hierarchy is depicted in Figure 3-12. Notice that in order to reduce the complexity of this figure, no data type properties are depicted. Furthermore, it should be mentioned that the maximum height of the relevant hierarchy tree is two. This is preferable for reduced complexity development.

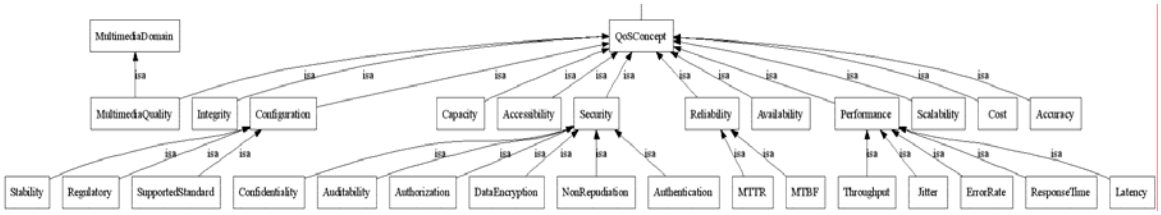


Figure 3-12: QoS concept hierarchy

- The QoS vocabulary ontology is incorporated into the Amigo vocabulary ontology, as the QoSConcept class has been introduced as a subclass of the AmigoConcept class. The QoSConcept class is the superclass of all QoS parameters defined in Amigo. The respective subclasses are also shown in Figure 3-12.
- MultimediaQuality is an example of a domain specific QoS concept. It concerns exclusively the QoS of multimedia services and may be part of multimedia ontology module in the future.

3.4.4 User context (Context.owl)

The user domain context ontology considers the requirements of the Amigo home environment and attempts to model all parameters that may potentially be related to the Amigo user. In this initial approach an effort has been made to provide a complete context ontology for the user domain, which probably addresses more context parameters than those involved in the Amigo

scenarios. However, whether this approach is best suited for Amigo or a less sophisticated context ontology version is more appropriate is still to be decided.

- The User class is the centric class of this domain. It represents humans that make use of the Amigo system and holds relationships to all main classes in the user domain context ontology. The User class is also related to classes identified in other context domains (i.e., service and device domain), to represent the cases he is using a specific device or service. There can also be multiple object properties relating two users. These properties represent relationships such as: studentOf, friendOf, motherOf, employeeOf, colleagueOf, etc.

The User Context Domain Vocabulary Ontology is depicted in Figure 3-13. Notice that the object properties carried by some User Context Domain Ontology classes towards classes outside this domain are also illustrated. In order to reduce the complexity of this figure, no datatype properties are depicted.

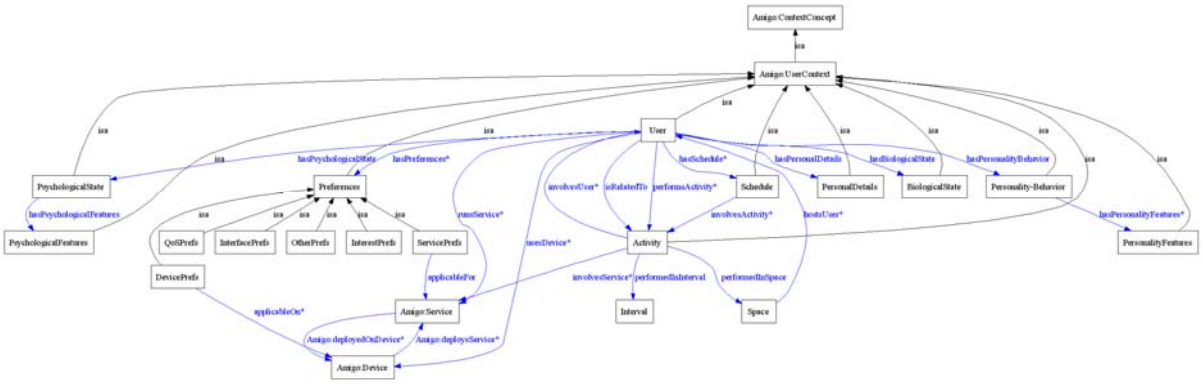


Figure 3-13: The User Context Domain Vocabulary Ontology

3.4.5 Physical context (Context.owl)

In this section the physical context domain vocabulary ontology is presented focusing on the Amigo home specific issues. This ontology incorporates the generic parameters that are related to the elements of the physical environment. It consists of four independent sub-ontologies: Core, Spatial, Temporal, and Environmental Domain ontologies.

In Figure 3-14, the Core Physical Context Domain Vocabulary Ontology is depicted. Notice that the object properties carried by some classes in this ontology towards classes outside this domain are also illustrated. Again, in order to reduce the complexity of this figure, no datatype properties are depicted.

The Space class of the spatial domain context ontology (see Figure 3-15) is an abstract class that corresponds to any physical place.

- AbsoluteLocation represents the physical location of the user in terms of Longitude, Latitude and Altitude. Based on these attributes we can locate any context entity having physical substance. Inside an AmI featured home it is necessary to locate objects based on their relative locations with regards to a specific reference point.
- ReferenceSystems are tailored on the location and shape of selected physical objects in a specific Area. Each object of the ReferenceSystem class is defined by three data properties that indicate the directions of the three orthogonal reference axes (x,y,z). Each physical object may have multiple relative locations with respect to the reference systems defined.

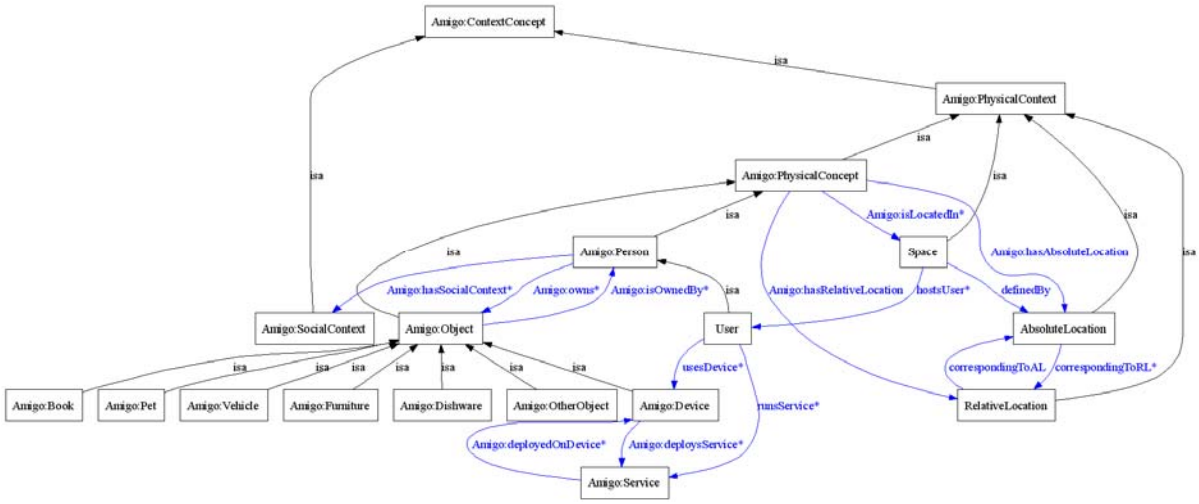


Figure 3-14: The Core Physical Context Vocabulary Ontology

- These are expressed by instances of the RelativeLocation class that uniquely define a physical object's position inside the Amigo home. Note that RelativeLocation is related to the AbsoluteLocation via two symmetric relationships that are used to express RelativeLocation in AbsoluteLocation coordinates and vice versa. The Distance class is introduced to represent the physical distance between two physical absolute or relative locations.
- The Area class is modeled as a subclass of Space and represents a physical area located inside a City or a Country. It may correspond to an indoor or outdoor area.
- The Building class is used to represent a physical building. A Building class is the superclass of many different buildings such as CompanyBuilding, Home, Cinema, Gym, etc. The Room class is a subclass of the Area class and corresponds to a room located inside a building. The Room is critical for all Amigo home based scenarios. Various subclasses of the Room class have been identified here, such as Bedroom, Bathroom, Kitchen, LivingRoom, Office, etc., modeling the variety of the possible rooms of the Amigo Home.

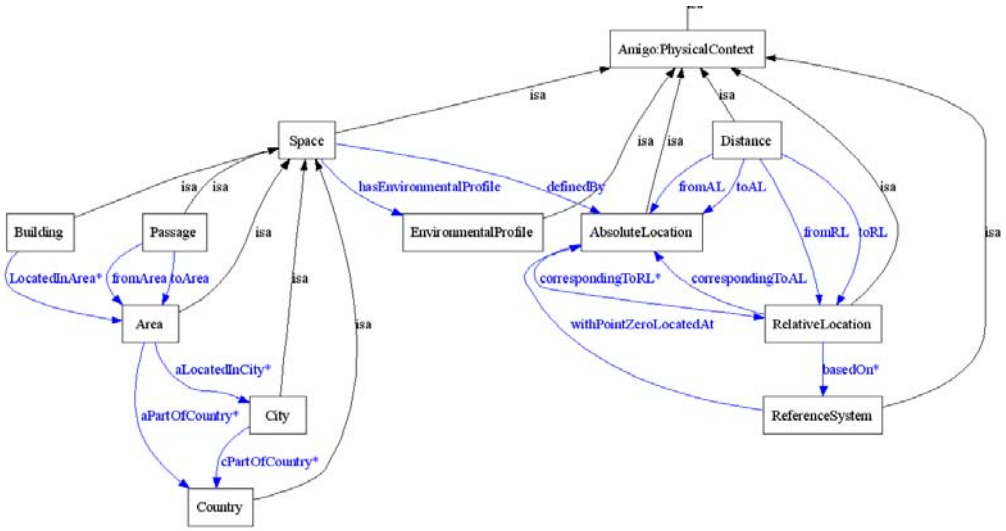


Figure 3-15: The Spatial Context Domain Vocabulary Ontology

- The Passage class represents all the passing areas that can be identified in any building or spatial area and do not belong in any of the other modeled classes. Such areas are the stairs, the corridors, entrances, exits, highways, avenues, etc, that are introduced as Subclasses of the Passage class.

The main concepts in Temporal Domain are Time and Interval (see Figure 3-16).

- The Time class is a collection of the temporal parameters that define a specific moment in time. It contains attributes such as TimeZone, Hour, Minute, Second, Day, Month, Year, etc. These attributes are necessary for scheduling and synchronizing.
- The Interval class represents a specific period of time and is related to the Time class.

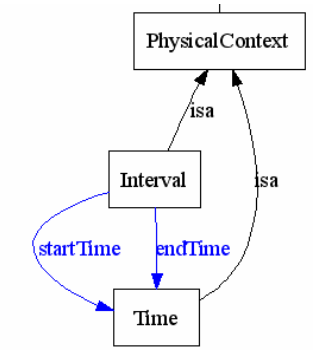


Figure 3-16: The Temporal Context Domain Vocabulary Ontology

- The EnvironmentalProfile (see Figure 3-17) incorporates various datatype properties such as: Temperature, Humidity, Pressure, WindSpeed, Visibility, Noise, Illumination, etc.

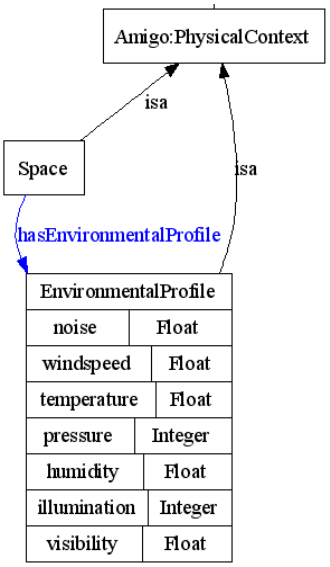


Figure 3-17: The Environmental Context Domain Vocabulary Ontology

3.4.6 Multimedia (Multimedia.owl)

Multimedia vocabulary describes the different contents that can be processed by the devices in an Amigo home. It consists of four classes that are explained below. The development of a

multimedia domain ontology that fully captures the content the devices can display is very helpful in order to both compose the adaptation services to integrate the content in the devices and to help Amigo home designers to make their decisions about the content their devices are able to reproduce. Multimedia ontology is further visualized in Figure 3-18.

In order to explain the content that will be displayed by the devices, the ontology integrates four classes:

- The MultimediaContent class describes the type of content the user (human or not) can access. It includes different kinds of contents such as video, audio and image. Several resources may be linked to the content.
- The MultimediaResource class models resources linked to a specific content. Because content can be deployed into different platforms it needs more than one resource to be successfully rendered. We can differ among Audio resources, Video resources and Still Image resources depending on the type of content the device is capable to handle.
- The ResourcesProperties includes information related to the format (and quality) of used multimedia content in particular situation/context. For example, we must consider properties such as bitrate, coding, resolution or media format. These properties will take different values depending on the type of content they are being related to. It is modelled as subclass of Amigo:QualityConcept.
- The Content Metadata Class contains non-technical information as the vendor, name and version of the multimedia content that is not needed for the devices to handle the specific content as is the case with ResourcesProperties.

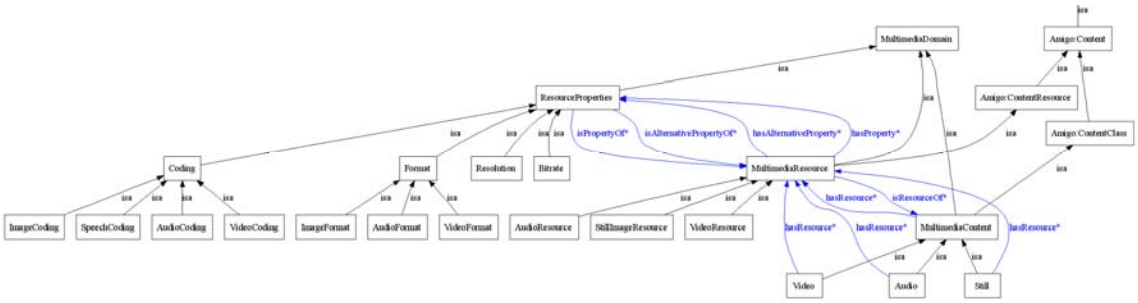


Figure 3-18: Multimedia Domain Vocabulary

3.5 Amigo Domain Vocabularies

3.5.1 Domotic domain (Domotics.owl)

The domotic domain in Amigo is related to identifying the requirements of home automation devices to Amigo system. Some examples of such devices could be a lighting system, a washing machine, a gas sensor, etc. Domotic domain focuses on classifying different types of domotic devices states and the capabilities they provide extending the Device and Capabilities domains. There are several device types that can be met in the domain:

- Sensors can either detect environmental changes or measure environmental conditions, providing relevant information. The first sensor category can notify about such changes, while the second provides specific values.

- Actuators, on the contrary, can interact with the environment changing some conditions. There are several types of actuators. We can identify switches, regulators, valves and engines.
- Appliances or white goods are another subset of domotic devices. Some of them can be scheduled (set a start and end time, the temperature of the oven according to a recipe, the washing machine program etc.). The device-related part of the domotic domain classification (i.e., extension) is depicted in Figure 3-19.

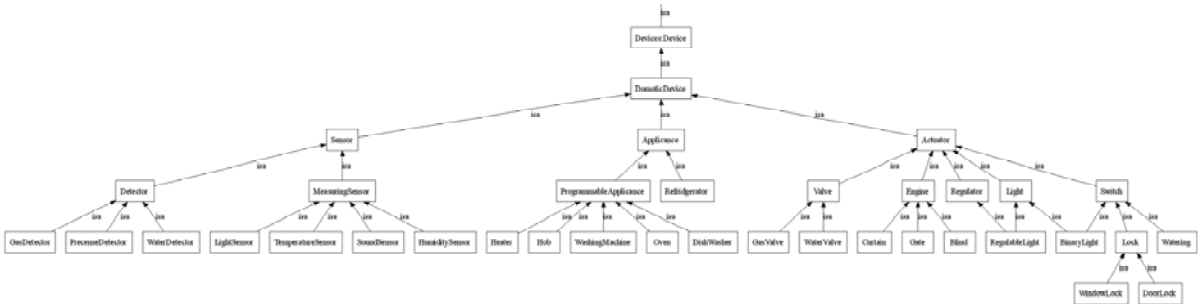


Figure 3-19: Domotic devices

Domotic bus technologies can be modelled as individuals of domotic bus specifying the supported protocols and physical layer technologies (Figure 3-20).

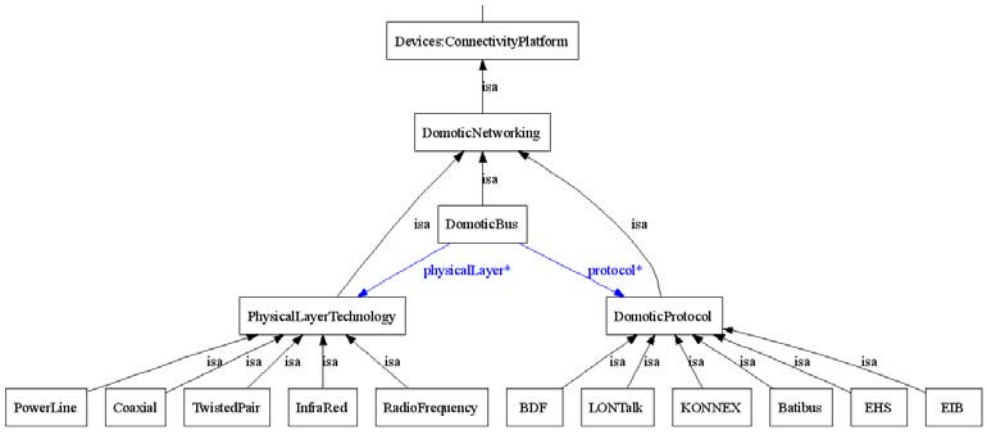


Figure 3-20: Domotic bus technologies

The rest of the concepts modeled in domotic domain include the status, events and capabilities related to domotic devices. DomoticVariables are measured by sensor devices and are related to PhysicalContext (Figure 3-21).

3.5.2 Consumer Electronics domain (ConsumerElectronics.owl)

In this section, the Consumer Electronics Device vocabulary is presented. In this initial approach, an effort has been made to state clearly the different types of devices used in Amigo. The consumer electronics devices are classified into three subgroups based on the type of the media they can handle.

- AudioDevice represents devices that are able either to reproduce or store audio with all the characteristics that it involves. Thus, we can differ between AudioCaptureDevice or AudioRenderDevice depending on their final purpose. Some examples of the AudioCaptureDevice are the Microphone or the Speech Recognition Device, and some

examples of AudioRenderDevice are the Barebone PC, the Speaker and the Speaker set.

- GamingDevices represents devices for gaming.
- StillImageDevice represents devices that either render or store images. It includes the Television, the PDA, the mobile phone or all kind of PCs into the StillImageRenderDevice subclass or the Digital Camera in the StillImageCaptureDevice subclass.
- VideoDevice represents devices that, in a similar way to the previous ones, are able to reproduce or store video. Television, PDA, TabletPC, WebCam and Digital Camera are subclasses in this group.

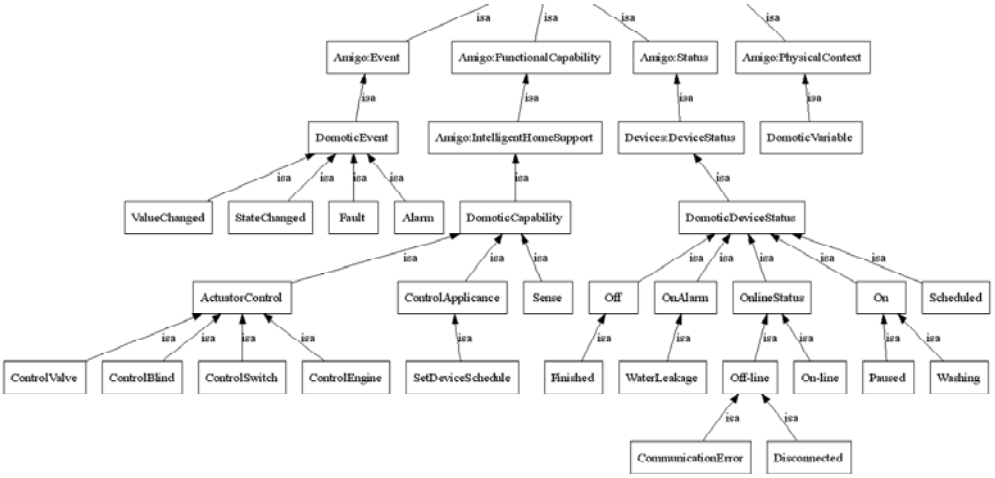


Figure 3-21: Remaining parts of domotic vocabulary

Many of the modern devices belong to several of those subgroups. For example, digital photo cameras are included in both StillImageDevice and VideoDevices as it can also record live video. This is taken into account in the Consumer Electronics vocabulary.

All the different devices described share some characteristics inherited from the main Amigo device class. Connectivity platform, software platform, user interfaces, discovery protocol, memory, CPU, capabilities, user Interfaces and location are object properties that must be linked with other parts of the main ontology. The “User Interfaces” that takes different forms depending on the type of device it is characterising, is one example of essential attribute for the Amigo home demonstrator.

The “state” attribute is an object property as it is linked to a class State that must contain a description of the current memory and CPU load. “Codecs” is referred to a set of instances of installed specific software modules allowing the acquisition/renderization of certain coding standards. “Capabilities” attribute refers to the services offered by the device when processing a particular type of content.

There is also a relationship between the Consumer Electronics Device Ontology and the Multimedia Ontology as the content has to be expressed in a format the devices have to understand. Therefore, the “format” attribute will be the property that links both ontologies.

The high level classes of the Consumer Electronics Device ontology are depicted in Figure 3-22.

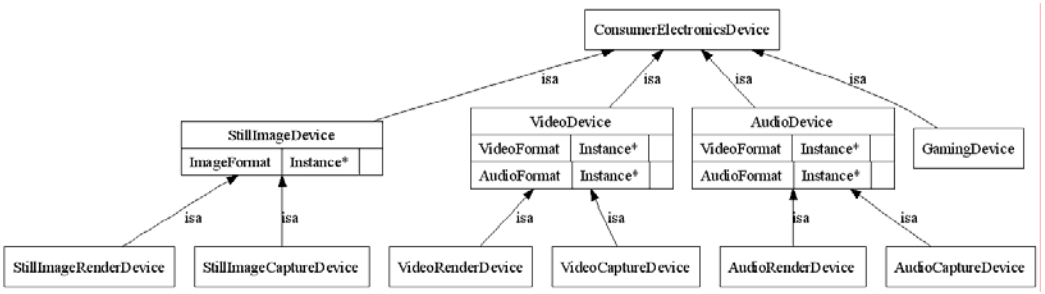


Figure 3-22: High level CE Device ontology

3.5.3 Mobile domain (Mobile.owl)

Mobile domain focuses on vocabularies related to various mobile platforms. Examples of such mobile devices are mobile phones and PDAs but also laptops, smartcards, etc. (see Figure 3-23). Specific models mobile devices can be defined as individuals of a device class. Mobile phones provide simple classification of environment profiles that can be understood as simple classification of environmental context in which the device has to be adapted into.

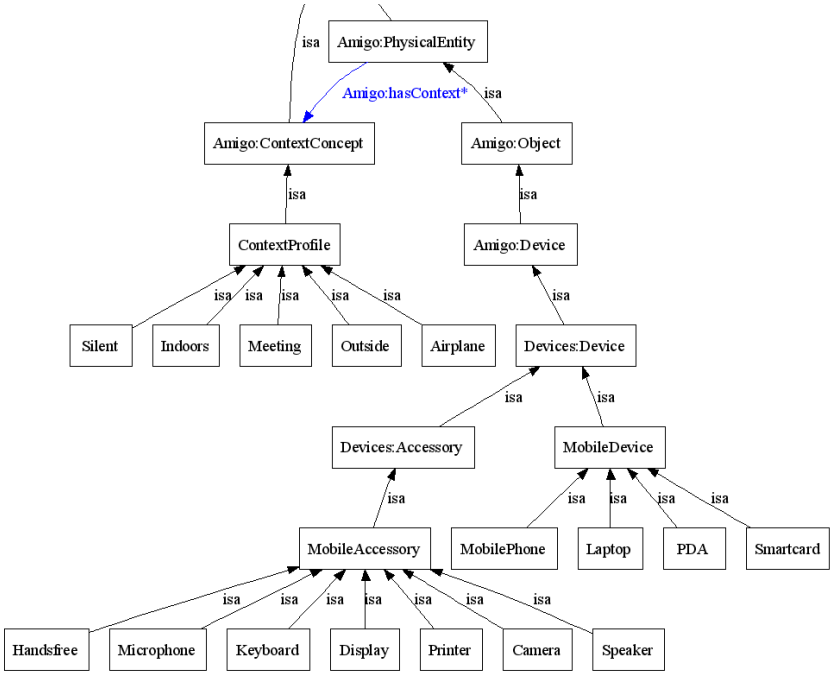


Figure 3-23: High level mobile device vocabulary

3.5.4 Personal Computing domain (PC.owl)

The PC domain is related to the “classical” view that we have about computers. Personal computers, Web cameras, peripherals such as printers, scanners, etc, are included in this domain. The nature of the resources of these devices can widely vary (e.g., in terms of connectivity, processing power, UI etc.). The role of PC domain in our every-day life is mainly related to storing, accessing and processing information. Relevant data may range from management information such as individual home preferences to access rights to multimedia

content. Because of its generality most of the vocabularies analysed from PC domain have been included into the generic device domain classification. However, a separate module for PC domain vocabulary is reserved here for future inclusion of information about specific device models.

4 Service description language, aspects of service discovery

This chapter presents the formal definition of the Amigo semantic service description language, which we call *Amigo-S*, and elaborates on a set of aspects of Amigo-aware service discovery; both of these topics were introduced in the previous deliverable D3.1b [Amigo-D3.1b]. Semantic specification of services is needed for defining 'Amigo-aware services', which enable enhanced, Amigo-aware, service discovery based on semantic matching of the functional and non-functional properties of services. Service matching makes use of the semantic knowledge associated with services, which may otherwise be syntactically different, thus increasing service availability. Semantic specification of services further allows interoperability mechanisms to be realized between heterogeneous services in the Amigo home.

Amigo-S is based on OWL-S, which is extended to support descriptions of Amigo services. Key features of Amigo-S are that it describes both the functional and non-functional properties of provided services in the Amigo environment, as well as their underlying middleware communication mechanism. Functional properties are given independently of the underlying middleware. It is further possible to specify how the discovery of the provided service and the interaction with it are enabled, by associating the service to a concrete grounding, i.e., an interaction protocol and a service discovery protocol. This makes possible to assess and enable the interoperability of two services whether they are designed for the same service technology (e.g., Web Services) or not. Finally, non-functional properties include context and QoS and may apply to both the service and its underlying middleware.

Based on Amigo-S, our aim is to provide tools that can be used for enhanced service discovery, supporting checking the conformance of services, i.e., their capacity to interoperate. These tools can be realized by integrating semantic reasoners and specific algorithms for matching services based on their semantic descriptions. Currently, we have studied several issues relative to these tools; however, we have not yet integrated Amigo-S into service discovery.

Hence, in this chapter, we first introduce the formal definition of Amigo-S, and describe how to employ formal tools for using it (Section 4.1). Then, we examine several aspects of enhanced service discovery, including: evaluation of existing tools for semantic reasoning and service matching and proposition of a matching tool appropriate for Amigo; context-aware service discovery; and service selection based on QoS information for optimizing resource consumption (Section 4.2). We conclude with a short discussion on the principal points of this chapter (Section 4.3).

4.1 Service description language

In this section, *Amigo-S*, the service description language for semantically describing Amigo services, is presented. Extensions to OWL-S for enabling specification of Amigo services were discussed in Deliverable D3.1b, without giving the exact relationships between OWL-S classes and the newly introduced classes. Based on that informal description of the language, we introduce here its formal definition in OWL. We first discuss its relationship with the OWL-S language (Section 4.1.1), and then present the description by the language of service functional (Section 4.1.2) and non-functional (Section 4.1.3) properties.

The current version of the complete Amigo-S specification in OWL is available, for the moment in a restricted way, on the Amigo OSS Repository - Public Web Site [Amigo-OSS-Pub]. Further accompanying material can be found there:

- An – incomplete for the moment – developer's guide document;
- Help – HTML page style – documentation produced in an automatic way with OWLDoc for the OWL specification of the language.

4.1.1 General properties of the language

Amigo-aware services are provided with semantic specifications enabling interoperability mechanisms to be realized between heterogeneous services in the Amigo home. Amigo-S is introduced for supporting such specifications, and is intended to be used by dedicated tools capable of performing formal reasoning on such specifications.

Amigo-S is based on the OWL-S language [OWL-S]. As discussed in deliverable D3.1b, OWL-S cannot be used as-is for describing Amigo-aware services for several reasons. First, 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. We, thus, extended OWL-S by enabling several groundings to be employed for a service. To this end, Amigo-S supports specifying the underlying middleware of deployed services. The second reason is that OWL-S lacks support for describing context and QoS-related information, which are key non-functional properties that we want to describe for Amigo-aware services. We included in the language generic classes for describing such non-functional properties. Another feature that is needed is the possibility to determine these properties globally for all the functionalities that an Amigo-aware service provides, as well as individually for each functionality. We, thus, extended OWL-S so that these properties could be expressed at different levels.

The OWL specification of Amigo-S extends the OWL-S language using the import mechanism of OWL, and defines new classes for describing the functional and non-functional properties specific to Amigo services. Furthermore, Amigo-S defines classes for specifying the underlying middleware on top of which services are to be deployed and will interact, complementing the WSDL grounding of the OWL-S specification.

Amigo-S reuses classes that are already formally specified as part of OWL-S; our aim is 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. This allows as well easy adaptation of existing service descriptions written in OWL-S for making them Amigo-aware. Indeed, an OWL-S service description is always compatible with the Amigo language ontology.

The formal specification of the Amigo-S language is given in the OWL DL sub-language of OWL to enable using existing OWL reasoners, which support OWL DL. In an OWL DL ontology, all entailments are guaranteed to be computed and all computations will finish in finite time. We have performed verification of our ontology specification for both correctness and conformity to OWL DL by employing the Pellet OWL DL reasoner, which is freely available as open-source software [Pellet] (in Section 4.2.1.1, Pellet is evaluated along with other existing reasoners for their suitability to be employed in Amigo service matching).

The Amigo-S language is presented in this chapter with OntoViz diagrams, which is a plug-in for the Protégé OWL editor. In the diagrams, the following namespaces are used:

- The default namespace refers to the Amigo-S language;
- p1: OWL-S Profile ontology (http://www.daml.org/services/owl- s/1.1/Profile.owl);
- service: OWL-S Service ontology (http://www.daml.org/services/owl-s/1.1/Service.owl);
- process: OWL-S Process ontology (http://www.daml.org/services/owl-s/1.1/Process.owl);

- expr: OWL-S expressions (http://www.daml.org/services/owl-s/1.1/generic/Expression.owl).

An Amigo service is described using the OWL-S profile class, which is used to define global properties of the service. A specific functionality offered by an Amigo service is further called a *capability*, and defined using as many capability classes as needed. The functional properties of a service capability are further described by its pre-conditions, effects and its inputs and outputs (Section 4.1.2). Non-functional properties of a service are either described globally or for each service capability by the service context and service QoS parameters (Section 4.1.3).

4.1.2 Description of service functional properties

In OWL-S, a service is defined using the Service class (see Figure 4-1), which is associated to three types of definitions. First, a service presents one or more Service Profiles [Profile], which describe functional and non-functional properties of the service. Second, the service is described by a Service Model [Process], which gives the supported conversations of the service, i.e., the correct invocation sequences of the service. Third, the Service Grounding [Grounding] enables the mapping of the Service Profile and the Service Model onto the underlying middleware.

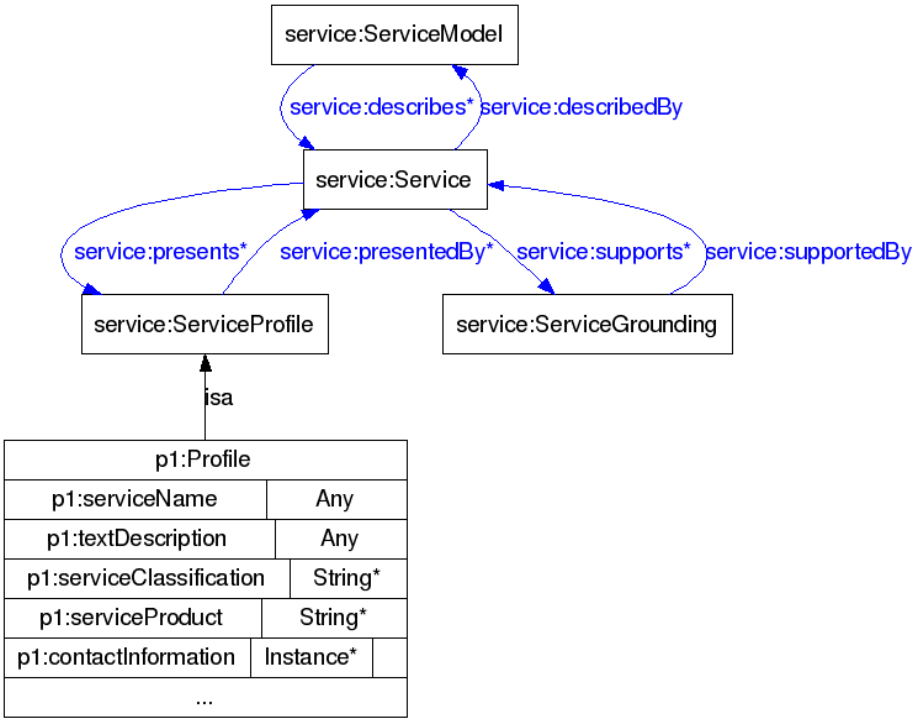


Figure 4-1: OWL-S top level ontology

In Amigo-S, a service is described by one or more OWL-S Service Profiles using the profile class. It defines global properties of the service, common to all provided service capabilities, such as the service name and type and global context and QoS properties. While the context and QoS properties are described using our newly introduced classes (see Section 4.1.3), other properties are described using the OWL-S Profile properties as illustrated in Figure 4-1. We reuse all the properties defined in OWL-S Profile, except the functionality description, which will be specified at the capability level in our language:

- The service name, contacts and description respectively with serviceName, contactInformation and textDescription datatype properties.
- The service parameters for quality guarantees of the whole Amigo service with the serviceParameter object property.
- The service category, which refers to an entry in some ontology or taxonomy of services with the serviceCategory object property.
- The service type and product with the serviceClassification and serviceProduct properties.

The functional properties of a service presented in the Service Profile are further specified with the capabilities of the service (Section 4.1.2.1), the conversations of the service (Section 4.1.2.2), and the underlying middleware (Section 4.1.2.3).

4.1.2.1 Service capabilities

Instead of specifying the functionality provided by the service at the Profile level, as in the OWL-S language, we associate a service Profile with one or more capabilities. We introduce provided capabilities for describing capabilities that are offered by the service and required capabilities that are capabilities that should be provided by external services. If a required capability is not available, the service cannot deliver the provided capabilities.

We can thus easily describe several functionalities offered and required by an Amigo device, where common properties of all functionalities will be specified in the Profile definition, while specificities of each functionality will be described in the Capability definition. The functional description of the OWL-S Service Profile is useless since the functionalities will be specified per capability.

The functionality of the service is given per capability, using the Capability class of the Amigo-S language, related to the OWL-S Profile class with the hasCapability property (see Figure 4-2). Similarly to the OWL-S Service Profile functionality description, the description of each capability is given by the data inputs and outputs of the service, the preconditions that need to be fulfilled for the execution of the service and the effects (results) produced to the world by the execution of the service. Typed inputs and outputs correspond to messages that will be sent and received to and from the service and are expressed in any type system with XML Literals. Effects and pre-conditions are given in a logic formula, as prescribed by the OWL-S Expression class as DPR [DPR], KIF [KIF] or SWRL [SWRL] expressions. Note that as of OWL-S 1.2, expressions in SPARQL [SPARQL], RDQL [RDQL] and SWRL-FOL [S-FOL] will also be supported.

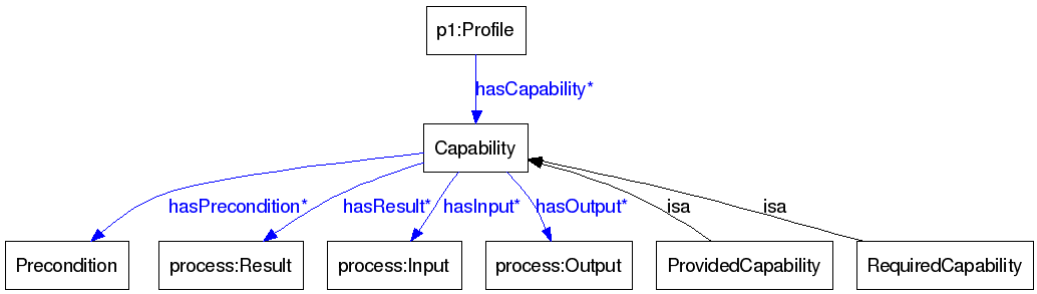


Figure 4-2: Specification of service capabilities

Classes related to the specification of a Capability are the following:

ProvidedCapability: Subclass of the Capability class. It is used to define capabilities offered by the service, which means that a user can make a request by invoking the service according to the inputs of the capability and get outputs, if any.

RequiredCapability: Subclass of the Capability class. It is used to define capabilities that should be provided by external services.

Input: The input class specifies the input messages that should be sent to the service for invoking the specific capability. It is a subclass of the Parameter class of the OWL-S Process ontology and has the following datatype properties, which allow to define messages in any type system:

- parameterType ≥1: anyURI : Defines the type system by giving the associated URI.
- parameterValue : XMLLiteral : Defines the value of the parameter in the type system defined by the parameterType.

Output: The output class specifies the output messages that are sent by the service after an invocation of the specific capability. Similarly to the input class, it is a subclass of the Parameter class and has the same datatype properties.

Result: The result class is used to define the effects in the environment of the capability after its execution. It is described in the Process ontology of OWL-S in terms of the object properties inCondition, hasEffect, hasResultVar and withOutput [Process] (see Figure 4-3).

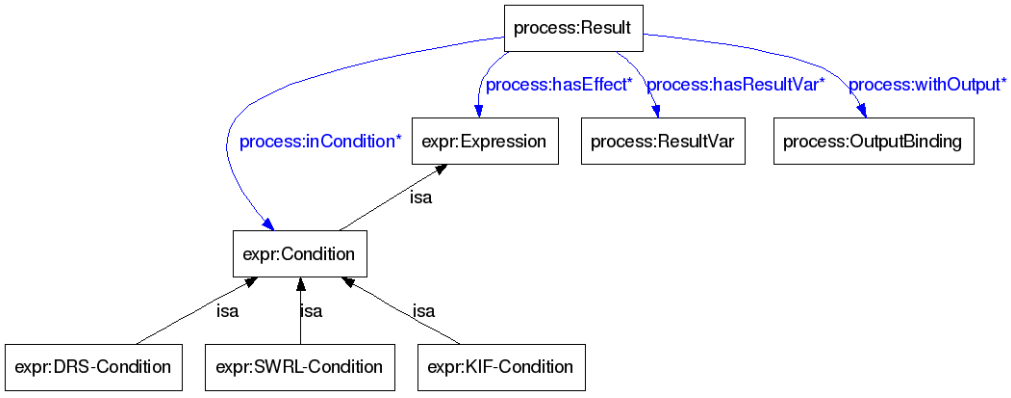


Figure 4-3: Definition of the Result parameter

- **inCondition**: Condition under which this result occurs. It can be expressed as a logic formula in DRS, SWRL or KIF (as defined in the 1.1 version of OWL).
- **hasEffect and withOutput**: States what ensues when the condition is true.
- **HasResultVar**: Declares variables that are bound in the inCondition.

Precondition: Preconditions (see Figure 4-4) that need to be fulfilled for the execution of the service for providing the specific capability. It is expressed as a DRS, SWRL or KIF expression.

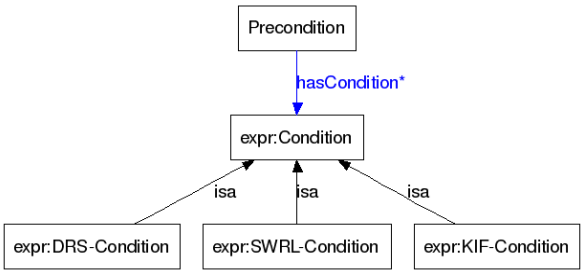


Figure 4-4: Definition of the Precondition class

4.1.2.2 Service conversations

A conversation gives how to interact with the service in terms of sequence of message exchanges. In OWL-S, each Service Profile can have a conversation description described with a Service Model associated to the profile with a has_process property. An Amigo service provides (or requires) several capabilities that describe each a different functionality. We thus introduce the hasConversation property for defining conversations associated to each capability (see Figure 4-5). The hasConversation property is a functional property stating that each capability has at most one conversation.

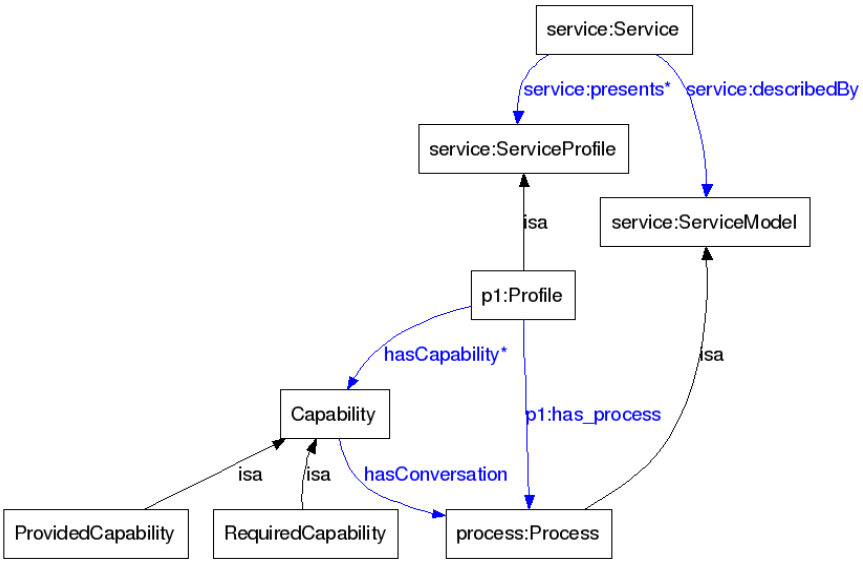


Figure 4-5: Specification of conversations

Conversations are described using processes, which can be atomic (single request-response interaction) or composite (consisting of several atomic or composite processes specified by using control constructs) (see

Figure 4-6, for the definition of a OWL-S process). The global conversation supported by the Amigo service would then be the union of all conversations of all the provided capabilities.

For easing definition of conversations, processes can be reused in several conversations that

implement similar interactions. A process describes a service expecting an input message and returning an output message.

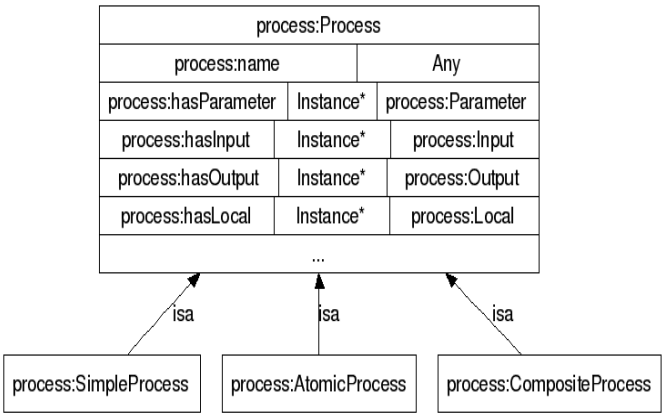


Figure 4-6: Definition of an OWL-S Process

Relationships between atomic and composite processes defined in the OWL-S Process ontology are given in

Figure 4-7, where simple processes provide an abstraction mechanism to provide multiple views of the same process.

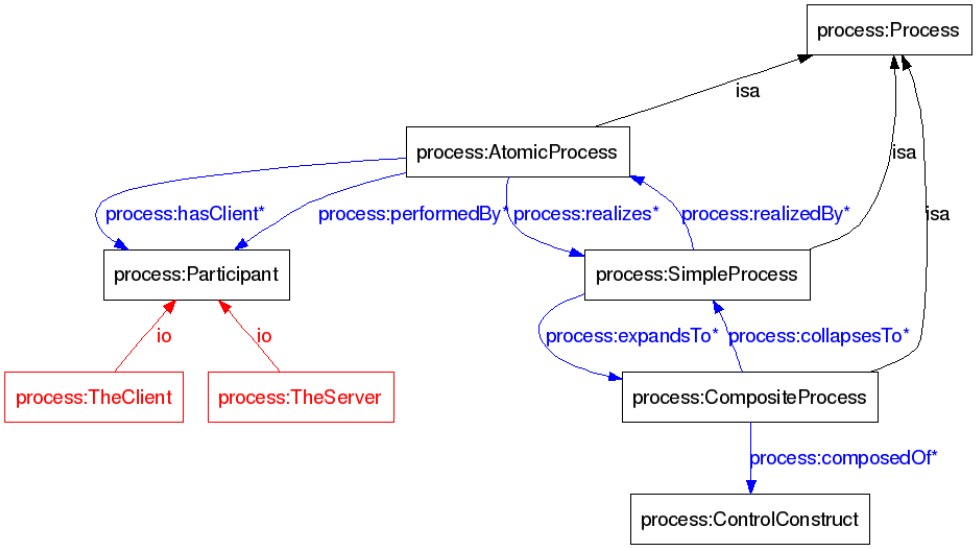


Figure 4-7: Relationship between Atomic and Composite Processes

4.1.2.3 Underlying middleware and network

In OWL-S, the specification of the underlying middleware of the service is given through a service grounding associated with the Service with the Supports property. We extend the OWL-S specification that gives only a WSDL grounding [Grounding] with the possibility of specifying other middlewares with a reference to the name of well-known middlewares and concrete connector specifications associated with each capability.

The Middleware class, represented in Figure 4-8, is used to define a ServiceGrounding, which is defined for the whole service, i.e., for all capabilities. A middleware is specified by giving the interaction protocol that is supported and the discovery protocol which should be used to publish and locate the service. External ontologies will be used to refine the InteractionProtocol and the DiscoveryProtocol defining the middleware. For instance, the discovery protocols should include at least the SLP, WS-Discovery and UPnP service discovery protocols used in the Amigo environment and the interaction protocols should include at least SOAP and JavaRMI interactions. An extensible ontology defining all specific middleware protocols used in the Amigo environment will be provided later. Note that no concrete mapping of sent and received messages with a particular protocol is defined, contrary to the WSDL grounding in the OWL-S specification. The referenced middleware protocols serve thus to instantiate appropriate interoperability mechanisms for enabling interactions between services deployed on top of different middleware infrastructures or for verifying their compatibility.

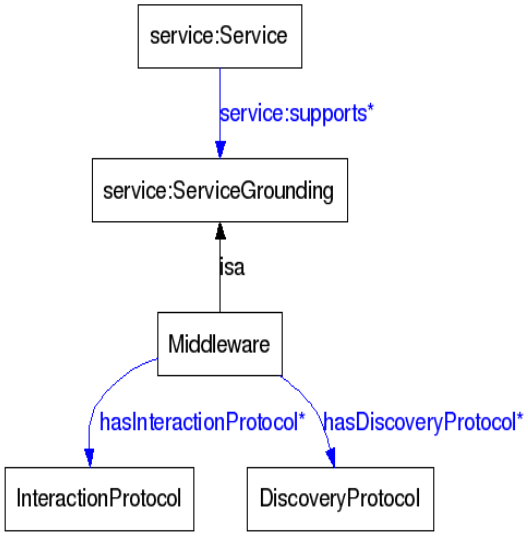


Figure 4-8: Specification of the underlying middleware

When no well-known middleware platform can be specified, the mapping of the capability with a concrete realization with a discovery mechanism and with an interaction should be specified using connectors. We introduce the Connector class, associated to a Capability with the hasConnector object property (see to the needs of Amigo services.

Figure 4-9). Similarly to the definition of well-known middleware infrastructures, a Connector can whether be an InteractionConnector or a DiscoveryConnector. Note that several

connectors can be specified for the same capability. The complete specification of the Connector class will be provided later, according to the needs of Amigo services.

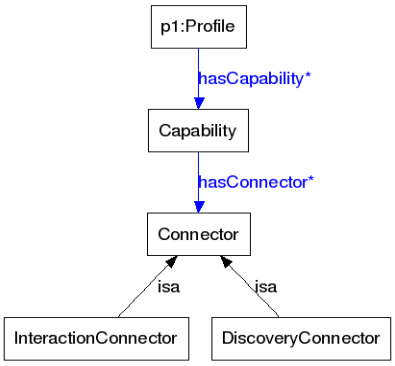


Figure 4-9: Specification of the Connector class

4.1.3 Description of service non-functional properties

Non-functional properties of an Amigo service include the specification of the context and QoS attributes of a service. The Amigo-S language defines generic context and QoS parameters representing different attributes.

The context parameter provides a standard generic modeling of arbitrary context information originating from various domains. The context specification of an Amigo service is described using the ContextParameter class, which can be defined globally for the whole Amigo service when associated to the Service Profile with the hasGlobalContextParameter property of a Profile, or separately for each capability of the service with the hasCapabilityContextParameter property of a Capability.

Similarly, the QoS parameter provides a standard generic model for arbitrary QoS attributes, while defining the nature of associations between QoS attributes and the way they are measured. The QoS attributes are either specified globally for the whole service with the hasGlobalQoSParameter property of the Profile class or for each capability with the hasCapabilityQoSParameter of the Capability class.

Specification of service context and QoS attributes is depicted in .

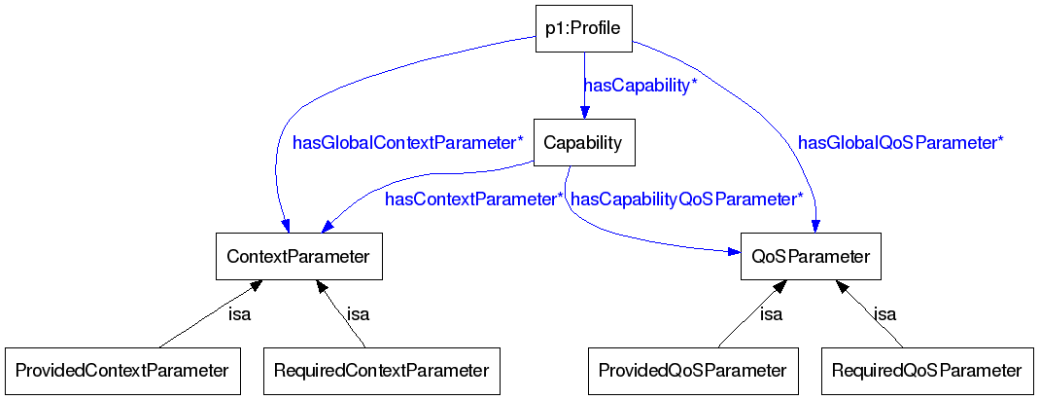


Figure 4-10.

Figure 4-10: Specification of Context and QoS Parameters

Furthermore, QoS parameters can be defined for the underlying middlewares using the hasMiddlewareQoSParameter property of the Middleware class (see

Figure 4-11).

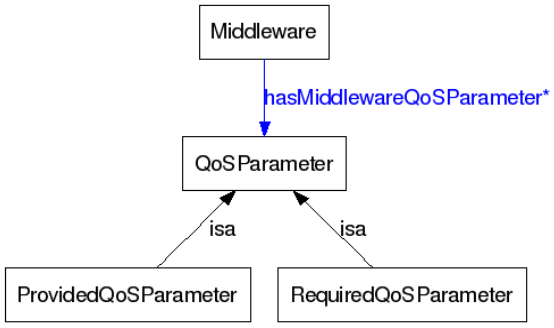


Figure 4-11: Specification of middleware QoS

Both the ContextParameter (see Figure 4-12) and the QoSParameter (see Figure 4-13) have been defined in deliverable D3.1b. They have been slightly updated for consistency with the language definition and formally specified in OWL.

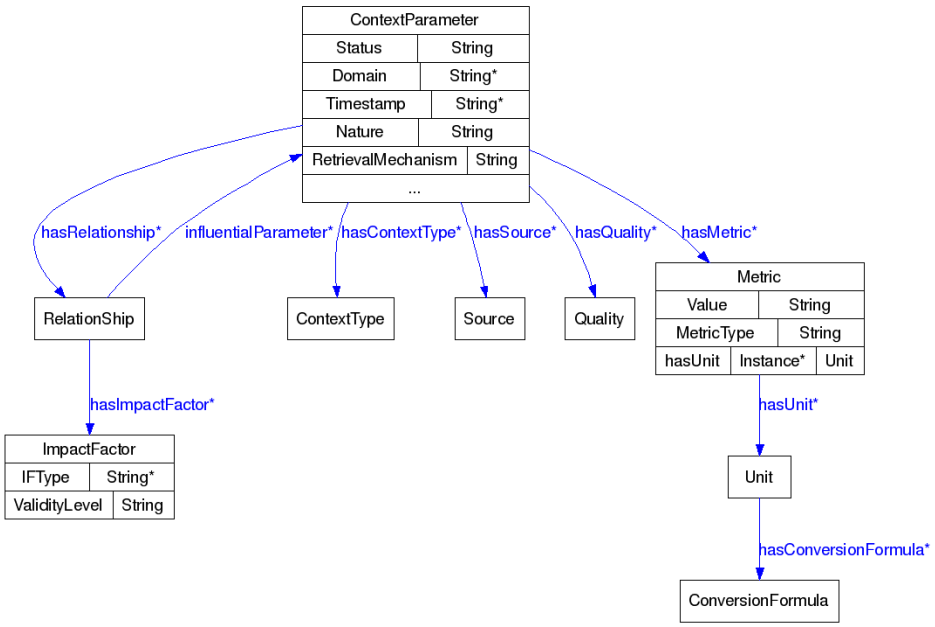


Figure 4-12 : The context Parameter ontology

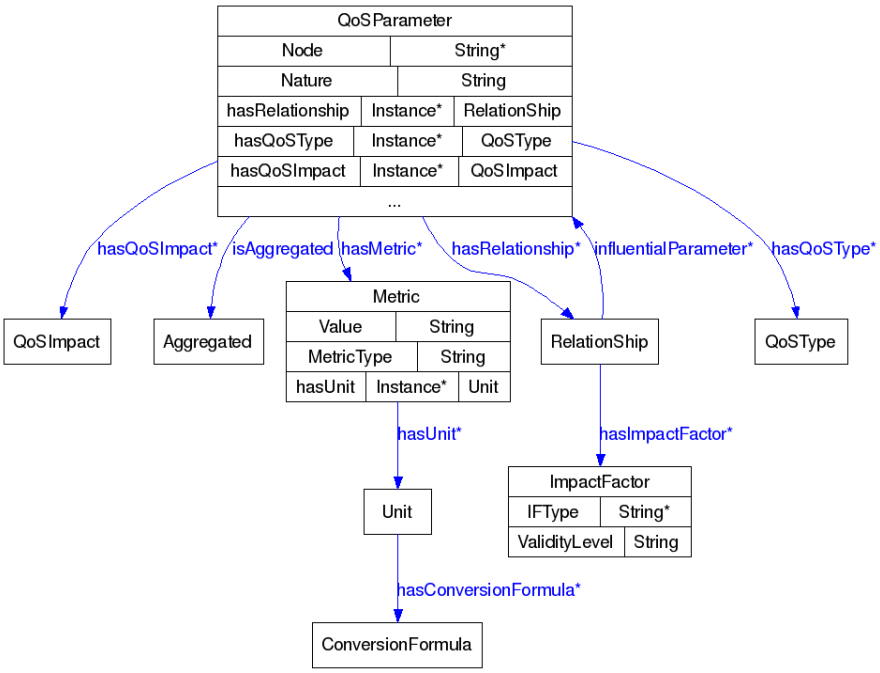


Figure 4-13 : The QoS Parameter ontology

4.2 Aspects of service discovery

In this section, we examine several aspects of enhanced service discovery in the Amigo environment. As already mentioned, the Amigo-S language introduced in the previous section will be the basis of Amigo-aware service discovery; however, for the moment, we elaborate on service discovery at a generic level without yet integrating Amigo-S. The first aspect that we deal with is related to semantic reasoning and semantic service matching. We assess existing tools and propose a service matching tool appropriate for Amigo (Section 4.2.1). The second aspect is related to context-aware service discovery. This concerns discovering context sources and using these context sources during service discovery to optimize the discovery process (Section 4.2.2). Finally, the third aspect is QoS- and resource-aware service selection, where we introduce an algorithm for selecting among semantically matching services by optimizing resource consumption (Section 4.2.3).

4.2.1 Service matching in the Amigo environment

Service matching is a key functionality in the Amigo environment. Amigo services should be matched based on their semantic descriptions. To describe Amigo services, we will employ the Amigo service description language (Amigo-S), detailed in Section 4.1. As already presented, Amigo-S is based on OWL-S and OWL, specifically the OWL DL sub-language of OWL. OWL DL represents semantics by supporting *Description Logics*. Description Logics (DL) are a family of knowledge representation languages which can be used to represent the terminological knowledge of an application domain in a structured and formally well-understood way [DL].

As one of the important functionalities to be executed while matching services is reasoning over semantic concepts to determine relationships between them, OWL reasoners or rather DL reasoners have an important portion in implementing the matching functionality. In Deliverable D3.1b, we presented an initial survey of semantic reasoning tools. In this section, we carry out comprehensive evaluation of DL reasoners that are currently available. As we are

interested in employing such tools in the very dynamic and resource-constrained Amigo environment, we look into the functioning of these tools from a systems perspective. Qualitative and quantitative evaluation of the tools is done in order to bring out the properties of the tools and their suitability for the Amigo environment. We evaluate DL reasoners based on various parameters such as their memory footprint, their support for a standardized interface, their query response time, etc. Section 4.2.1.1 deals with the various DL reasoners currently available and presents the evaluation of their efficiency in tasks which are particularly important with respect to matching of services.

Further we are interested in the existing service matching tools, which integrate (in a modular or non-modular way) DL reasoners. Currently in the research community, there are some prototype tools which have been implemented for semantic matching of Web services. These tools are implemented with the general intent of being used on the Web. In Section 4.2.1.2, we evaluate these tools – again from a systems perspective – and discuss their suitability for the Amigo environment. We further present the basic matching algorithm used by all currently available service matching tools.

From the evaluation of the tools currently available for semantic matching of services, it was inferred that they are unsuitable to be used in the Amigo environment. Therefore, based on the various features that were elicited during the evaluation of both DL reasoners and service matching tools, an architecture of a service matching tool which best suits the requirements of the Amigo environment is suggested and conclusions are drawn about the important factors affecting the performance of the newly proposed matching tool (Section 4.2.1.3). Our tool implements the basic matching algorithm and can integrate any DL reasoner providing a standardized interface; we further provide recommendations about the reasoners most suitable for AmI environments. We have developed a basic prototype of our matching tool and evaluated its performance.

The performance of the developed matching tool in terms of response time can be enhanced by applying certain optimization, keeping in mind the properties of the AmI environments. Related conclusions, recommendations and future work are presented in Section 4.2.1.4.

4.2.1.1 Evaluation of available semantic reasoning tools

Description Logics (DL) reasoners are software tools which, based upon the knowledge provided to them, try to compute inferences for drawing further conclusions making implicit knowledge explicit. The knowledge presented to a reasoner is in the form of a *knowledgebase*. A *knowledgebase* is a machine-readable collection of concepts, facts, rules, heuristics, models and procedures organized into schemas that can be used for problem solving. The assembly of all the information and knowledge in a knowledgebase pertains to a specific field of interest. A knowledgebase contains both concepts and relationships among concepts. A manually constructed knowledgebase in which not all relationships among various concepts are provided is called *asserted class hierarchy.* In order to determine relationships between concepts, it is necessary that all the relationships between the concepts are explicitly mentioned. For inferring these relations, the knowledgebase needs to be classified. Classification is the task of computing the *inferred class hierarchy* [HK]. This explicit representation of the knowledgebase where all the relationships among all the concepts are present in the knowledgebase is called the *inferred class hierarchy.* In addition to inferring information from the provided information, DL reasoners also have the capability to respond to various types of queries posed to them, based on the explicit knowledge initially provided to them and on the inferences that they have computed. Various DL reasoners have been developed in the research community e.g. FaCT++ [FaCT++], RACER [Racer], Pellet [Pellet], etc.

Description Logics reasoners form an integral part of any system realizing some aspect of the Semantic Web paradigm. DL reasoners reason on ontologies. A memory representation of an ontology acts as a knowledgebase for DL reasoners. As semantic web entities are specified by

using some semantic language such as OWL which implements DL semantics, DL reasoners come handy in extracting information regarding the relationships between such entities.

Requirements for a DL reasoner to be used in OWL-based semantic reasoning are as follows [GTB01]:

1. Dynamic – the reasoner should be dynamic as advertisements and ontologies would be added, removed and modified which would result into the re-classification of the knowledgebase
2. Ability to deal with multiple interconnected ontologies – as different ontologies would be used and concepts can be based on external concepts and relations between them
3. Scalability – the reasoner should be able to cope with large amounts of information in an efficient manner
4. Support for OWL syntax – which would avoid unnecessary translations

In this section, we present a set of DL reasoners and present a brief overview of these systems. We provide information such as the implementation programming language, dependencies on various other tools, uniformity of interface provided by the reasoner etc. We also discuss the conformance of the reasoners towards the four properties aforementioned. In terms of support for a uniform interface to the external world, a standard called DIG (Description Logics Implementation Group) interface [DIG] has been proposed. The DIG interface provides uniform access to DL reasoners. This interface defines a simple protocol (based on HTTP PUT/GET) along with an XML Schema that describes a concept language and accompanying operations. The interface is not intended to be a complete specification of a reasoning service; rather, it provides a minimal set of operations (e.g. satisfiability, subsumption checking and classification reasoning) which are useful in applications using DL reasoners. Further, we provide an evaluation of the reasoners with reference to a simple experiment which captures the essence of the matching process and thus helps us to evaluate the reasoners on a small and uniform task. Various efficiency parameters, both from the system point of view, such as memory footprint of the reasoner, and parameters that are important with respect to the matching tool, such as query response time, are provided. Finally we make recommendations on as to which reasoners would be most effective in the Amigo environment keeping in mind the key parameters of resource efficiency and support for a uniform interface amongst other parameters.

RACER

RACER (Renamed ABox and Concept Expression Reasoner) [Racer] is the first reasoner for reasoning over concepts, i.e., classes, and individuals, i.e., instantiations of classes. It is developed at the Computer Science Department of the University of Hamburg. It is able to deal with multiple ontologies, but they are not interconnected. It does not let the user define a concept in an ontology in terms of concepts and properties from other ontologies. RACER does not provide support for a dynamic knowledge base as it is not possible to add or remove concepts once the classification has been done. Another interesting feature of RACER is its ability to reason about individuals. RACER is implemented using Common LISP and it provides support for the DIG [DIG] interface and supports the OWL syntax.

FaCT++

FaCT++ [FaCT++] is the new generation of the FaCT [FaCT] (Fast Classification of Terminologies) OWL-DL reasoner. FaCT++ uses the established FaCT algorithms, but with a different internal architecture. The FaCT system is a DL classifier developed by Ian Horrocks from the Department of Computer Science at the University of Manchester. The FaCT system cannot deal with individuals or concrete datatype domains. Furthermore, it also does not

support multiple ontologies. FaCT allows adding classes over time to the knowledgebase and deals with the addition of new classes over time, even after the classification, but doesn't provide a mechanism for removing classes in the classification. FaCT is implemented in Common Lisp whereas FaCT++ is implemented with tableaux algorithms using C++ in order to create a more efficient software tool, and to maximize portability. FaCT++ provides support for a DIG interface and also supports the OWL syntax.

Pellet

Pellet [Pellet] is an open-source OWL DL reasoner. Pellet provides functionalities for checking consistency of ontologies, classifying the taxonomy and answering queries among other features. Pellet is an OWL DL reasoner based on the tableaux algorithms developed for expressive Description Logics. It supports the full expressivity of OWL DL including reasoning about nominals (enumerated classes). Therefore, OWL constructs owl:oneOf and owl:hasValue can be used freely. Currently, Pellet is the first and only sound and complete DL reasoner that can handle this expressivity. Pellet is implemented using Java to maximize portability and it also provides support for the DIG interface. Pellet provides support for the OWL syntax.

OWLJessKB

OWLJessKB [OJKB] is a description logic reasoner for OWL [MH-OWL]. The semantics of the language is implemented using Jess, the Java Expert System Shell [JESS]. OWLJessKB is a successor to DAMLJessKB [KR03]. Currently most of the common features of OWL Lite are supported. The exact details of which constructs of OWL are supported are not known due to a lack of documentation and nearly no support is provided by the implementers. OWLJessKb is implemented in Java and it does not provide support for the DIG interface.

KAON2

KAON2 is an open-source ontology management infrastructure targeted for semantics-driven business applications and is part of the Karlsruhe Ontology and Semantic Web Framework. It includes a comprehensive tool suite allowing easy ontology management and application. Important focus of KAON2 is on integrating traditional technologies for ontology management and application with those used typically in business applications, such as relational databases. For reasoning, KAON2 supports all features of OWL-DL apart from nominals (also known as enumerated classes). Since nominals are not a part of OWL Lite, KAON2 supports all of OWL Lite. KAON2 has been fully implemented in Java and provides a support for the DIG interface. In terms of support for OWL, KAON2 provides support for OWL.

Pocket KRHyper

Pocket KRHyper [SK05] is a reasoning system for Java-enabled mobile devices. The core of the system is a first order theorem prover and model generator based on the hyper tableau calculus. The development of Pocket KRHyper was motivated by the arising need for reasoning on mobile devices for mobile semantic web applications. To satisfy this need, a Description Logics interface is provided, which allows DL reasoning by transforming DL Expressions into first order clausal logic. Pocket KRHyper is implemented in J2ME [J2ME] and it does not provide a DIG interface yet, although there is ongoing work to support this. This reasoner is the first known effort towards a reasoner for a resource-constrained mobile environment, which could execute on a mobile device that supports J2ME. The executable code of this reasoner was not available at the time of writing this report and hence it was not

evaluated. The executable is to be made available soon. In terms of support for OWL, it's not yet known if Pocket KRHyper provides support for OWL or not.

Table 4-1 presents a comparison of the reasoners presented above in terms of implementation language, license type, DIG support and memory footprint.

Reasoners evaluation

In this section, results of a simple experiment which captures in essence the process of matching service capabilities are detailed with respect to various DL reasoners. In order to have a flexible design of the tool, it was felt that the reasoning functionality must be separated from the matchmaking functionality, which was only possible if the reasoner existed as an independent system providing a uniform interface to the matching tool. Thus, only reasoners which provide the support for the DIG interface were tested.

Matching service descriptions as a problem can be essentially reduced to the atomic operation of matching two concepts. For matching service descriptions, this operation is looped over the number of various concepts which are either inputs or outputs of the services. As this experiment was carried only in order to evaluate reasoners, other parameters in the Service Profile were ignored here.

Reasoner	Language	License	DIG Support	Base Memory Footprint
RACER	Lisp	Commercial/ Research	Yes	~ 10 MB
FACT++	C++	GNU PL	Yes	~ 21 MB
Pellet	Java	MIT License	Yes	~ 15 MB (JVM)
KAON2	Java	Commercial/ Research	Yes	~ 15 MB (JVM)
OWLJessKB	Java	GNU GPL	No	> 12 MB
KRHyper	J2ME	- Not mentioned -	Not yet	Development Stage

Table 4-1: Comparison of various reasoners based on different parameters

The operation of matching two concepts in itself consists of five steps:

1. Creating a memory model of the ontology by parsing the ontology – Ontologies are expressed in OWL and follow an XML like syntax. The ontologies can be local files are remote URI's. In order to be used for the purpose of reasoning, ontology files need to be read and parsed to create a memory representation in the form of graphs. A very useful tool for this purpose is Jena [JENA], which is a Java API that can be used to create and manipulate RDF [B2004] graphs.
2. Loading the model into the reasoner as the knowledgebase – A reasoner needs a knowledgebase to reason over. This knowledgebase is actually the memory representation of the ontology which the reasoner would reason in order to find a

relation between the concepts. As mentioned earlier, the memory model acts as the knowledgebase.

3. Classifying the knowledgebase to compute the inferred hierarchy – An interesting point to note is that by explicitly telling the reasoner to compute the inferred hierarchy results in a lot of gain in the performance of the matching operation as shown later in this section by Table 4-2 and Table 4-3.
4. Extracting the details of the concepts to be matched from the memory model – implies extracting the details of the concepts that need to matched by the reasoner from the knowledgebase.
5. Requesting the reasoner to reason over the knowledgebase in order to extract relationship between the two concepts – wherein the reasoner actually performs the reasoning to return results about the relation between the two concepts submitted.

We assume that due to the use of Semantic Web paradigm, concepts used in different descriptions for describing inputs and outputs would be using same ontologies, and, thus, loading a single ontology into the reasoner as the knowledgebase suffices. If this is not the case, the reasoner can be loaded with a second knowledgebase and reasoning can be done. This could arise if two concepts belong to two different ontologies which import each other. In any case, the construct owl:import would be used, and, thus, in the final memory model, both the concepts would be present.

We have implemented a Java program for matching of two concepts and we have evaluated for each reasoner the following parameters:

1. Time for the creation of the memory model
2. Time to classify the ontology
3. Response time for a single query for matching two concepts

Out of these three parameters, the second and the third parameters are paramount in determining the performance of the reasoners because the memory model creation time is dependent on the processing capabilities and the available main memory of the machine on which the tool is executed. Further it is also dependent upon the network latency, as concepts defined in the currently used ontology may refer to some other external concepts. The experiments were done with ontology files being accessed over the web. The observations regarding these results are made later in this section. The ontology used for the experiment can be found at [Pizza.owl]. The ontology contains 99 OWL Classes, 4 Datatype Properties, 11 Object Properties, 24 Annotation Properties and 5 Individuals. Times needed to execute the code to calculate the value of the parameters was done by taking the time difference between readings of the absolute system time measured before and after executing the code. All the experiments were conducted on a Toshiba Satellite notebook with 1.6 GHz Intel Centrino processor and 512 MB of RAM. For the Java Virtual Machine the Java Runtime Environment v. 1.5.0_02 was used. Each experiment was conducted ten times, and Table 4-2 documents the average values for parameters as detailed earlier.

	Time to create memory model		**Time taken to classify**		**Time taken to match concepts**		**Run time memory footprint**
Reasoner	**Average**	**SD**	**Average**	**SD**	**Average**	**SD**	
RACER	3986 ms	152.7	2323 ms	217.3	22 ms	1	~ 18 MB
FACT++	4967 ms	187.2	869 ms	97.2	26 ms	2.3	~ 23 MB
Pellet	4014 ms	164	3060 ms	151.3	16 ms	1.5	~ 30 MB

KAON2	4142 ms	176.3	Out of Memory		Out of Memory		<< 80 MB

Table 4-2: Average times, with classification done before matching

	Time to create memory model		Time taken to match concepts		Run time memory footprint
Reasoner	**Average**	**SD**	**Average**	**SD**	
RACER	3923 ms	151.1	1573 ms	32.1	~ 18 MB
FACT++	4371 ms	161	580 ms	21.7	~ 23 MB
Pellet	4963 ms	189.3	733 ms	53	~ 30 MB
KAON2	4155 ms	167.3	Out of Memory		<< 80 MB

Table 4-3: Average times taken, without the ontology being classified

With reference to the results displayed in Table 4-2, the following observations can be made.

1. The most expensive operation in the whole process is the time to create a memory model from the OWL description. This is due to XML parsing and the fact that ontologies import and use other ontologies to describe their concepts. As the access to the other ontologies is over the Web, it is a time consuming process. Furthermore, even if an ontology refers to just a part of any other ontology, it needs to import the whole ontology, which is a drawback in the design of OWL itself [H2003].
2. Classification is expensive in terms of time, but as it is a one time operation and is it greatly reduces the time to compare – it should be done before matching.
3. It was observed that the memory consumption of reasoners increases with use as many knowledgebase's are loaded into the reasoner over a period of usage. The reasoners must be reset periodically in order to limit their memory consumption.

Based on the experimental results and the above mentioned observations, the following conclusions can be drawn:

1. FaCT++ is the most efficient for the AmI environment of the tested reasoners. It supports the DL family that is required for the matching process. It is memory efficient, provides support for DIG interface and has the best ontology classification and query response times. RACER and Pellet are also competitive candidates and can be used instead of FaCT++.
2. Ontologies need to be properly engineered providing efficiency of use in an AmI environment. This implies that the ontologies used for describing services must be designed and implemented with the specific purpose of being used in the AmI context. This can be achieved as follows:
    a. Maintaining local copies of all the ontologies being used by service descriptions
    b. Enforcing that while importing external ontologies, the ontologies import the local copies instead of the ontologies available on some external source such as the web

3. From the experimental results it can be calculated that the total time taken by the process of matching two concepts is of the order to 6-7 seconds, which is quite long in terms of a response time for a user requesting a service. Also in case matching has to be done with multiple services, the response times are bound to exceed the response times calculated in the present experiment. The response times can be reduced by doing the following:
    a. Memory models of ontologies are prepared offline and stored as memory models (Objects) directly. This would greatly reduce the response time.
    b. Classification of the ontologies greatly reduces the times to match concepts. Classification of the ontologies can be done offline.

The optimizations mentioned above are part of our future work and are detailed in Section 4.2.1.4.

4.2.1.2 Evaluation of available service matching tools

In this section, we evaluate available tools that carry out semantic matching of services. We first provide some insight into the base algorithm that is being used in all these tools. This base algorithm is implemented in various manners to suit the individual requirements of the tools. Then, qualitative evaluation of these tools is done based on their suitability for the Amigo environment, and comments are made about their interesting features and their drawbacks in general and in terms of efficiency and resource consumption. Finally, we provide a summary of the drawbacks and interesting features of these tools. Lessons from the evaluation of these tools are used in Section 4.2.1.3 for the design of an efficient tool suiting the Amigo environment.

The base matching algorithm

The base algorithm that is used in all the tools mentioned in this section has been proposed in [PKPS02]. The main rational behind this matching algorithm is that a service advertisement matches a service request when the service provided by the advertiser can be of some use for the requester. More specifically, an advertisement matches a request when all the outputs of the request are matched by the outputs of the advertisement, and all the inputs of the advertisement are matched by the inputs of the request. This criteria guarantees that the matched service satisfies the needs of the requester, and that the requester provides to the matched service all the inputs that it needs to operate correctly [PKPS02].

The main control loop of the matching algorithm, in which a request is matched against all the advertisements found in a registry, is shown in Figure 4-14. Whenever a match between the request and any of the advertisements is found, it is recorded and scored in order to decide the matches with the highest degree.

```
match(request) {
        recordMatch= empty list
        forall adv in advertisements do  {
                if match(request, adv) then
                recordMatch.append(request, adv)
        }
        return sort(recordMatch);
        }
```

Figure 4-14: Main control loop [PKPS02]

A match between an advertisement and a request consists of the match of all the outputs of the request against the outputs of the advertisement; and all the inputs of the advertisement against the inputs of the request. The algorithm for output matching is described in detail in Figure 4-15: a match is recognized if and only if for each output of the request, there is a matching output in the advertisement. The degree of success depends on the degree of match detected. If one of the request's output is not matched by any of the advertisement's output the match fails. The matching between inputs is computed following the same algorithm, but with the order of the request and the advertisement reversed: whereas the request's outputs are matched against the advertisement's outputs, the advertisement's inputs are matched against the request's inputs.

```
outputMatch(outputsRequest, outputsAdvertisement) {
    globalDegreeMatch= Exact
    forall outR in outputsRequest do {
        find outA in outputsAdvertisement such that degreeMatch= maxDegreeMatch(outR,outA)
        if (degreeMatch=fail) return fail
        if (degreeMatch<globalDegreeMatch)
        globalDegreeMatch= degreeMatch
        return sort(recordMatch);
        }
```

Figure 4-15: Algorithm for output matching [PKPS02]

The degree of match between two outputs or two inputs depends on the relation between the concepts associated with those inputs and outputs. The degree of match is determined by the minimal distance between concepts in the taxonomy tree. The four degrees of matching are differentiated according to the rule displayed in Figure 4-16, where outR corresponds to one output of the request and outA corresponds to one output of the advertisement.

```
degreeOfMatch(outR,outA):
    if outA=outR then return exact
    if outR subclassOf outA then return exact
    if outA subsumes outR then return plugIn
    if outR subsumes outA then return subsumes
otherwise fail
```

Figure 4-16: Rules for the degree of match assignment [PKPS02]

The rational for the degree assignment is described below:

- *Exact* - If outR = outA then outR and outA are equivalent, which is then labeled as exact. The second clause is a bit more complicated; if outR subclassOf outA then the result is still exact under the assumption that by advertising outA the provider commits to provide outputs consistent with every immediate subtype of outA.
- *Plug in* - If outA subsumes outR than outA is a set that includes outR, or, in other words, outA could be plugged in place of outR [ZW97].

- *Subsumes* - If outR subsumes outA, then the provider does not completely fulfill the request. The requester may use the provider to achieve its goals, but it likely needs to modify its plan or perform other requests to complete its task.
- *Fail* - Failure occurs when no subsumption relation between advertisement and request is identified.

Degrees of match are organized along a discrete scale in which exact matches are of course preferable to any another; plug in matches are the next best level because the output returned can probably be used instead of what the requester expects. Subsumes is the third best level since the requirements of the requester are only partially satisfied: the advertised service can provide only some specific cases of what the requester desires. Fail is the lower level and it represents an unacceptable result.

The last piece of the algorithm to discuss is the scoring system used to sort the resulting matches. The rules used to sort are shown in Figure 4-17. The rationale behind them is that the requester expects first and foremost that the provider achieves the output requested at the highest degree. This is reflected in our rules by establishing that the main sorting criterion is to select the match with the highest score in the outputs. Input matching is used only as secondary score to break ties between equally scoring outputs.

```
sortRule(match1,match2) {
        if match1.output > match2.output then match1 > match2
        if match1.output = match2.output
        & match1.input > match2.input then match1 > match2
        if match1.output = match2.output
        & match1.input = match2.input then match1 = match2
        }
```

Figure 4-17: Rules for the degree of match assignment [PKPS02]

It might be interesting to note that in spite of the fact that much more information is contained in the OWL-S service profile, for matching purposes only Inputs and Outputs are used. The reason for this is that only a part of the information contained in the profile is useful for matching purposes. This is because the other parameters do not help to define what a service actually does. They just provide auxiliary information for human or machine consumption and do not contain any semantic information which might be used for matching purposes. Matching Inputs and Outputs gives us a high probability that the service that we have found is what we are looking for. This is because it's highly unlikely that a service that is modeled for a particular functionality and which is found as a match will do something totally different than what is expected. E.g. a service which takes as inputs arrival and destination airport and gives an output as a flight ticket is highly unlikely to actually be a book selling service. The probability of matches is increased if we add Preconditions and Effects to the matching procedure. Preconditions give us the required contextual conditions that are required for the successful execution of a service. The Effects give us produced results of a service that is other than data output and is in the form of the impact that the execution of the service has on the context of the service and its users. For example, the precondition for a service selling books might be that the user has a valid credit card number and the effect can be the generation of an invoice. Although, preconditions and effects are important parameters for the execution of services in the real world but very little work can be found in the literature about how to match them. Secondly, the preconditions and effects are expressed in First Order Logic, which is undecidable in terms evaluating the truth value of expressions. We plan to look at the matching of preconditions and effects in our future work.

With this overview of the algorithm provided, we now present a number of available semantic service matching tools. All these tools adopt the aforementioned algorithm. Nevertheless these tools have defined various degrees of matches based on the degrees mentioned above and modified them according to individual design consideration and implementations.

OWL-S Matcher

The OWL-S Matcher [TTUBa] is a Java implementation of a matchmaking algorithm for matching OWL-S descriptions. The matchmaker compares two descriptions (one from a service requester and another from the service provider) and identifies different relations between the two descriptions (e.g. "match" or "no match"). The implementation of the OWL-S matcher can be found in [TTUBa]. The initial version of the matcher was an implementation for matchmaking DAML-S descriptions, which is also available at [TTUBa]. It was later updated to be compliant with OWL-S as the semantics of OWL were somewhat different than its predecessor DAML+OIL. The matcher demonstrates an algorithm that returns different degrees of matching for individual elements of OWL-S descriptions. Particularly, the algorithm considers various elements of the service profile and provides ranks for them after matching. Ranking allows the selection of a service among a large set of results. The tool provides a Swing-based GUI, which allows selecting OWL-S descriptions, one description each for the requester and the provider and compares them to give a result in terms of the matching categories pre defined by the authors. OWL-S Matcher uses OWLJessKb as the reasoner for the reasoning purposes. The OWL-S Matcher starts its execution by loading the service profiles of the requested service and the advertised service which are specified in OWL-S. It gives an option to the user to select the expected minimum matching degrees for Input matching, Output matching and Profile matching. The various degrees for matching are as follows:

1. *fail*
2. *unclassified*
3. *subPorperty*
4. *type_Invert*
5. *type_Subsume*
6. *match*

The matching categories *match*, *type_subsume*, *type_invert* and *fail* are the same categories as Match, Subsumes, Plug-in and Fail categories respectively as detailed earlier. The match category unclassified is used to prevent false matches in case either the input/output of the requested service or of the advertised service that are being matched is not classified. The match category subProperty is used in case the input/output of the requested service is a subproperty of the input/output of the advertised service. The complete details of these various degrees and what these imply in terms of subsumption relation are provided in [27, 28].

The reasoner used in the tool is OWLJessKb. The reasoner is embedded into the tool and thus the design of tool is not very scalable. As the reasoner is embedded into the code and due to the coding practices adopted, it's not possible to replace the reasoner with any other more efficient and robust reasoner, without more or less completely rewriting the code. Moreover keeping the drawbacks of the OWLJessKb in mind, as detailed in Section 4.2.1.1, the tool provides a high degree of design level inflexibility to be used in an AmI environment. The tool gives the user a choice to compare one request against one advertisement, and does not implement the main control loop of the base algorithm presented in Figure 4-14 to match over a set of services present in a repository. Considerable modifications need to be done, in order to suit such requirements. As the tool does not implement the main control loop, it is not

able to provide any rankings or weights with respect to the degree of match of the advertised services.

In terms of efficiency, the tool is relatively slow to load profiles into the knowledgebase of the reasoner. The reason for that is that for every operation that is done in case of loading the profile, e.g., parsing the OWL-S description of the service for extracting the information about the service, OWL-S Matcher uses the reasoner, i.e., the profile is loaded into the reasoner as a knowledgebase and queries are executed to extract simple parsing information. As it was observed in Section 4.2.1.1, creating a memory model of an ontology is a highly time consuming process, such an operation must be undertaken with care. The tool again loads the profiles into the reasoner at the time of matching thus greatly reducing the efficiency of the reasoner. The results of loading and matching the same profile are provided in Table 4-4. The option selected for the minimum degree of match was “Fail” for all inputs, outputs and profile matching.

Average time to load the requested profile	6680 ms
Average time to load the advertised profile	6679 ms
Time to match two services	5947 ms

Table 4-4: OWL-S Matcher performance

Besides all the above drawbacks, the system has not been systematically tested [TTUBa]. According to the authors, the tool contains a few software engineering errors and improvements need to be done. The usage of this tool in the Amigo environment can be totally ruled out based on the observations made above.

In spite of the drawbacks mentioned above, there are some positive points about the tool which can lead to a robust implementation of a matching tool. The tool employs a split algorithm where matching is done in four parts, all the parts being independent of each other. The reason for adopting the split algorithm [TTUBb] is that OWL-S allows for a very detailed description of a Web Service and it might easily occur that two profiles will be declared as non-compatible (i.e. no semantic relation could be determined) because one (probably less important) property in a profile stands in contrast to a property in the other profile. Also the algorithm is extensible in the sense, that it has an option to incorporate user defined plug-ins which might be extending the matching to several other parameters which might be of more importance to the user, e.g., a user can specify to additionally match the QualityOfService parameter enabled by the profile in the plug-in as it might be of important consequence for the user while selecting the service. Thus the algorithm is split and the final result for the matching depends on the individual matching results produced by the four parts of the algorithm and the user defined plug-in.

OWL-S/UDDI Matchmaker

OWL-S/UDDI Matchmaker [SUDDI] is a tool which takes advantage of UDDI’s [UDDI] proliferation in the web service technology infrastructure and OWL-S’s explicit capability representation. UDDI is an acronym for Universal Description, Discovery, and Integration – A platform-independent, XML [XML] based registry for businesses worldwide to list themselves on the Internet. In order to achieve the symbiosis the OWL-S profile descriptions are stored inside an UDDI registry and a mapping between the OWL-S profile and the UDDI data model is provided. The UDDI registry is enhanced with an OWL-S matchmaker module which can process the OWL-S description, which is present in the UDDI advertisements. The matchmaking component is embedded in the UDDI registry. The belief is that such architecture brings together both these two technologies, working toward similar goals. A capability port is also added to the UDDI registry, which can be used to search for web

services based on their capabilities. The contributions of this tool are an efficient implementation of the matching algorithm proposed in [PKPS02], an architecture that is tightly integrated with UDDI, an extension of the UDDI registry and the API to add capability search functionality. The tool employs RACER [Racer] as the reasoner, using the DIG interface.

The OWL-S /UDDI matchmaker starts its execution by publishing the details of a web service described in OWL-S in the UDDI registry. Once published, some processing is done in the UDDI registry, by annotating all the ontology concepts in the matchmaker with the degree of match that they have with the concepts in each published advertisement. The authors assume that the publishing phase of a web service is not time critical and hence exploit this time to pre-compute the degree of match between the advertisement and the possible requests. The matchmaker maintains a taxonomy that represents the subsumption relationship between all the concepts in the ontologies that it loaded. Each concept in this taxonomy is annotated with two lists *output_node_information* and *input_node_information,* that specify to what degree any request pointing to that concept would match the advertisement [PKPS02]. As a consequence to this, the effort needed to answer a query is reduced to a little more than just a lookup into the annotated taxonomy. The rationale behind this approach is that since the publishing of an advertisement is a one time event, it makes sense to spend more time to process the advertisement and store the partial results and speed up the query response time. As the query response time is critical and queries can be very frequent, the tradeoff works.

The tool defines various degrees of match which are:

1. *exact*
2. *plug in*
3. *subsume*
4. *fail*

The details of these degrees of match were detailed earlier in this section.

This implementation implies that most of the work required by the matching algorithm is done during the publishing phase itself, thereby spending considerable amount of time in this phase. In the querying phase, since most of the matching information is pre-computed at the publishing time, the matchamker's query phase is reduced to a simple lookup in the hierarchical data structure.

For more interested readers the details of the tool can be found at [SPS05].

The OWL-S/UDDI Matchmaker makes uses of UDDI, which is a centralized repository and needs the support of a database management system in order to store the published services. Although scalable, the repository is memory and computation intensive. Moreover it requires a lot of time and effort to set up due to the complexity of the repository itself, which in the context of an AmI environment might not be acceptable. In terms of efficiency the publishing takes a lot of time. The results of publishing an advertisement at a local web server with jUDDI [jUDDI] running on it and a MySQL [MySQL] Server running at the backend produced the following results:

Service publishing time	57,533 ms
Querying time	601 ms

Table 4-5: OWL-S/UDDI Matchmaker performance

The authors of the tool assume that the services are published only once and might never be removed or modified. This might be true in a World Wide Web setting but in the context of the current work, this however might not be true as services may enter or leave the AmI environment. This will create a considerable time and memory overhead for updating a

taxonomy that is being used in the dynamic AmI environment similar to the one that is maintained by this tool and might also create consistency issues.

In terms of querying the repository for matches the results are very promising as it takes less than a second to respond to a request. Moreover, the response from the tool provides response in terms of multiple matches found in the repository giving a weight to each of them in terms of the degree of match. The higher the match the better the advertised service matches the request. Also, one of the interesting features of the architecture of the tool is that, it uses a reasoner separately running perhaps on a separate machine, and that all the communication happens over the DIG [DIG] interface of the reasoner. This gives a high degree of flexibility: in case a better performing reasoner is developed which supports the DIG interface, it can be replaced into the system without modifying the code of the matchmaker. This design feature of separating the reasoner from the matchmaker is a must for the tool that is to be used in an AmI environment.

OWLS-MX

OWLS-MX [KKS-MX] presents an approach to hybrid semantic Web service matching, that utilizes both logic based reasoning and content based information retrieval techniques for services specified in OWL-S. In content based service retrieval, the meaning of concept expressions of service descriptions is not a function of the way the parts are syntactically combined by description logical language operators and model-theoretically interpreted. Rather, it is implicit in the relative frequencies of indexed terms of these expressions and exploited by string edit or token based information retrieval similarity metrics with associated term weighting schemes [KKS-MXb]. The tool is motivated by the belief that building semantic Web service matchmakers purely on description logic reasoners artificially limits their potential. The authors believe that purely logic-based reasoning on respectively annotated content and services may not be enough. It would artificially limit service matching to one type of representation only where expressiveness and value reasoning has been compromised at the expense of computational properties such as decidability. The approach adopted by the tool to cope with this problem is to tolerate logical failures by complementary approximate matching based on syntactic similarity. It is acknowledged by the authors that the adaptation to the latter eventually is on the user end. OWLS-MX matcher is implemented using Java and for experimental results used Pellet [Pellet] as the reasoner.

The OWLS-MX matcher starts its execution by loading a service description described in OWL-S into the tool. The registered services could be called the advertised services. Next the user requests that need to be checked for matches are registered into the tool. While registering these advertised and requesting services, the services are just loaded into the tool but not added to the actual matchmaking module. This is done in order to save memory. The tool also has an option to add a *Test Collection* which is a collection of OWL-S descriptions for testing the performance of the tool. Again as with other tools, the tool offers the various degrees of match for the concepts that are being matched. These are:

1. *exact*
2. *plug-in*
3. *subsumes*
4. *fail*
5. *subsumed-by*
6. *nearest neighbour*

The first four degrees of match are the same as defined above, whereas the *nearest neighbour* match uses logic as well as syntactic matching for determining the match between two concepts. The result of the matching is produced as a *nearest neighbour* match when for

all inputs in the advertised service there exists at least one input in the requested service which is subsumed by one of the advertised services inputs and for all outputs in the requested service there exists at least one input in the advertised service which is subsumed by one of the requested service outputs and the syntactic similarity between the matched parameters is greater than a threshold value. The *subsumed-by* match is produced when advertised service's output data is more general than requested, and hence subsumes the request. More details of these degrees of matches can be found at [KKS-MXb].

The tool gives results in terms of the answer set, i.e., the set of services that matched with the request, average query response time and memory consumption during execution. The results of the experiment for matching a registered service description against a request are shown in Table 4-6. The time required to register the service could not be noted due to the programmatic complexity of the tool.

Average time taken to match services	623 ms
Memory consumed during matching	~ 64 MB

Table 4-6: OWLS-MX Matcher performance

OWLS-MX Matcher uses Pellet as the reasoner for the reasoning purposes. The reasoner is embedded into the tool and perhaps static queries have been hard coded into the code and thus the design of tool is not very scalable. Again as in OWL-S Matcher the reasoner is embedded into the code and due to the coding practices adopted, it's not possible to replace the reasoner with any other more efficient and robust reasoner, without more or less completely rewriting the code. Thus, the tool provides a high level of inflexibility to be used in a pervasive computing environment such as AmI. In terms of efficiency, a visible delay is shown by the tool to load service descriptions. As can be seen by the results detailed in Table 4-6, the memory consumption of the tool was too high to be suitable for use in the AmI environment.

In terms of querying the registered services for matches, the results are very promising as it takes less than a second to respond to a request. The tool implements algorithms for syntactic matching using various similarity measures. This can be an interesting feature to add in the new tool to get the best of the both worlds. The tool also matches a request against a set of services, which is a desirable feature in the context of the current work.

Evaluation summary

The summary of the interesting features of the tools presented and their drawbacks are presented in Table 4-7. Based on these observations and the constraints posed by the AmI environment the desirable features and the architecture of the new matching tool is detailed in Section 4.2.1.3.

Tool	Interesting Features	Drawbacks
OWL-S Matcher	• Employs a split algorithm • Extensible in terms of matching over some additional parameters such as QualityOfService	• Uses OWLJessKb as the reasoner while the semantics supported by OWLJessKb are not known • Uses the reasoner embedded into the tool • Does not provide

		functionality to match over a set of services • Slow in terms of loading profiles • Slow in terms of matching
OWL-S UDDI/Matchmaker	• Provides results by matching a request against a set of services • Provides raking of services in terms of degree of match • Makes use of a reasoner as an independent system, decoupled from the matching logic • Very good query response times for matching services	• Is coupled with a registry type which is not suitable for the AmI environment • Publishing of a service takes large amount of time • Memory intensive as a taxonomy is maintained • Performance degradation if services are published and removed frequently
OWLS-MX Matchmaker	• Uses semantic as well as syntactic matching	• Uses the reasoner embedded into the tool • Memory intensive

Table 4-7: Summary of the properties of currently available matching tools

4.2.1.3 A tool for on-line service matching in the Amigo environment

Any software component that executes on resource-constrained devices in the AmI environment must be lightweight in terms of resources consumption, such as memory and processing time. This requirement of being lightweight is significant due to the resource constraints that the devices such as PDA's, smart phones have. In order to meet this critical requirement of having a lightweight implementation of software being used in the AmI environment, it is important to devote considerable amount of work to the design and architecture of the software. Monolithic design's implementing all the functionality in a single module must be avoided. The design should be modular and the modules should be loosely coupled. Modular and loosely coupled design enables flexible distribution of functionality.

All the matching tools discussed in Section 4.2.1.2, fail to qualify the basic requirement of being lightweight. All of these tools consume excessive memory and processing time while executing. Other reasons why the tools are not suitable for the Amigo environment are:

1. The design of the tools is monolithic – all the functionality is placed in a single system
2. Internal modules are too tightly coupled – e.g. in OWL-S Matcher [TTUBa] and OWLS-MX [KKS-MX] the reasoner is embedded into the software and it is not possible to replace the reasoner with some other reasoner
3. Implemented for systems that don't fit the AmI context – e.g., OWL-S UDDI Matchmaker is coupled with a registry that is not suitable for the Amigo environment

Based on the observations made above and in the last two sections, and keeping in mind the constraints imposed by the Amigo environment, the desirable features that must be present in a matching tool can be listed. The design of the tool should incorporate the various positive features that have been elicited from evaluation of the tools. The new tool should have:

1. a modular non-monolithic design
2. loosely coupled subsystems
3. be lightweight in terms of memory consumption and the processing capabilities required to execute this tool

Architecture

With reference to the overview of the requirements of a service matching tool fitting the Amigo environment, we can translate the requirements into design considerations and architecture of the new tool. The prime design factors that govern the architecture of the tool are:

1. Reasoning is an expensive task in terms of memory and processing power required. Thus the reasoner should be a standalone individual subsystem. This would avoid the tool being monolithic and separating the reasoning functionality from the matching operation. A reasoner instance can be executed in server mode on a fixed resource rich device which can be ubiquitously accessed over a standardized interface.
2. Ontology imports from external sources such as the Web take a considerable amount of time and increases the query response times. Thus the ontologies that are being used in the environment should be pre-fetched and made available locally within the environment. The ontologies downloaded can be present on fixed repositories that may act as a cache. The service description referring to these ontologies must be carefully designed and engineered to include the local versions of the imported ontologies for maximum efficiency.
3. There might be multiple services in an environment matching a particular request. Thus the tool should be able to match a request with a set of advertised services, which might be advertised in a distributed manner.
4. As a multitude of service might be present in an environment, some service might be more specific than the request and some might be more general. Thus the tool should provide results as a set of services with a ranking for each service in the set, depending on the degree of match of the request and the advertised service.
5. It's possible that the service description capabilities are enhanced with the specification of languages which support additional parameters for services. The tool should have an extensible design, in terms of having the capability to plug-in new defined plug-ins for incorporating matching over these newly define additional parameters.
6. For a user requesting a service, the response times should be as little as possible. The tool should have a response time of the order of milliseconds for responding to the user requests.

Based on these design considerations elicited, the distribution layout of the service matching tool as it fits into the Amigo environment is shown in Figure 4-18. As the reasoning functionality needs to be separated from the matching functionality, a separate entity is designated to run a reasoner instance providing a uniform access interface such as DIG [DIG]. Multiple instances of the matching tool are proposed so that matching can happen simultaneously at multiple locations – either on a fixed node such as a desktop, a portable device such as a laptop or on a mobile device such as a PDA or a smart phone. The fact that there are multiple instances of the service matching tool will distribute the matching load on any particular instance. A storage repository is designated for the task of containing the local versions of ontologies that are being used to describe services. This will remove the need to access ontologies over the web repeatedly and thus greatly improve the query response time of the matching procedure.

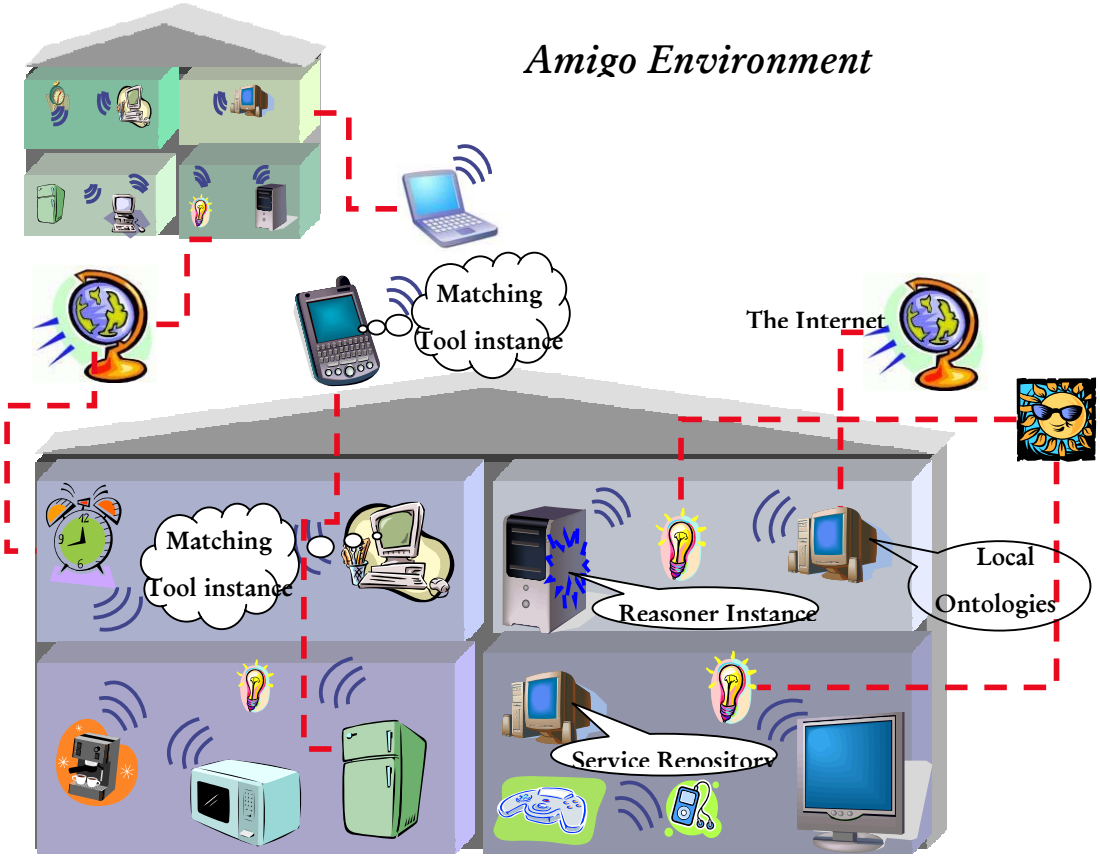


Figure 4-18: Architecture of the matching tool in the Amigo context

A logical representation of a published service repository without binding it to any physical representation is shown. The repository might be a central, distributed or semi-distributed in its design and implementation. As mentioned in the design requirements, in presence of a repository, the service matching tool needs to match the request over the whole set of services published in it. There might be no repository as such and a device providing a service might itself act as a publisher, while devices receiving such advertisements store them in their local repository. Then the task of matching needs to be integrated with a discovery protocol, into which we will be looking in the future.

Semantic Web Services rely heavily upon XML [XML] data for various purposes such as for publishing the service, describing the service functionality using concepts from external ontologies and how to communicate with the service etc. One of the basic tasks to be implemented for working with semantic Web services is to parse XML files for various purposes such as discovery, matching of services and finally invocation. In our case, we particularly access OWL files specifying the profile of a service. There are many approaches proposed and implemented for parsing XML files. The most used implementations are DOM [DOM] and SAX [SAX]. Although both theses models have their own pros and cons, they are not suitable for application requiring just accessing a part of an XML document, e.g. extracting just the profile in our case. DOM provides extensive control over manipulating data in a XML file, and it is memory-intensive, whereas SAX being extremely memory-efficient provides almost negligible support for manipulating data. Also as SAX implements the push model for parsing XML it does not provide a very scalable solution for parsing to extract large chunks of data.

A recent approach called Streaming API for XML (StAX) [StAX] is based on the pull model for parsing XML and provides an extremely memory efficient and flexible solution satisfying our requirements for low memory consumption and flexibility over data manipulation. For our implementation purposes, we have used StAX for parsing the service profiles of the advertiser and the requester.

Figure 4-19 depicts the subsystem design and processing flow of the matching tool. The processing steps are listed below. The numbers in the figure correspond to the step numbers in the list. For each service published in the repository:

1. Extract the profile of the web service from the web service description
2. Extract the inputs and outputs used by the service from the profile
3. Access the ontologies used by inputs and outputs of the selected service and create a memory model for the ontology
4. Set the memory model as a knowledgebase for the reasoner
5. Classifying the ontologies using a Description Logic (DL) Reasoner
6. Extracting information about individual concepts from the memory model
7. For all inputs and outputs of the request and advertised service
8. Matching outputs of the advertised service to the outputs provided by the request, using the DL Reasoner
9. Matching inputs of the advertised service to the inputs provided by the request, using the DL Reasoner
10. Get the matching relation between the matched concepts in the DL Reasoner
11. Return the result of the matching the individual concepts
12. Assign a rank to the advertised service, depending on the degree of match between the request and the advertised service

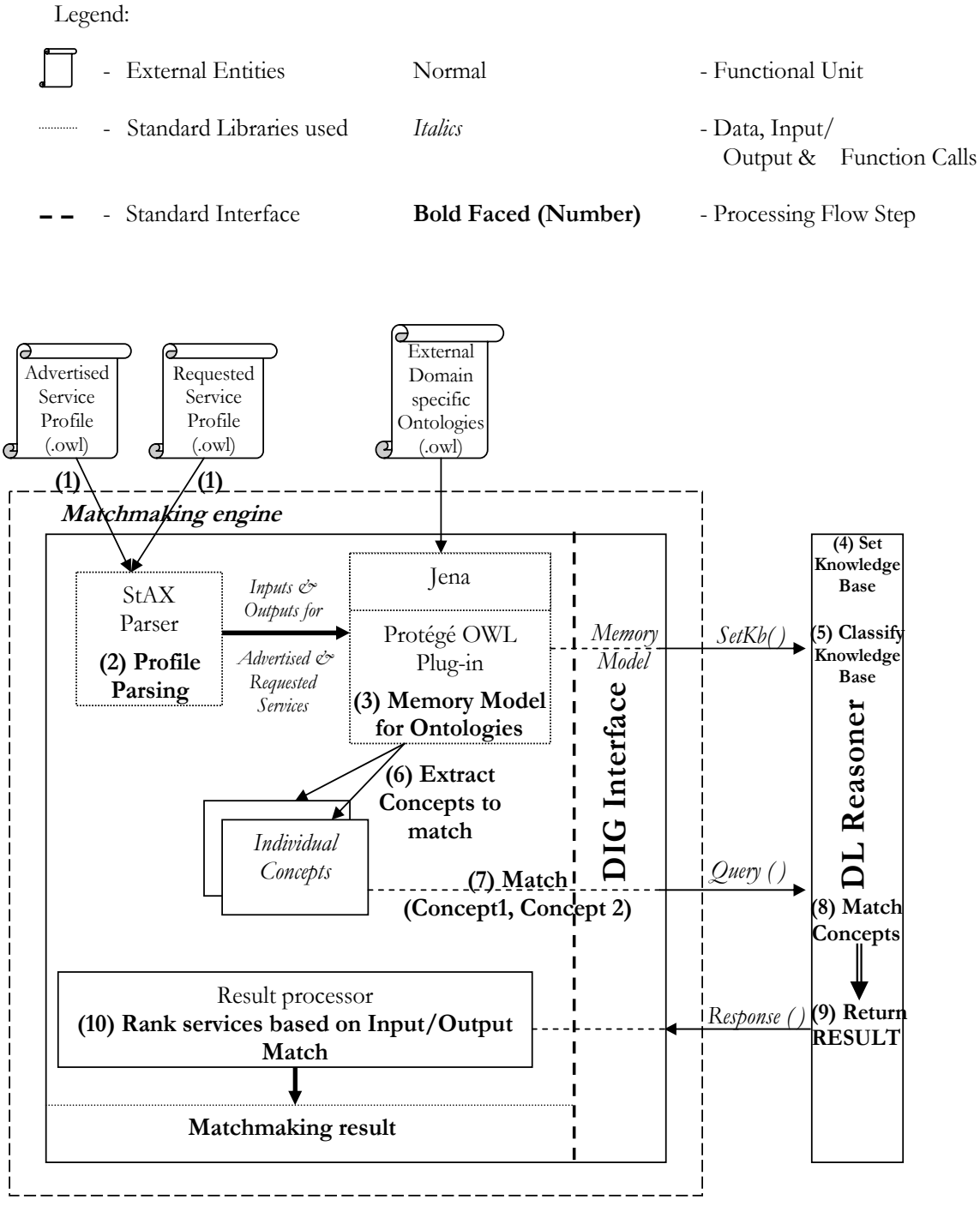


Figure 4-19: Subsystem design and processing flow of the matching tool

The current implementation however matches one service against another one and returns the degree of match of between the advertised and the requested service. In order to execute the aforementioned steps, some standardized libraries were used. For implementing the calls to the DIG interface of the reasoner, the Protégé OWL Plug-in [OWLPlugin] was used as it implements APIs which can be used to communicate over the DIG interface with a reasoner. Other functionalities offered by the plug-in help in handling OWL ontologies. The API gives complete control over manipulating and extracting information from OWL ontologies. The

Protégé OWL Plug-in uses Jena [JENA] for creating the subject-verb-object RDF [B2004] triples.

The enhancements for implementing the matching over a repository needs to be worked out depending on the design and implementation of the service registry and the discovery protocol used.

Implementation & performance evaluation

The implementation was done in order to have a proof-of-concept prototype, where the proposed architecture of the tool to suit the Amigo context could be evaluated in terms of performance. The basic algorithm as detailed in Section 4.2.1.2 was used.

The experiments conducted were aimed at checking the efficiency of the tool in terms of query response time. As an actual AmI environment was not available for testing, some localizations were made to the prototype implementation.

1. The ontologies used were made available on a local web-server
2. The reasoner instance and the matching tool instance was running on the same machine
3. Requested functionalities were simple OWL files conforming to the OWL-S profile containing a list of inputs and outputs without mentioning any details about the process model and grounding

The number of inputs and outputs used by the advertised service and the request were varied and results for the response times taken by the tool to match the request and the advertised service were noted. For noting the times the System.currentTimeMillis() call of the Java API was used and all the experiments were conducted on a Toshiba Satellite notebook with 1.6 GHz Intel Centrino processor and 512 MB of RAM. For the Java Virtual Machine the Java Runtime Environment v. 1.5.0_02 was used. The code snippets for various operations done during matching are given ahead.

- Create the memory model from an ontology

```
OWLModel model = ProtegeOWL.createJenaOWLModelFromReader
(new URL(ontology_URL));
// creates a Jena memory model from the given URL/URI of the ontology
// defining the concepts
```

- Create a logical reasoner instance to representing the actual running reasoner

```
ReasonerManager reasonerManager = ReasonerManager.getInstance();
```

- Set the memory model as a knowledgebase of the reasoner

```
ProtegeOWLReasoner reasoner = reasonerManager.getReasoner(model);
// set the Jena model created as the knowledgebase of the
//reasoner. The model then resides inside the reasoner
//and multiple knowledgebase's are identified by unique URI's
```

- Extract concepts to be matched from the memory model

```
Input reqInput = (Input) requestedInputsIterator.next();
// Extract inputs from a list of inputs
OWLNamedClass reqInputOWLClass = model
        .getOWLNamedClass(reqInput.getRestrictedTo()
```

```
                .toString().substring(
        reqInput.getRestrictedTo().lastIndexOf("#")+ 1));
// Get the OWL information associated with the input which is
// used to match the concepts
// Similar operation are executed for extracting outputs. These
// operations are executed over a loop till information about all
// the inputs and outputs are extracted
```

- Match the concepts (inputs/outputs) with the help of the reasoner

```
int result = reasoner.getSubsumptionRelationship(
                                reqInputOWLClass, advInputOWLClass, null);
// Get the subsumption relationship between the two concepts that
// are being matched
```

- Determine the result of the match

```
if (result == ProtegeOWLReasoner.CLS1_EQUIVALENT_TO_CLS2) {
// necessary processing done to modify the rank of the match with
// reference to the fact that the two concepts are equivalent
}
if (result == ProtegeOWLReasoner.CLS1_SUBSUMED_BY_CLS2) {
// necessary processing done to modify the rank of the match with
// reference to the fact that the requested concept subsumed the
// advertised concept }
if (result == ProtegeOWLReasoner.CLS1_SUBSUMES_CLS2) {
// necessary processing done to modify the rank of the match with
// reference to the fact that the advertised concept subsumed the
// requested concept
}
if (result == ProtegeOWLReasoner.NO_SUBSUMPTION_RELATIONSHIP) {
// necessary processing done to modify the rank of the match with
// reference to the fact that there is no relationship between
// the concepts being matched
}
```

The results for various reasoners in terms of increasing number of inputs and outputs are shown in Table 4-8, Table 4-9, and Table 4-10. A large difference that can be seen in the parsing times of requested and advertised profiles is due to the simplicity of the request files which did not contain much information. For the parsing of the profile in the last set of experiment with 10 Inputs and 4 outputs, both the advertised and the requested service profile were accessed online from the Web, whereas for the other set of experiments the advertised profile was available on a local Web server and the requested profile was a local file.

Number of Inputs and Outputs	Time taken to parse advertised profile		Time taken to parse requested profile		Time taken to create memory model		Time taken to match services	
	Avg.	SD	Avg.	SD	Avg.	SD	Avg.	SD
2 Inputs 2 Outputs	178.4 ms	11.5	20 ms	0	3717 ms	160.6	395 ms	28
4 Inputs 2 Outputs	182.2 ms	12	20 ms	0	3688 ms	62.4	567 ms	39.1
7 Inputs 3 Outputs	190 ms	10	20 ms	0	3720 ms	205.2	995 ms	61.5
10 Inputs 4 Outputs	1227 ms	30	661 ms	48.1	3687 ms	85.1	1471 ms	27.5

Table 4-8: Times taken to match a request and a service using RACER

Number of Inputs and Outputs	Time taken to parse advertised profile		Time taken to parse requested profile		Time taken to create memory model		Time taken to match services	
	Avg.	SD	Avg.	SD	Avg.	SD	Avg.	SD
2 Inputs 2 Outputs	160 ms	14.3	20 ms	0	3715 ms	153.6	421 ms	22
4 Inputs 2 Outputs	190 ms	9.3	20 ms	0	3525 ms	167.3	551 ms	37.1
7 Inputs 3 Outputs	170 ms	10	20 ms	0	3685 ms	200.2	871 ms	51.5
10 Inputs 4 Outputs	861 ms	27	461 ms	42.1	3595 ms	185.3	1252 ms	26.5

Table 4-9: Times taken to match a request and a service using FaCT++

Number of Inputs and Outputs	Time taken to parse advertised profile		Time taken to parse requested profile		Time taken to create memory model		Time taken to match services	
	Avg.	SD	Avg.	SD	Avg.	SD	Avg.	SD
2 Inputs 2 Outputs	160 ms	14.3	20 ms	0	3926 ms	157.7	472 ms	21
4 Inputs 2 Outputs	171 ms	9.7	20 ms	0	3675 ms	162	561 ms	42.1

7 Inputs 3 Outputs	180 ms	11.5	20 ms	0	3945 ms	197.2	982 ms	29.3
10 Inputs 4 Outputs	941 ms	22	491 ms	38.1	3746 ms	173.3	1542 ms	26.7

Table 4-10: Times taken to match a request and a service using Pellet

The obviously expected trend that can be observed is that as the number of inputs and outputs increase the time to match increases. This is depicted in Figure 4-20.

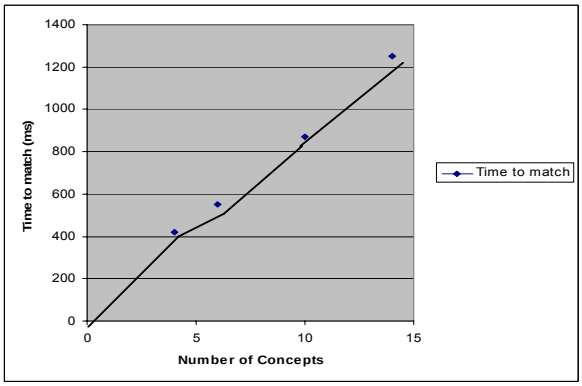


Figure 4-20: Times taken to match a request and a service using FaCT++

The time to respond to a users query for matching of services is of the order of 4-5 seconds. These times are not optimal ones. The time to match will further increase, if the matching is done over a set of services published in a repository. The prototype tool implemented is an online one, where all the processing happens once a request is launched to match services. Keeping in mind, that the query response time should be as little as possible, we need to consider some optimizations. These optimizations are intended as a future work and are discussed in the next section.

4.2.1.4 Discussion

Service matching is an essential functionality in the Amigo environment. Matching of services based on semantic information provided in a service description is the main goal of this work. Further, for any software tool to be useful in an AmI environment, it needs to be lightweight in terms of resource consumption such as memory and processing power required to execute it. In the research community, some service matching tools based on semantics of services are available. The goal of the work presented in Section 4.2.1 has been to evaluate these current approaches and discuss their suitability for the Amigo environment. Based on the evaluation of these tools, a new service matching tool suiting the Amigo context has been designed and a basic prototype for this tool has been developed.

More specifically, the currently available matching tools for semantic matching of web services were listed and evaluated from a systems perspective. We studied the base matching algorithm followed by all the listed tools. As a Description Logic (DL) reasoner is an inseparable part of any tool designed for semantic matching, various DL reasoners currently available were listed and evaluated. The evaluation of the current matching tools brought out drawbacks and features in their design and implementation. The evaluation of the DL reasoners helped us in eliciting the properties of an appropriate reasoner that could be suitably used in the Amigo environment. Based on the evaluations and keeping in mind the

requirements of a tool that suits the AmI context, some observations were made about the design of a tool that would fit in the AmI context. The major contribution of this work is the detailed analysis of the matching process from a system point of view, clearly eliciting the various steps involved in the matching process and the analysis of the costs associated with them in terms of time and memory consumption. Consequently, the architecture of a tool in which the matching functionality is decoupled from the reasoning tasks was proposed to suit the AmI context. For evaluating the performance of the proposed approach, a prototype implementation was undertaken, and preliminary results of the implementation were presented in terms of query response times.

As discussed in Section 4.2.1.3, the response times for matching are not optimal. Hence, some optimizations need to be done. The prototype tool implemented is an online one, where all the processing is carried out once a request is launched to match services. In order to meet the requirement that the query response time should be as little as possible, some optimizations that can be applied are:

1. Grouping of services according to some predefined standards such as UNSPSC – The United Nations Standard Products and Services Code [UNSPSC] or NAICS – North American Industry Classification System [NAICS]. These standards provide taxonomies which separate businesses and services into specific categories and provide unique codes for individual entities listed in these taxonomies. Once the services are grouped using some classification standard and when a request is received which contains a code from the classification system, we need only to match the group of services which corresponds to the same code category. E.g., if a request is received to with a code referring to a ticket selling service, only services which belong to the ticket selling services category need to be matched. This would considerably reduce the time to match as we can restrict matching to only a set of services, in case there are a large number of services present in the environment.
2. Enhancing matching by trying a syntactic match between the URIs of the concepts being matched before requesting the reasoner to find a relation between the two concepts. This implies that in case two concepts that are being matched are referring to exactly the same concept in the same ontology, a syntactic match i.e. a simple string comparison can be evaluated for their equivalence. This would avoid, use of a reasoner for matching, thus saving on time to match. However, this could only check concept equivalence and cannot assert subsumption relationships.
3. Dividing the tool into an offline and an online component. The offline component of the tool could perform some of the following functions
    - Pre-fetch or simply cache ontologies; additionally, create from them memory models and store them locally as memory objects; thus when a model, it is immediately available; the model creation time from parsing files and extracting the required information is spared.
    - Once the memory model is created, these models can further be classified and then stored as objects. This will further reduce the time to match as classification is a time intensive process. As classification needs to be done only once, it can be delegated to the off-line component.
    - Pre-parsing service profiles as and when they are created or imported from external sources, and storing them as memory objects. This will in turn save the time for parsing profiles.

Further, in order to take advantage of the multitude of information provided in the service profile, matching of Preconditions and Effects is one of the tasks that need to be investigated. Matching of Preconditions and Effects would lead to an increase in the probability that the matched service actually supports the functionality which is sought by the requester.

Moreover, currently the tool matches a request against one service; this functionality needs to be enhanced, so that the tool matches and ranks accordingly services in a repository. This functionality was not undertaken under the current work as this enhancement needs the integration of the tool with a service discovery protocol and associated service publishing repository in the Ambient Intelligence environment. The service publishing repository can be either centralized, distributed or may not exist at all as the service hosts might act as publishers themselves. In the light of this requirement, the design and realization of the matching tool should be sufficiently generic to allow integration with different service discovery protocols and repository architectures.

4.2.2 Context-aware service discovery

One of the most pressing issues in AmI environments is that of ever-changing context. This applies equally well to mobile devices as to the (fixed) services they use: mobile devices are regularly subject to location, network, and power context changes, whilst services can for instance be subject to changes in the types of devices they have to serve. In the Amigo AmI environment, such changes should result in a service being dynamically adapted to the new context of a mobile device or of the service itself.

The *Context-Aware Service Discovery (CASD) service* is our proposition towards dealing with context as part of the Amigo-aware service discovery mechanism. In the following sections, we assume that there is a *base service discovery service* available, which we enhance with context-awareness. We further elicit specific requirements towards the base service discovery service. The new features that we introduce into the base service discovery service concern discovering context sources and using them during service discovery to optimize the discovery process (Sections 4.2.2.2 - 4.2.2.6). However, first, we will briefly introduce some key context-awareness concepts that will be used by the CASD (Section 4.2.2.1).

4.2.2.1 Context sources and brokers

A *context source* is a service that provides access to context information, such as the location of a user or the activity a user is currently engaged in. Context source clients can directly access context information via a request-response interaction or by subscribing to events that signal a change in context information (e.g., when a user moves from one room to another).

A *context broker* is a service that provides a single point of access to the context information about a particular object (e.g., a device, a user, or a service). It also acts as a container for the (potentially composite) context sources that can provide this information.

4.2.2.2 CASD functions and interfaces

Figure 4-21 shows the CASD discovery model. It adds context sources to the classical model of service discovery by associating a context broker with each ‘object’ (e.g., a device, a user, or a service) whose context needs to become discoverable. Since a context broker is a service, it registers with the base service discovery service to become discoverable.

As with established discovery services, the CASD service provides three interfaces:

- A registration interface, which enables services to become discoverable by registering their descriptions with the base service discovery service;
- A discovery interface which allows discovery clients (the base service discovery service or the clients of the platform) to find services by matching their discovery requests with the descriptions of registered services; and
- A bootstrapping interface, which clients and services use to discover the base service discovery service (we assume this has been addressed by the base service discovery service).

It is expected that the CASD service will be tightly integrated with the base discovery service and as such, will inherit some of this functionality. The CASD service exposes these interfaces to services and context sources.

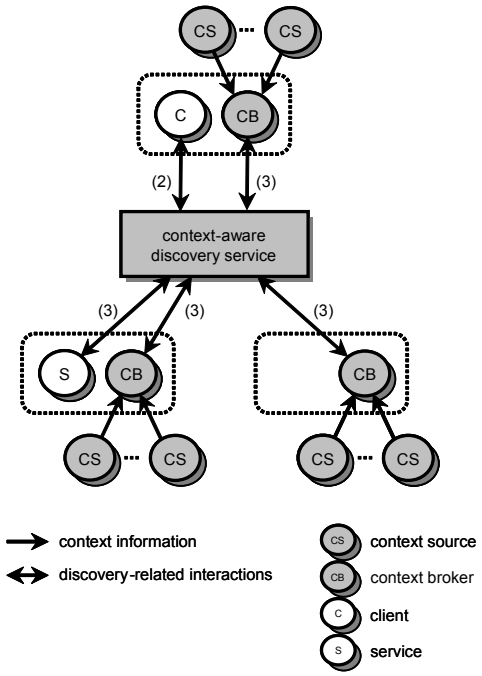


Figure 4-21: CASD service discovery model

The discovery interface supports active and passive discovery. In *active discovery*, discovery clients actively request the discovery of certain services and context sources, whereas in *passive discovery*, they wait for the base service discovery service to push such services and context sources to them (on a subscription basis). Passive discovery is particularly useful when a discovery client is constantly looking for 'better' services or context sources. Passive discovery might for instance be useful when a client is moving throughout the Amigo home and is continuously looking for a display service that is nearby, thus taking into account the client's current context.

4.2.2.3 Active discovery interface

The active discovery aspect of the discovery interface consists of a discovery request primitive and a discovery response primitive. As in traditional service discovery, clients use a discovery request to invoke discovery and subsequently receive a response that contains references to matching services. The request contains the usual parameters, which are a semantic specification of the services the client is trying to find (e.g., transcoding services), a set of constraints (e.g., transcoders that support MP3 audio), and a description of the scope in which the base service discovery service should look for matches (e.g., in terms of a geographical area or a number of network hops).

The CASD-specific parameters in the discovery request are:

- A client context specification, which describes the context information of the discovery client and either consists of actual context information (obtained from the client's context broker) or of a reference to the client's context broker. The client context

specification is optional because some clients may not aware of having a context broker.

- An (optional) set of additional constraints that describe the context that prospective services should to be in (e.g., printing services that must be located in a certain room).

Each of the references in a discovery response primitive comes with a description of the corresponding service, which enables the client to more intelligently select the most appropriate service out of a number of alternative matches.

4.2.2.4 Passive discovery interface

The passive discovery aspect of the interface consists of three primitives: a persistent discovery request, a persistent discovery response, and a persistent discovery notification. A persistent discovery request is essentially an active discovery request that has a specified lifetime. Discovery clients use a persistent discovery request to instruct the CASD service to generate a discovery notification when it discovers services that are 'better' than the ones it proposed in previous notifications. Before issuing such notifications, the discovery system first confirms the receipt of the discovery request by passing a persistent discovery response back to the discovery client.

With passive discovery, the scalability of the CASD service is an important concern, because it needs to maintain state for each outstanding persistent discovery request. The base service discovery service therefore uses softstate persistent requests, which means that it removes the state associated with a persistent request unless that state is refreshed before a specified time. At the protocol level, the base service discovery service can accomplish this by utilizing lease mechanisms.

The parameters of a persistent discovery request are similar to those of an active discovery request. The differences are that a persistent discovery request contains:

- A reference to the client (e.g., in the form of a URL), so that the base service discovery service can asynchronously deliver discovery callback notifications; and
- An event specification instead of a service specification. The event specification indicates the (type of) events that the client wishes to subscribe to and to which (types of) services.

A persistent discovery response indicates if the base service discovery service successfully served the preceding request.

4.2.2.5 Registration interface

The registration interface of CASD is almost the same as for established service discovery services. The most important primitives are registration requests and registration responses. A service (or context broker) uses a registration request to register with the base service discovery service, which then passes back a registration response.

The usual parameters of a registration request are a service description (augmented with semantic descriptions), a specification of the scope in which the service is available (e.g., a number of network hops or an administratively defined scope), and a self-reference so that discovery clients can actually contact the service.

The CASD-specific parameter of a registration request is the context of the service (optional), either in the form of the actual context information or as a reference to the service's context broker.

4.2.2.6 Approach to realizing Context-Aware Service Discovery

In Figure 4-22, we show an example of the interaction between the Client and the CASD Service. In the example, the client wishes to determine which service (out of Service A and Service B) is "best" for it, given a Service Specification, a set of constraints and also a reference to its Context Source. Note that we assume that Service Registration has already occurred and that Services A and B are known to CASD. Similarly, we also assume that all of the Context Sources shown in the figure have registered themselves with the Context_Mgt_Service.

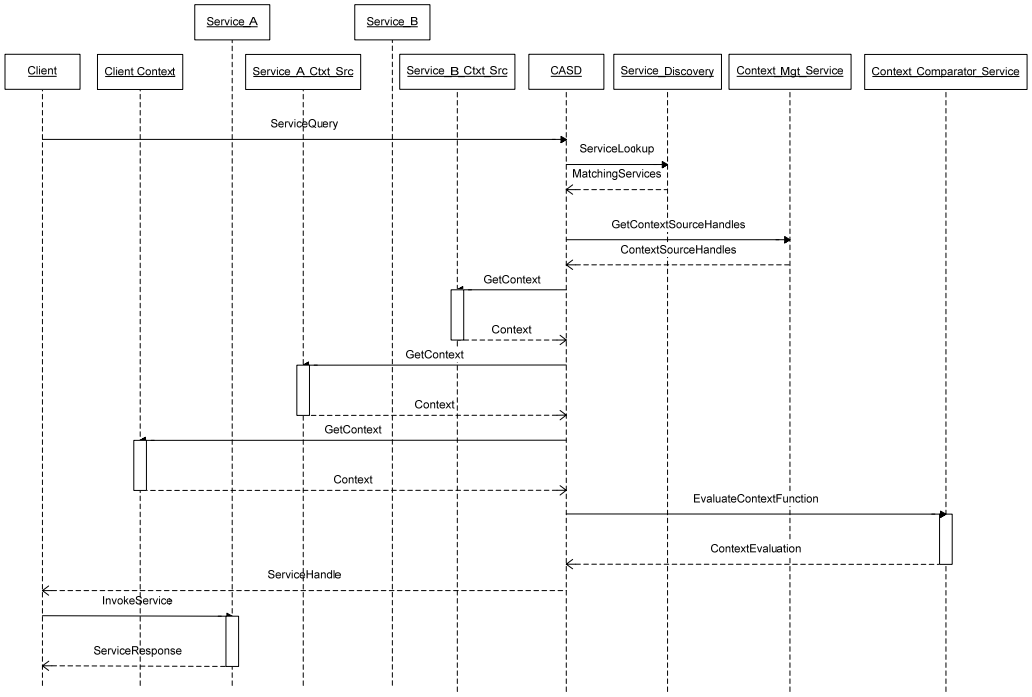


Figure 4-22 Example interaction between the Client and the CASD service

The high-level algorithm which we will use to realize CASD is as follows:

The CASD acts as a "co-ordinator" and conducts the sequence of operations, making use of the existing base Service Discovery service and Context_Mgt_Service to achieve its goal. The steps are as follows:

- The client submits a request to the CASD, indicating the Service Specification that it requires, a constraint expression on the matching and also a reference to its Context Source.
- The CASD uses the base Service Discovery service to obtain a set of registered services that match the provided Service Specification.
- The CASD uses the Context_Mgt_Service to obtain references to the Context Sources of the matched set of Services.
- For each of these Context Sources, the CASD requests the current Context.

- With these Contexts, the CASD then instructs the Context Comparator Service to determine which of the services is the best match, given the Client-supplied context constraints.

Finally, the CASD hands a Service Reference handle back to the Client which is then free to interact with the Service as it sees fit.

4.2.3 QOS- and resource-aware service selection

A plethora of services will eventually be deployed in the AmI home. Many of these services will be offering similar functionality. For serving a specific service request of a user, the AmI 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, the selection process will depend heavily on these parameters and will also consider the number, features and identity of the services that are already selected and/or available. Additionally, another issue that may have a significant impact on the performance of the AmI system is the fact that multiple users may concurrently submit several service requests to a *server* residing on a networked home device, expecting the services to be delivered at the same time, each being compliant with the user preferences. However, as the resources of the AmI middleware and the server in question will never be unlimited (e.g., with regard to capacity, bandwidth, processing capabilities, storage), a service selection tool must be established to decide on the services that will eventually be delivered, and on their configuration and properties, so that the system resources are used in the best possible manner, while users enjoy the services that address their requirements as much as possible.

Hereafter, an illustrative example is described that aims to clarify the grounds of this service selection optimization problem. Let's assume that in the Amigo networked home there is a LAN established and that there is a 1024 kbps DSL line shared by all networked devices. There is an incoming request of an Amigo user for playing a game on the Internet. This gaming service is offered in various versions that require different bandwidth rates (e.g. 56 kbps, 128 kbps, 256 kbps and 384 kbps). In order for the Amigo middleware to decide which service version is the most appropriate, it has to consider several parameters. First, it has to filter out the service versions that do not address the user requirements (e.g., with regard to bandwidth, price, image resolution). Then, it has to discover if other service requests are in place and which ones, and consider the resources that will potentially be consumed by the relevant service deliveries. At this point, the system needs to select the services to be delivered considering the service features, user requirements and resource constraints. Even if the system resources are enough to satisfy all service requests, the service selection process is still necessary in order to ensure that the Amigo users' objectives and needs are efficiently fulfilled.

The proposed service selection tool will be implemented by the Amigo Middleware, which will reside on the networked home devices. Among these devices, there is the Gateway that enables the Amigo Home to be connected to the Internet. In Figure 4-23, the suggested service selection process is illustrated, along with the involved modules/actors. Upon the reception of a new service request, the *Service Matching Tool (SMT)* (see Section 4.2.1.3) discovers the available services and, based on a semantic matching mechanism, it filters out the ones that do not meet all the requirements of the service request. The set of the filtered services are then delivered to the *Service Selection Tool (SST)*, which reduces to one the service set size based on an efficient selection algorithm that considers the user requirements/constraints, the current status of available network resources and the features of the on-going service sessions. Thus, in step (1) of Figure 4-23, an additional service request (Req. 5) is submitted by an Amigo User. Notice that at that time, there are already four on-going sessions for services that were previously requested (Req. 1 – Req. 4). In step (2), the SMT identifies the available services that address all the requirements of the new service

request. Finally, in step (3), the SST determines the services that will be finally selected for serving all service requests of the Amigo Users (i.e., Req. 1 – Req. 5), reevaluating the service selection performed upon the reception of previous requests.

From the aforementioned analysis, it is clear that not all the requests can be served in a real home environment, where the available resources (e.g., capacity, bandwidth, processing capabilities, storage, etc.) are limited. Thus, we will quite often face the problem of not having enough resources to address all the users' requirements. Thus, not every service request is always possible to be served, or at least not in the most preferable service version for the users. The proposed SST will depend heavily on priorities. We adopt a priority-based selection model as it is desired: (i) to serve as many requests as possible, in order to satisfy the majority of Amigo users; and (ii) to firstly serve all requests carrying a higher priority. For example, safety-related requests should be served in any case, and should thus have the highest possible priority. The proposed priority model consists of two levels. The most important level (first level) depends on the kind of service that is requested (e.g., safety, gaming, information, entertainment, etc.). Of course, safety-related services are assigned with higher priority than the entertainment-related services. So, in the figure above, request 5 will be served first with regard to requests 1 or 2. The second level (less important level) of the proposed priority model depends on the person who submits the request. In this case, parent-originated service requests, for example, are assigned with higher priority than children's requests. In the future, additional priority levels may need to be distinguished.

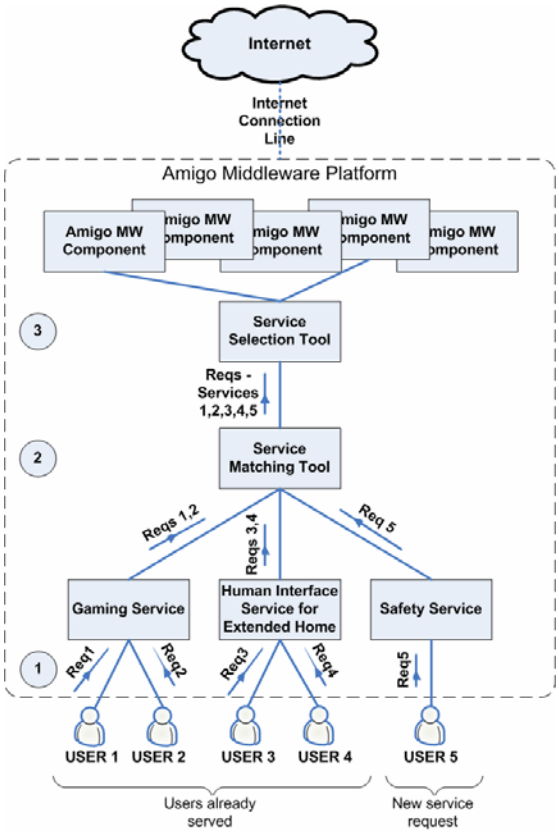


Figure 4-23: The Service Selection process

The proposed SST algorithm will initially determine the priority of each request. In order to calculate these priorities, a binary representation of the number I_i of the priority instances of each level i is used. The number of bits assigned at each level is estimated based on the population of the priority instances of this level. For example if we have 19 instances under

priority level 2, then the binary representation of this level's priority will require 5 bits (19:10011). Thus, the combined priority (*combined_prio*) of each service request is calculated based on the following equation:

$$combined\_prio = \sum_{i=1}^{k} prio_i 2^{A_i} \ , \ A_i = \sum_{j=i+1}^{k} a_j \qquad (1)$$

Where $prio_i$ is the service priority for level *i*, a_i represents the number of bits required for the individual priority binary representation of level *i*, A_i indicates the number of bits required for the combined priority binary representation of level *i*, and k represents the number of priority levels. The above are illustrated in the following example: Consider the case of four main priority levels. The first level has 23 instances ($\rightarrow a_1 = 5$), the second has 35 ($\rightarrow a_2 = 6$), the third has 12 ($\rightarrow a_3 = 4$), and the last has 19 instances ($\rightarrow a_4 = 5$). Based on Equation (1), we have: $A_1 = 15$, $A_2 = 9$, $A_3 = 5$ and $A_4 = 0$.

The service selection problem can now formally be stated as follows: Given *N'* service requests from Amigo Users and given all the available services that address their requirements, select the most appropriate set of services to be delivered so that the maximum possible number *N* of requests is served, the total priority-weighted cost is minimized, while the overall bandwidth of the selected services does not exceed the one provided by the established infrastructure. This can be reduced to the following linear programming problem:

$$\text{Objective function: min} \quad \sum_{i=1}^{N}\left[\sum_{j=1}^{M_i}\left[(c_{ij} / combined\_prio_{ij})S_{ij}\right]\right] \qquad (2)$$

$$\text{Restrictions:} \quad \sum_{i=1}^{N}\left[\sum_{j=1}^{M_i} b_{ij} S_{ij}\right] \le B \qquad (3)$$

$$\sum_{j=1}^{M_i} S_{ij} = 1 \ , \ i = 1,2,...,N \qquad (4)$$

$$S_{ij} \in \{0,1\} \ , \ i = 1,2,...,N \ , \ j = 1,2,...,M_i \qquad (5)$$

Where *N* is the overall number of concurrent service requests, M_i is the overall number of the services that address the requirements of service request *i*, S_{ij} is the decision variable for service *j* that addresses the requirements of service request *i* b_{ij} is the required bandwidth by service *j* for service request *i*, c_{ij} is the corresponding service cost and $combined\_prio_{ij}$ is the combined priority of service *j* for service request *i*. This is a minimization problem seeking to minimize the overall cost. The restrictions of this problem suggest that: the overall bandwidth of the selected services does not exceed the available bandwidth B (Equation 3), every request is served (Equation 4) and the decision variables are Boolean (Equation 5), i.e., $S_{ij} = 1$ in case service *j* is selected to serve service request *i* or $S_{ij} = 0$ otherwise.

It stands that $N \le N'$. The number N of requests that can be served simultaneously can be estimated based on the priority model defined above. The estimation process is as follows: First, the priority of each service request is calculated. Then, the service requests are order based on their priority (i.e., *i=1* for the highest priority and *i =N'* for the lowest). For each service request *i*, the lowest bandwidth service *I* is selected in the set of services that address the request's requirements. N is provided by the following equation:

$$N = \left\{ \max k : \left[\sum_{i=1}^{k} b_{il} \right] \leq B \right\} \quad (7)$$

Of course, in case $\sum_{i=1}^{N'} b_{il} \leq B$, then $N = N'$.

After having identified an initial solution to our service selection problem, we will refine our solution in order to reduce the overall priority-weighted service delivery cost. The problem is currently being studied. The solving algorithm currently under evaluation is inspired by previous research on problems of the Knapsack family [MT90].

4.3 Discussion

In this chapter, we addressed Amigo-S, the Amigo semantic service description language, and several aspects of the Amigo service discovery. These are two topics that are closely interrelated, as the main objective of defining formal semantic description of Amigo services is to enable their automated discovery in the open, dynamic Amigo home environment, where no *a priori* knowledge of existing services can be assumed. However, as Amigo service discovery is still at an early design stage, we currently elaborated on a number of its aspects in a disconnected way, and at a generic level where the integration of the service description language is not yet needed.

More specifically in this chapter, we formally specified Amigo-S, by introducing a number of OWL classes that complement the OWL-S specification, mainly towards: (i) enabling description of services belonging to different service technologies (besides Web Services) by adding specification of the underlying middleware; and (ii) enabling description of non-functional service properties, such as QoS and context. Further, we addressed service matching in the Amigo environment: we evaluated existing semantic reasoning and service matching tools, and proposed an architecture for a service matching tool suitable for the Amigo environment; an early prototype of this tool allowed us to obtain a first performance evaluation and to identify necessary optimizations. We then discussed context-aware service discovery in Amigo, and outlined an architecture that enhances a basic service discovery mechanism with context-awareness; we elicited as well a set of relative requirements for this basic discovery mechanism, which should be eventually supported by the Amigo service discovery. Finally, we addressed the problem of service selection – among a set of suitable discovered ones – on a networked server in the Amigo home, which receives a number of concurrent service requests and has to satisfy them in an optimal – related to user preferences and QoS – way for users, respecting the existing resource constraints.

Following the current developments, our next step is to integrate these efforts in a consistent architecture enabling semantic description of Amigo services and related service discovery including service matching and service selection, particularly taking into account context and QoS features of services.

5 Service discovery and service interaction interoperability

Service discovery interoperability (SDI) and service interaction interoperability (SII) are two key functionalities provided by the Amigo middleware core, which enable integration of heterogeneous devices and hosted services in the networked home environment.

In Deliverable D3.1b [Amigo-D3.1b], we carried out detailed design and first prototype implementation of SDI. Interoperability between the SLP and UPnP service discovery protocols is supported, while our modular design enables the easy integration of other protocols. We further provided evaluation of our prototype in terms of implementation footprint and performance. We elaborated our design in UML at a programming language-independent level and carried out our first prototype implementation in Java.

In the same deliverable, we elaborated an early design and implementation of SII. This was essentially a case of study to test our solution for a special case of configuration of client and service interaction protocols. More specifically, in this first design of SII, we addressed only a special case of client/service configuration: the client is RMI-based and the service is UPnP-based. Further, the internal mechanisms of the generated *proxy* to implement interaction interoperability are not based on protocol units (and their related components, that is, parsers and composers) and semantic events mechanisms. The simplified solution that we adopted is to generate a proxy that has the client interface (RMI) and for each method contains the code to generate directly UPnP calls to the remote service. A subset of the UPnP stack to make RPC calls must be available on the client. As in the case of SDI, we elaborated our design of SII in UML and carried out the implementation in Java.

Both our implementations for SDI and SII were successfully incorporated in the Integrated Prototype demonstrated at the first Project Review. This integrated prototype provided a first, proof-of-concept integration of several interoperability mechanisms across the Amigo domains, i.e., the PC, mobile, domotic and CE domains.

Building on our results reported in D3.1b, our further elaboration on SDI and SII led us to the decision to employ C as the programming language for SDI and SII in the place of Java. This decision is based on our goal to make SDI and SII independent of any platform and any execution environment, such as the JVM in the case of Java. Thus, our current work is twofold:

- We are working on porting our Java-based implementation of SDI into C.
- We have elaborated a detailed design of SII and provided an early implementation for evaluating its performance, following right from the beginning a C-based approach. This detailed design covers both cases of client- and service-side RMI and SOAP interaction protocols.

In this document and chapter, we report on the latter work, introducing the NEMESYS (NEtwork MEtacommunication SYStem for middleware interoperability) interoperability system, which enables any application in the open networked environment, to interoperate with any networked service, irrespectively of their underlying communication protocol. As for the Amigo solution to SDI presented in Deliverable D3.1b, NEMESYS builds upon event-based parsing techniques to achieve efficient on-line protocol translation. NEMESYS is currently focused on RPC-based communication protocols, as they are still the most widely used to access services in the networked home. Nevertheless, NEMESYS is designed to further support other styles of communication like asynchronous eventing. NEMESYS provides efficient interoperability between networked devices, including resource-constrained ones, without requiring any

change to applications/services and related middleware. Indeed, NEMESYS transparently interposes at the network level.

In the following, we outline why existing middleware do not address efficiently the heterogeneity issue of open networked environments (§5.1). This leads us to introduce NEMESYS that transparently enables devices to interoperate, without requiring any change to hosted applications and their related communication protocol (§5.2). We then show how middleware interoperability is achieved in open networked environments using NEMESYS (§5.3). To validate the design of NEMESYS, we have developed a first prototype, which is both platform-agnostic and efficient (§5.4). Finally, we summarize our contribution (§5.5).

5.1 Background

In a dynamic open networked environment like the Amigo networked home, devices need to adapt themselves to the context by switching, for instance, on the fly, their communication protocol. This is currently not feasible as the way applications are designed depends strongly on the middleware upon which they are developed. Thus, applications can not be decoupled from their underlying middleware. For instance, considering RPC-based communication, a RMI client needs to be redeveloped and bound with a CORBA middleware to interact with a CORBA service. Dually, a CORBA service needs to be redeveloped with a RMI-based middleware to interact with a RMI client.

The above issue outlines the need for a system enabling interoperability among heterogeneous middleware. We have identified four basic requirements for such a system:

- First, interoperability must be available to all kinds of devices, including resource-constrained ones. Hence, the cost of interoperability in terms of resource requirements (i.e., CPU, memory, and network bandwidth) must be reduced to a minimum.
- Second, depending on the context, networked devices may act either as consumers or providers of services. Interoperability must be effective whatever the behavior of devices. In other terms, the interoperability system should be suitable for both clients and providers.
- Third, still in order to support as many devices as possible, the interoperability system must be independent of hardware, operating systems, or programming environments (e.g., Java, .NET).
- Fourth, interoperability must be provided transparently to any client and service applications without requiring changing the middleware API they use.

Middleware bridges provide interoperability between two middleware, and thus related communication protocols. Bridges can be direct or indirect. Direct bridges[10],[11] provide interoperability between two fixed middleware, whereas, indirect bridges assume the predominance of one specific middleware that acts as an intermediary [SGGB01]. Bridges may appear as an attractive solution to provide interoperability. However, bridges are not suitable for dynamic open networks. Indeed, bridges are a static mean to overcome middleware heterogeneity in a known and controlled networked environment since the bridges to be used must be known in advance. Furthermore, middleware-based applications need to be at least recompiled and redeployed with the networked bridge, and possibly re-written in the case of indirect bridges like RMI-IIOP. Thereby, although our fourth requirement is almost supported for indirect bridges, human intervention is required. And if our first and second requirements are met, this is not the case of the third one for direct bridges: dedicated bridges must be specifically developed for each pair of heterogeneous middleware. Greater flexibility to bridge-based interoperability is brought by Enterprise Service Buses (ESBs), which allow

[10] http://java.sun.com/products/rmi-iiop/

[11] http://iiop-net.sourceforge.net/index.html

integrating various bridges[12]. Still, services need to be made ESB-aware explicitly through the development of wrappers, hence not complying with our second and fourth requirements.

ReMMoC is one of the pioneering middleware introducing an interoperability system for open (wireless) networks [GBS03]. ReMMoC is a reflective middleware that defines a generic communication interface, which hides the communication protocol used to invoke remote services. Consequently, client applications are not aware of the actual communication protocol. Indeed, the latter is dynamically selected by REMMoC, which, thanks to its reflection mechanisms [CBMEG02], chooses the most appropriate communication protocol according to the context. Although ReMMoC is currently one of the most efficient and innovative middleware to perform interoperability, it is confronted to several constraints. First, client applications must be developed using the ReMMoC middleware, which introduces a proprietary API. Thus, applications become ReMMoC-specific. Thereby, interoperability is available only to ReMMoC-based clients, hence violating our fourth requirement. In addition, ReMMoC is dedicated to client applications, excluding so interoperability to service providers. Providing interoperability to service providers, as our second requirement suggests, enables clients, which are not interoperable (e.g., not based on REMMoC) to interoperate with services that are based on a different communication protocol.

In a way similar to ReMMoC, RMIX is a middleware that permits transparent dynamic binding with multiple communication protocols [KWSS03]. RMIX originality comes from its programming model that is based on RMI. Hence, the reengineering of existing RMI applications, to take benefit of RMIX, is reduced to a minimum. However, interoperability still requires human intervention. Additionally, RMIX is dedicated to Java and uses functionalities inherent to the Java platform. Thus, interoperability is restricted to Java-compliant devices and/or services. The need to embed a Java Virtual Machine (JVM) and to rewrite non-Java applications to be interoperable is a strong limitation. As a result, RMIX does not meet our third and fourth requirements. The same applies to OSGi, which is a popular Java-based middleware that provides the capability to integrate different communication protocols for OSGi-specific applications.

Summarizing, from the above survey of existing solutions to middleware interoperability, there is, to the best of our knowledge, no satisfying solution to middleware interoperability in open dynamic networks

5.2 The NEMESYS interoperability system

This section introduces the NEMESYS event-based system for transparently achieving interoperability among heterogeneous communication protocols, focusing on RPC protocols in a first step. Section 5.2.1 first recalls the characteristics of RPC communication protocols in order to introduce, in Section 5.2.2, efficient event-based techniques to overcome protocol heterogeneity. Then, Section 5.2.3 presents the ability of the NEMESYS system to provide transparent interoperability among middleware.

5.2.1 RPC communication stack

According to the OSI model, RPC communication protocols can be decomposed into layers, providing a functional division of the tasks required to enable successful interaction. As depicted in Figure 5-1, RPC communication protocols decompose into 5 layers, defining a reference RPC communication stack. The *network* and *transport* layers are similar to the OSI ones. The former determines how data are transferred between networked devices whereas

[12] http://www.iona.com/products/artix/welcome.htm

the latter specifies how to manage end-to-end message delivery among networked entities. The *invocation* layer, refining the OSI session layer, defines how to manage sessions with remote services across the network and then specifies the types of messages exchanged during an opened session. Then, the *serialization* layer, refining the OSI presentation layer, encodes messages according to a format specification. Finally, the *application* layer provides to applications an interface to perform remote procedure calls.

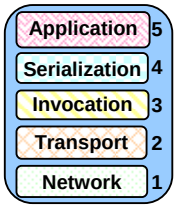


Figure 5-1: RPC communication stack

.

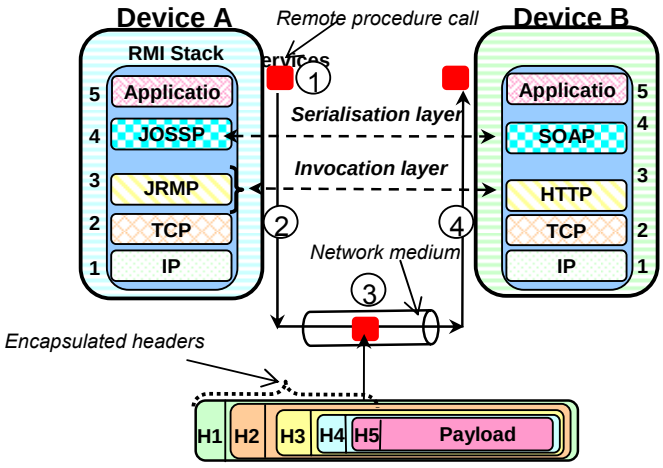


Figure 5-2: Layer-to-layer communication

For illustration, consider a device *A* hosting a RMI communication stack and another device *B* hosting a Web services stack. As depicted in Figure 5-2, if an RMI-based application of *A* wishes to invoke a Web service of *B* (Figure 5-2, Step❶), the corresponding request message passes through the 5 layers of the stack hosted on *A* (Figure 5-2, Step ❷). Specifically, the request is first passed to the application layer, which adds a header to the data. The resulting message is passed to the serialization layer, which adds its own header to the message it just received from its upper layer and so on, all the way down to the IP network layer. At the IP layer, the resulting message is transmitted through the network medium to *B* (Figure 5-2, Step ❸). The message should then traverse the 5 layers of the communication stack (Figure 5-2, Step ❹). Each crossed layer shall extract its header and passes the message payload to the next layer and so on, all the way up to the topmost layer. Each added header contains information dedicated to the crossed layer and normally enables a direct layer-to-layer communication between the two stacks that are respectively hosted on *A* and *B*. However, although the communication stacks have a similar design, interoperability is not supported as the stacks of *A* and *B* are bound to a specific data format.

In the RMI stack, the serialization layer offers functions to encode/decode application data in binary format according to the Java Object Serialization Stream Protocol[13] (JOSSP) specification. In the Web services stack, the same layer encodes/decodes data in the XML format according to the SOAP specification. Thus, regarding the serialization layer, RPC-based communication protocols do not differ in terms of functionalities but in the way their communication stack represents/transforms data. Similarly, for the invocation layer, applications based on RMI send messages across the network in a binary format following the JRMP specification whereas Web services use HTTP specifications. Regardless of the RPC-based communication protocol, the invocation layer offers always the same functions but differs, as previously, in the way messages are sent across the network

A way to achieve communication protocol interoperability is to offer per-layer interoperability among heterogeneous communication stacks. For instance, we should enable the invocation layer from RMI and Web services to interoperate. Although these layers use different specifications to marshall/unmarshall network messages, this challenge can be addressed because these layers provide identical functions. The same applies to the serialization layer. Obviously, if the stack related to one communication protocol is enriched with new features through the adjunction of a new layer, interoperability may be compromised. However, our aim is not to modify existing communication protocols by enriching them with functionalities that they do not implement even if others do. Particularly, according to our fourth requirement, we can not add new features if this implies changing existing applications. Hence, we enable interactions among different middleware only if there exist enough similarities in their corresponding protocol stack. In other terms, the quality of the interoperability among different middleware depends on the degree of their similarities. This is measurable in terms of the number of similar functions shared among the different communication stacks, independently of the heterogeneity of the message/data formats that is efficiently overcome through the use of event-based parsing techniques, as described in the next section.

5.2.2 Event-based interoperability

Following the design of the Amigo interoperability system dedicated to service discovery protocols, which was introduced in Deliverable D3.1b, interoperability for one layer of the protocol stack is the result of the composition of a *protocol parser* with a *protocol composer*. Specifically, the parser generates semantic events according to input messages and the composer does the inverse process, each for a specific protocol. Obviously, cooperation between a parser and composer is achievable because the parsed and produced protocols share similar functions, which are abstracted as events. An *interoperability process* is a translation process, resulting from the composition of a parser and a composer. Thus, for each layer, we have an interoperability process based on a specialized set of events.

According to the RPC communication stack, at least 5 interoperability processes are required to enable interoperability between two middleware based on different communication protocols. However, these processes cannot always be known in advance (e.g., RMI stack using JRMP or HTTP for the invocation layer). In this particular case, NEMESYS must dynamically discover the structure of the remote protocol stack to select the appropriate parser/composer in order to create and chain the interoperability processes. This challenge is naturally overcome through the structure of the network-layer message, as illustrated in Figure 5-2.

Every network message embeds the headers corresponding to the layers previously crossed. The set of headers is therefore a signature that reveals the composition of the protocol stack. Furthermore, by definition, a header always contains a magic number and/or a field to specify the current protocol used and/or the protocol expected in the next upper layer. Hence, this

[13] http://java.sun.com/j2se/1.5.0/docs/guide/serialization/

property enables chaining progressively the adequate parsers belonging to the different layers to generate a stream of events that semantically represents the RPC call. Similarly, the events are progressively forwarded to a chain of composers for generating a RPC call specification matching the service's protocol stack.

The chaining of interoperability processes is depicted in Figure 5-3. In Step ❶, the RPC call from device *A* is first parsed by the *network parser*. The parser decomposes the message into two distinct parts: the *header* and the *payload*. The former is transformed into an event stream that is forwarded to the *network composer* and the payload is passed to the *transport parser*, which is the next parser in the chain. Recursively, the *transport parser* extracts from the received payload a new header translated into events that are sent to the *transport composer* and a new payload that is directed to the *invocation parser* and so on, all the way down to the *application parser* that finally translates the data of the RPC call into an event stream. Events from each parser are sequentially forwarded to composers (Figure 5-3, Step ❷). However, composers are not able to generate a message until the last parser of the chain has parsed the last payload. In fact, the composer from the bottom level generates the payload that is required for the composer of the level immediately above and so on, all the way up to the network level (Figure 5-3, Step ❸). The resulting message is finally compliant to the protocol stack of device *B* (Figure 5-3, Step ❹). A similar process applies to the RPC reply from *B* to *A*. Therefore, to provide bidirectional communication between two different communication protocol stacks, at each protocol layer corresponds a *protocol unit*, which embeds the protocol parser and composer for the specific protocol layer as depicted in Figure 5-4. Further details about protocol units are introduced in Deliverable 3.1b in the context of service discovery.

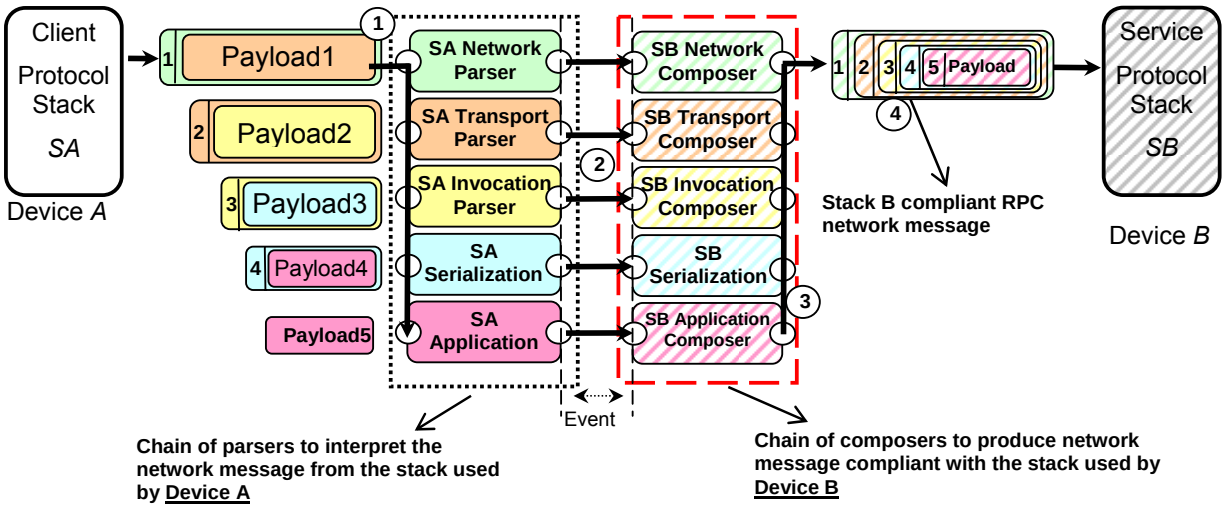


Figure 5-3: Event-based interoperability

In a way similar to Horus [RBM96], [RBFHK95], Ensemble [HR97] and Coyote [BHS98], protocol units are independent protocol modules or blocks that are stacked on top of each other to constitute a vertical protocol stack. However, we further introduce a dynamic composition of protocol stacks that is both vertical and horizontal. Vertical stack composition (i.e., vertical unit chaining) enables translating an RPC call to a stream of semantic events whereas the horizontal stack composition (i.e., horizontal unit chaining) translates the stream of semantic events to another protocol (See Figure 5-4). Also, note that, contrary to the above systems that provide reconfiguration of protocol stacks [RBFHK95, BHSC98, HR97], with NEMESYS, applications and services are not aware of the reconfiguration of protocol compositions and are therefore not bound to the NEMESYS system. The latter acts at the

network layer on top of the operating system and below legacy middleware (see Figure 5-5). Further, NEMESYS needs only to be deployed on one of the nodes involved in the communication, whether the client, the service host, or even a gateway.

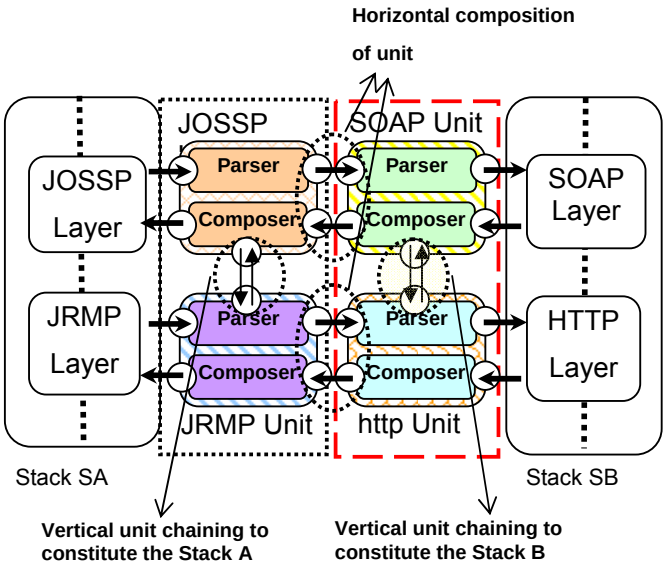


Figure 5-4: Vertical and horizontal stack composition to provide interoperability

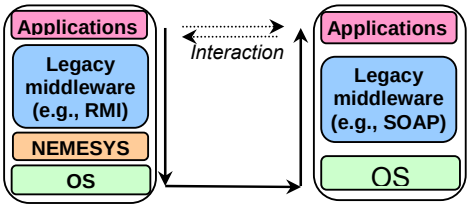


Figure 5-5: Localisation of the NEMESYS system

5.2.3 NEMESYS instances

In practice, there are not as many units to compose as protocol stack layers. In general, protocol stacks share a number of identical layers, reducing so the number of units involved in the interoperability processes. For instance, as illustrated in Figure 5-2, a majority of RPC protocols are based on TCP/IP, simplifying the interoperability system, which works only from Layers 3 to 5 (i.e., invocation to application layers). The TCP/IP drivers of the operating system act as units dedicated to the two first layers. However, if the latter are heterogeneous, our system also enables dynamically, through adequate units, interoperability among different networks.

NEMESYS is built around the concept of vertical and horizontal, dynamic unit chaining. However, dynamic chaining is not without cost in terms of resource consumption and is not always required. To improve efficiency, the vertical composition of protocol units, for each supported protocol stack, can be achieved statically. In this case, the service discovery process enables NEMESYS to select the adequate vertical stack which is statically composed. The stack corresponds to the middleware associated (if any) with the specific service discovery protocol that is run (e.g., JINI implies RMI middleware). Interoperability among heterogeneous stacks is still dynamic as the horizontal composition of protocol units is. Specifically, the NEMESYS interoperability system is defined as a set of protocol units that can be either statically or dynamically composed. As illustrated in Figure 5-6, specification of a NEMESYS instance defines the supported units (for invocation and serialization layers) and the vertical protocol stacks that are statically composed. However, at run-time (see Figure 5-7), NEMESYS may still dynamically create new vertical stacks, or reconfigure the existing stacks, which were statically composed, by adding, removing or changing one protocol unit by another, according to the context. Protocol units are not necessarily specific to one communication protocol and may be stacked in various ways. For instance, the vertical stack, named RMI_2 in Figure 5-6, that handles mobile code of RMI-based clients/services, depends on the HTTP unit, which is also used by the SOAP stack.

System specification at design-time

```
System NEMESYS= {
        Component Unit JRMP;
        Component Unit JOSSP;
        Component Unit MOBILECODE ;
        Component Unit HTTP;
        Component Unit SOAP;
        Component Unit IIOP;
        Component Unit CDR;
Unit Chain SOAP     = {HTTP, SOAP}
Unit Chain RMI_1    = {JRMP, JOSSP}
Unit Chain RMI_2    ={HTTP,MOBILECODE}
                    …}
```

Figure 5-6: Specification of a NEMESYS instance

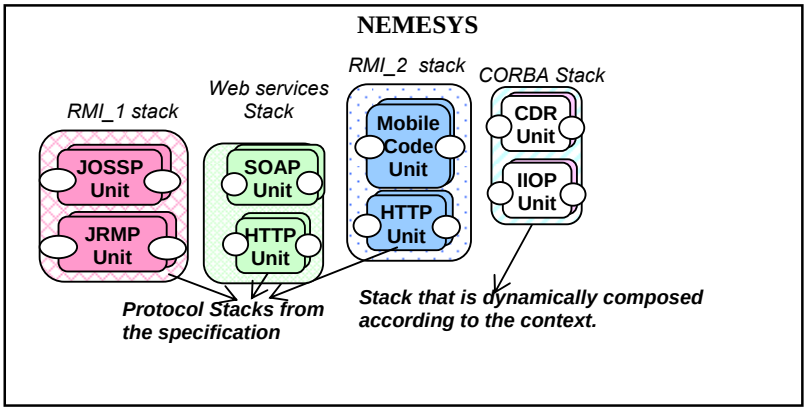


Figure 5-7: NEMESYS instances

In general, NEMESYS instances evolve across time due to the communication protocols used by both the hosted applications and the available networked services. Accordingly, protocol units are reconfigured in order to provide interoperability between clients and services

5.3 Interoperable middleware

From the reference RPC-based middleware architecture (§5.3.1), it appears that middleware for dynamic open networks must overcome the heterogeneity of Service Discovery Protocols (SDP) in addition to the one of communication protocols. In the service discovery domain, the Amigo solution to SDI (see Deliverable D3.1b and [BI05]) has proven the efficiency of event-based interoperability for service discovery protocols. Consequently, an interoperability system based on the cooperation between the Amigo SDI subsystem (referred to as INDISS for INteroperable DIscovery System for networked Services) and NEMESYS can be coupled with any middleware architecture to provide: (i) full middleware interoperability (§5.3.2), (ii) and a universal service registry (§5.3.3).

5.3.1 RPC-based middleware architecture

In order to interact with services in open networked home environments, clients must first find remote services using some Service Discovery Protocols (SDPs). Then, they rely on specific information to actually interact with discovered services. Service registries are logical centralization points that allow clients to lookup the information needed to interact with services. Each middleware depends on a dedicated registry. For instance, Web Services, which are based on the *SOAP* protocol, use *UDDI*[14] whereas the *RMI* and *CORBA* middleware use repositories respectively called *rmiregistry* and *corba naming services*.

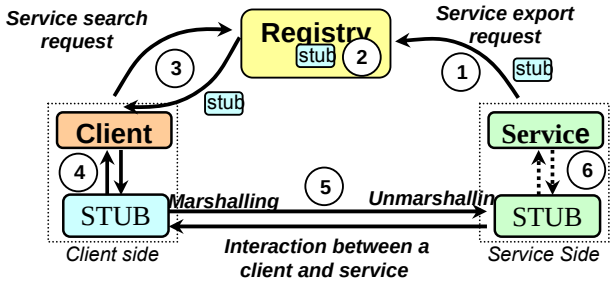


Figure 5-8: RPC-based middleware architecture

Remote services must be first published/exported to a registry to be accessed by clients (Figure 5-8, Step ❶). Through the export process, services advertise both their interface and their unique reference. The former is a set of methods describing the service's communication contract whereas the latter provides a mean to locate the service's instance. These data enable producing a stub that acts as a proxy for the remote service. Clients then use the stub as a handle to make method calls to the remote service. The way stubs are produced and obtained by clients may differ from one middleware to another. Stubs can be obtained statically or dynamically. In the former case, stubs are generated at design-time, avoiding clients to get it at run-time. In the latter case, stubs are transparently created by the export process and registered to the registry (Figure 5-8, Step ❷). The repository's location is either known in advance by the client or dynamically discovered using some SDPs.

[14] http://www.uddi.org/specification.html.

Once the client gets the stub, either dynamically (Figure 5-8, Step ❸) or statically, it can interact with the desired service. To invoke a method on the remote service, the client makes a local call on the corresponding stub (Figure 5-8, Step ❹). The latter first marshals the call into a request message according to the invocation protocol used by the middleware (e.g., IIOP for CORBA, JRMP for RMI) and then sends the message to the remote service (Figure 5-8, Step ❺). Hence, clients are not aware of the implementation specifics of services; stubs abstract their location, programming language and invocation protocol. Finally, on the service side, the incoming request message is unmarshalled by the service stub into a local call (Figure 5-8, Step ❻)

5.3.2 Interoperable service discovery and communication

As both INDISS and NEMESYS are event-based systems, they can easily cooperate. NEMESYS and INDISS may then be collocated on a client, service or even gateway device.

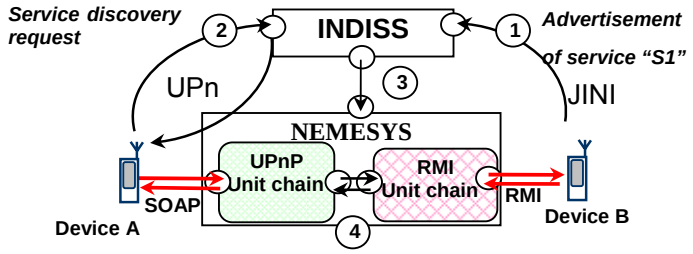


Figure 5-9: INDISS & NEMESYS cooperation

Figure 5-9 illustrates cooperation between INDISS and NEMESYS for a UPnP client to access a JINI service; only the main steps of service discovery are illustrated. From the SDPs used in the networked environment, INDISS knows the communication protocols associated with both the client and the service. For instance, when the service, named *S1*, hosted on device *B*, advertises its existence through JINI (Figure 5-9, Step ❶), INDISS identifies that the communication protocol to be used to interact with *S1* is RMI (i.e., JINI always implies RMI). Similarly, when the client sends a UPnP request to find a service (Figure 5-9, Step ❷), INDISS knows that the communication protocol used by the client is SOAP-based (i.e., UPnP always implies SOAP). Assuming *S1* matches the service requested by *A*, INDISS replies to the client's request, by indicating the IP address of NEMESYS as being the one of the requested service. Then, INDISS dispatches its knowledge about the communication protocols that are used to NEMESYS (Figure 5-9, Step ❸). The latter learns that a UPnP client, identified by the IP address of *A*, wishes to interact with a RMI-based service, identified with the IP address of *B*. Note that the UPnP stack is similar to the one of Web services (see Figure 5-7). Thus, NEMESYS configures protocol units in order to horizontally compose a UPnP stack with a RMI stack to enable communication interoperability from the clients to the service (Figure 5-9, Step ❹). UPnP messages from *A* are ready to be translated to RMI messages towards *B* and vice versa.

There exist SDPs for which INDISS is unable to predict the communication protocol used by clients and/or services. Indeed, some SDPs, such as SLP, are not coupled with a specific middleware architecture and are thus not dedicated to a particular communication protocol. In this particular case, NEMESYS is able to detect the structure of the communication protocol by assembling dynamically adequate protocol units, according to the concepts presented in §5.2.2.

5.3.3 NEMESYS universal repository

As presented in §5.3.1, a majority of RPC-based middleware depends on service registries. Registries are bootstrap naming services dedicated to one RPC protocol. Services are able to export their interface only if a registry, compliant with their RPC-based middleware, can be found. The same applies to clients that may need to get a reference to a remote service: if no compatible registries are found, clients are unable to get a corresponding stub compliant with their middleware. In a dynamic open network, it cannot be considered that: (i) there exist as many registries as existing RPC-based middleware, and/or (ii) all clients have already the stub corresponding to the remote services they access. To cope with the above issue, NEMESYS provides to clients and services common functions delivered by standard registries (i.e., export, search and get functions). Accordingly, the requirement to have one repository for each communication protocol is overcome: NEMESYS acts as a universal registry and is able to provide standard registry functions, irrespectively of the communication protocol, through its core system.

In Figure 5-10, INDISS enables clients and services to discover the universal registry service provided by NEMESYS. As illustrated in Figure 5-9, the discovery of the registry enables NEMESYS to know the communication protocol used by clients and services. Once the registry is found, services are able to export to NEMESYS both their interface and their unique reference (Figure 5-10, Step ❶). The interface and the unique reference are transformed to a set of semantic events that are saved into NEMESYS. Then, clients can send a search request to NEMESYS, which acts as a universal registry, to find a service (Figure 5-10, Step ❷). Thereafter, NEMESYS takes in charge the interoperability between the client and service middleware (Figure 5-10, Steps ❸ & ❹), as introduced in Section 5.2.

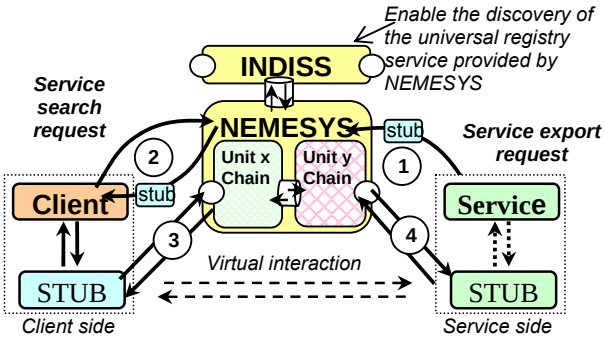


Figure 5-10: Universal registry

5.4 Prototype implementation and performance

We have implemented a first proof-of-concept prototype of the NEMESYS interoperability system. The prototype includes units for the RMI and Web services RPC communication protocols (See Figure 5-2). Currently, RMI and Web services vertical stacks are statically composed. However, the horizontal composition of protocol units is dynamic and context-dependent. Furthermore, we are enriching our prototype to support both vertical and horizontal dynamic composition of protocol units and more communication protocols.

The following discusses key features of the prototype. We first outline the capability of the system to provide interoperability between RMI and Web services middleware, without requiring any Java Virtual Machine (JVM) (§5.4.1). We then evaluate the performance of

NEMESYS by comparing the latency needed for a successful method call with and without NEMESYS (§5.4.2)

5.4.1 Prototype implementation

The NEMESYS prototype is implemented in ANSI-C. The C programming language has been chosen for several reasons: (i) it enables the deployment of NEMESYS without requiring any additional software (e.g., requirement of the Java virtual machine) as embedded system kernels are mainly developed in C, and (ii) it increases the execution speed, which is a key requirement. However, NEMESYS may be developed in any other programming language and/or dedicated to one specific software platform to increase further its efficiency.

NEMESYS provides 2 instances of the SUN compliant RMI stack (See RMI_1 and RMI_2, Figure 5-7) through the use of 4 units developed in C: the Java Remote Method Protocol (JRMP), Java Object Stream Protocol (JOSSP), HTTP protocol and Java Mobile Code. As given in Table 5-1, the RMI stack of NEMESYS requires at most 636 Kb against about 3Mb for the Java Micro Edition environment with the additional packages to support RMI as a client. Note that we reuse existing non optimised HTTP library. In addition, through an adequate configuration of the protocol units, NEMESYS can act not only as an RMI client but also as an RMI service and is therefore able to generate dynamically Java proxy/stub code on the fly. This behaviour is, normally, only possible on the desktop Java runtime environment whose size is of 45 Mb. NEMESYS drastically reduces the size requirements to support the full features of the RMI specification, as it needs neither a JVM nor Java class libraries at all.

NEMESYS					**SUN JVM**	
Units		**Size**	**RMI Stack 1**	**RMI Stack 2**	**JRE**	**J2ME**
Mobile Code	*Parser* *Composer*	140	-	X	-	-
JOSSP	*Parser* *Composer*	56	X	-	-	-
JRMP	*Parser* *Composer*	40	X	-	-	-
HTTP	*Parser* *Composer*	164	-	X	-	-
IO abstraction		36	X	X		
Event Manager		200	X	X		
TOTAL in Kb		**636**	**332**	**540**	**45000**	**3000**

Table 5-1: The RMI stacks of NEMESYS vs. Sun JVM

To support the Web services communication protocol, the NEMESYS prototype builds on an existing SOAP library developed in C, to implement the required SOAP and HTTP units. Unfortunately, to the best of our knowledge, there does not exist any optimised SOAP library, developed in C, dedicated to resource constrained devices in the open source community. Consequently, we reuse the CSOAP[15] library, which has the severe constraint to be memory consuming, as given in Table 5-2. GSOAP [EG02] is known to be more appropriate for saving

[15] http://csoap.sourceforge.net

resources, but it does not provide the ability to create dynamically SOAP calls at run-time. It is interesting to note that some commercial SOAP versions require only 150Kb against the 1524Kb for CSOAP. Accordingly, it is very promising for the next NEMESYS prototypes in terms of memory cost. Nevertheless, although the current NEMESYS prototype is half optimised, its size is already less than the J2ME runtime, while providing interoperability.

NEMESYS			
Units		**Size**	**Web services Stack**
SOAP Unit	*Parser* / *Composer*	1360	**X**
HTTP Unit	*Parser* / *Composer*	164	**X**
Event Manager		200	**X**
TOTAL in Kb		**1724**	**1724**

Table 5-2: The CSOAP-based Web services stack of NEMESYS

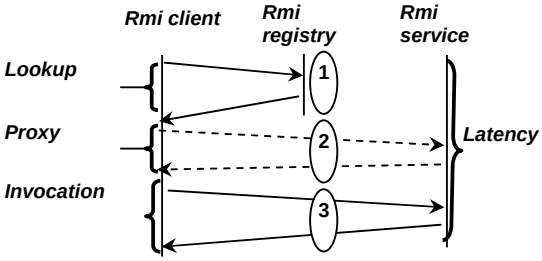
5.4.2 Experimental results

We evaluate the performance of NEMESYS by investigating the latency required for a client to get an answer to its RPC request from a remote service based on a different RPC protocol. The latency does not include the time needed for the service to export its interface (i.e., Figure 5-8, Step ❶). Although the exporting step is mandatory, it is more related to the service/registry discovery process than the interoperable interaction mechanism. Accordingly, our experiments focus on the latency of remote service invocation for which we implemented an *echo* service that echoes to the client the string given as an argument in the RPC request. We compare then the resulting latency with the one of a native RPC between a client and service based on an identical RPC protocol.

Although our solution is dedicated to various devices, including resource constrained ones, all tests are performed on a workstation equipped with 256Mbytes RAM on Intel IV processor rated at 1.8GHz as our focus is on assessing performance against native cases. Hence, the operating system is Linux Redhat Fedora Core 2. NEMESYS is compiled with the *gcc* compiler and the *glibc library* version 3.2.2. The Web services client and service are based either on the CSOAP library or Java Apache Axis[16], whereas the RMI client and service are based on JDK 1.4.2 from SUN. The given measurements are in ms and are the median of 15 successful tests to avoid a mean skewed by a single high or low value. Moreover, all the tests are run on a single host to avoid the network delays, as we want to measure the NEMESYS performance. Indeed, NEMESYS provides interoperability without affecting the existing protocols and therefore does not increase the network bandwidth consumption.

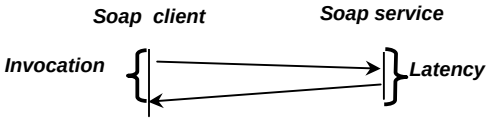
Figure 5-11 depicts a RMI request/response between a RMI client and service. If the client has already the proxy byte-code of its desired remote service, the overall latency (Figure 5-11, Steps ❶ & ❸), including both the RMI invocation and the RMI lookup request (i.e., to get the stub of the remote service from a regular RMI registry), is 201ms. However, if we consider exclusively the RMI invocation from the client perspective, the request/response latency takes only 1 ms against 8.08 ms or 20 ms for a similar SOAP interaction between a Web services client and service developed respectively in C or Java (See Figure 5-12).

[16] http://ws.apache.org/axis/.



	Elapsed time (ms)
❶ RMI lookup	200
❸ RMI request/response	1
Total with proxy	201

Figure 5-11: Native RMI RPC with and without mobile code

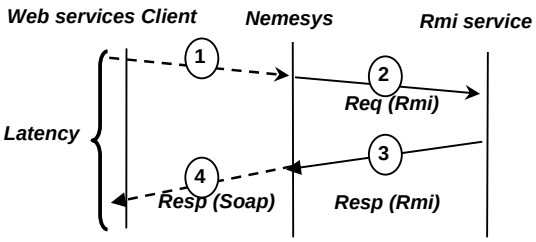


	Latency (ms)	
	CSOAP	Java AXIS
Total	**8**	**20**

Figure 5-12: Native SOAP invocation in C and Java

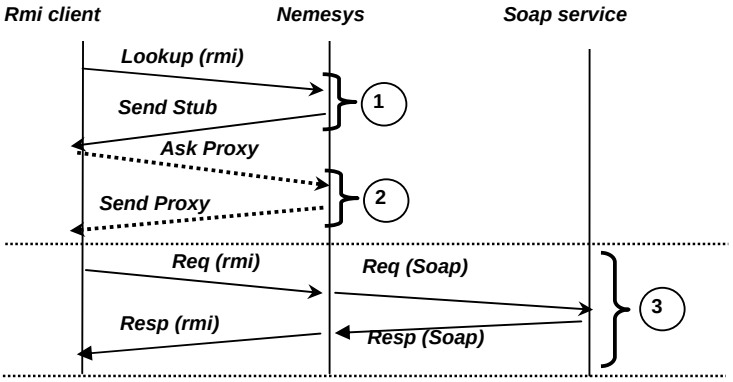
Since the RMI RPC is binary oriented, RMI invocations are obviously faster than SOAP ones. Furthermore, the latency difference between the C and Java SOAP native call hints at the impact of the C programming language on performance.

Consider now the case where the client and service are based on heterogeneous RPC protocols and rely on NEMESYS as a transparent intermediary that achieves interoperability. When the client is SOAP and the remote service is RMI-based, the overall latency of the SOAP interaction, from the client perspective, is of about 9 ms (see Figure 5-13). Comparing to the C-based SOAP native call, the latency of 1 ms overhead corresponds to the latency of a Java-based RMI interaction. In other terms, the interoperability between a SOAP client and a RMI service takes as much time as is needed for exactly both one C-based SOAP interaction and one Java-based RMI interaction. Comparing now the 9 ms with the 20 ms required for Java-based SOAP interaction, NEMESYS clearly performs better. However, if we compare solely with the RMI native case, NEMESYS performs poorly but this is inherent to the SOAP protocol.



	Latency (ms)
❶SOAP Parser	5
❷RMI Composer	0.2
❸RMI Parser	0.2
❹SOAP Composer	3
Total	9

Figure 5-13: Interoperable invocation between a Web service client and a RMI service with NEMESYS



	Latency (ms)
❶ STUB Generation	0.30
❷ Mobile Code Generation	0.85
❸ Invocation	9
Total with proxy	9.30
Total without proxy	10.15

Figure 5-14: Interoperable invocation between a RMI client and a Web service with NEMSYS

Consider next that the client is RMI-based and the service is SOAP-based, NEMESYS acts, from the client side, as both a compliant RMI registry and a RMI remote service (See Figure

5-14). The mandatory lookup request from the client to get the stub of the service takes about 0.30 ms when NEMESYS acts as a RMI registry, whereas it takes 200ms with a standard java-based registry. In the case where the client does not have found in its JVM the proxy byte-code corresponding to the received stub, the latency increases of 0.85 ms. This overhead corresponds to the cost for the client to get from NEMESYS the proxy byte-code, which is dynamically generated from the interface exported by the Web services remote service (Figure 5-14, Step ❷). Moreover, excluding Steps ❶&❷, the latency of the client RMI invocation (Figure 5-14, Step ❸) is almost equal to the similar C-based SOAP invocation of the previous scenario. In fact, once clients have all the necessary information to perform their RPC call (i.e., endpoints, stubs, proxy byte-code), the cost of the interoperability processes between Web services and RMI entities is finally independent of the nature of the client/service (i.e., either RMI or SOAP based) and stays nearly constant: about 9 ms.

Summarising, for sending a lookup, the latency increases of 0.30 ms whereas for getting the proxy byte-code, the latency increases of 0.85 ms. Therefore the overall latency is, in the best case, of 9ms, and in the worse case, of 10.15 ms (See Figure 5-14) . It is clear that the latency required for an interoperable interaction between RMI and SOAP entities can not be smaller than the sum of the latency required for both a native RMI call and a native C-based SOAP call. Hence, the overhead of NEMESYS is negligible

5.5 Concluding remarks

NEMESYS, the Amigo solution to Service Interaction Interoperability (SII) in the open networked home environment, enables any application, in the networked home, to interoperate with any networked service, irrespectively of their underlying middleware. Our solution is specifically designed for open networked environments, possibly wireless, which require both minimizing resource consumption, and introducing an interoperability system that may be deployed easily on any platform. Building upon our solution to Service Discovery Interoperability (SDI), called INDISS, presented in Deliverable D3.1b, NEMESYS is an event-based system that provides dynamic interoperability through the dynamic composition of protocol units that achieve on-line protocol translation. The latter are vertically composed to constitute vertical protocol stacks, whereas they are horizontally composed to provide interoperability among heterogeneous protocol stacks by performing per-layer protocol translation. Applications and services are not aware of such a configuration: they are not bound to the NEMESYS system, which transparently acts at the network layer on top of the operating system and below legacy middleware. Additionally, coupled with INDISS, NEMESYS provides a full interoperability system that provides both service discovery and communication interoperability. As demonstrated by the first NEMESYS prototype, experiments results are encouraging, as the overhead of using NEMESYS is negligible.

Currently, NEMESYS is focused on RPC-based middleware. We are both investigating solutions to overcome such a limitation and enriching the prototype to support more communication protocols. The prototype will further be released as Open Source Software as part of the overall Amigo middleware.

6 Domotic infrastructure

6.1 Overview

The Amigo Domotic Infrastructure aims at presenting heterogeneous physical hardware devices as unified software services using standard services 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, as detailed in D3.1b [Amigo-D3.1b], 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:

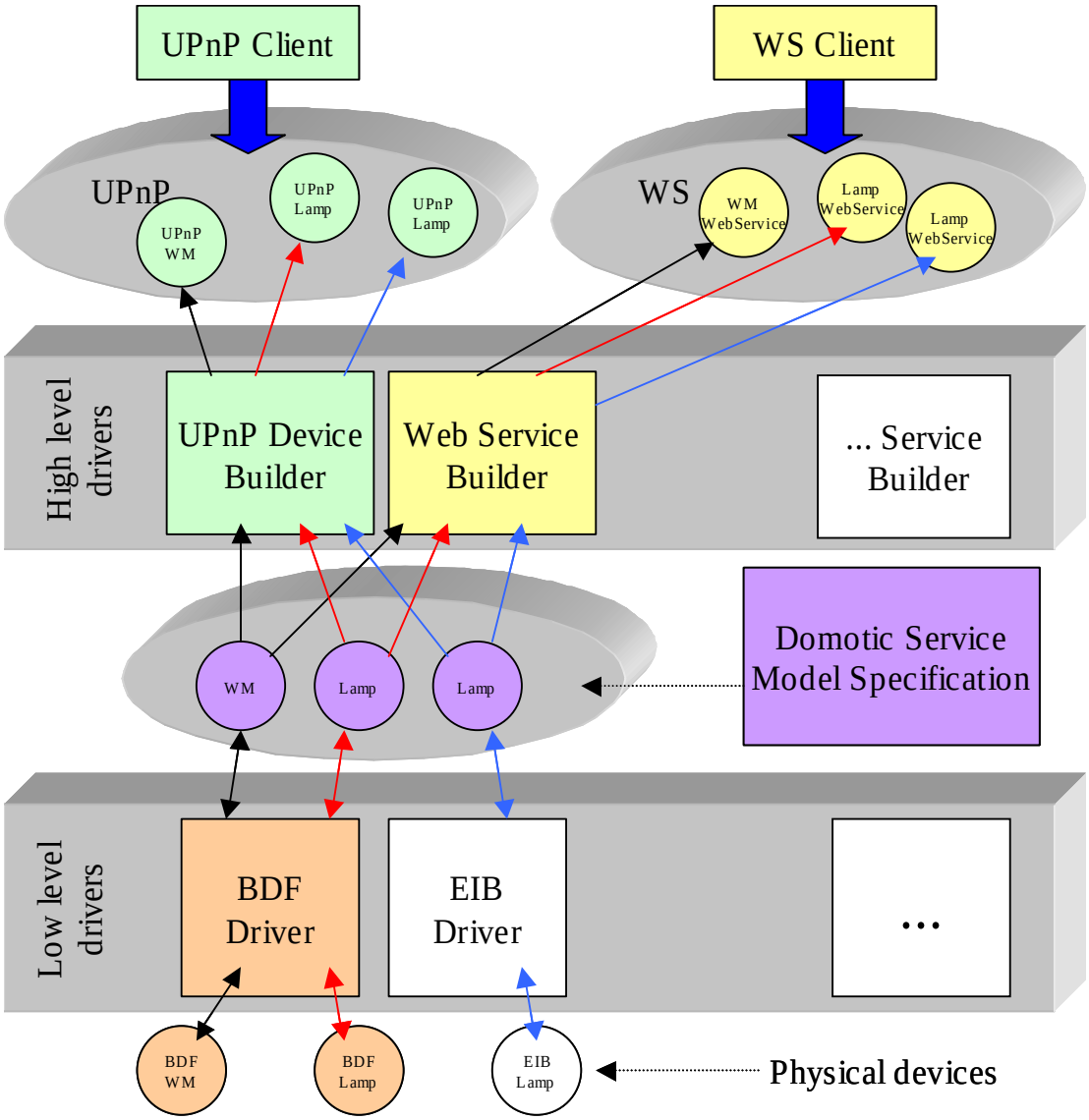


Figure 6-1: Domotic Infrastructure

The architecture is based on extracting the required information about the physical devices by means of drivers to the base technologies (BDF, EIB…); modeling 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.

The following components will be developed:

- Domotic Service Model Specification
- BDF Driver (low-level Driver)
- UPnP Device Builder (high-level Driver)

The Web Service Builder and WS Client in Figure 6-1 serve as an example to clarify the proposed architecture, but are not intended to be developed now.

6.2 Domotic Service Model

Provider

IKERLAN

Introduction

In order to integrate heterogeneous domotic devices, an abstract description of the available services, not attached to 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.

Development status

Development was started in Q1 of 2006.

Intended audience

Low-level driver developers must translate and instantiate services from the corresponding legacy technology to this generic description. High-level driver developers use this reference as a starting point for the high-level proxy generation process.

License

The reference will be released under a LGPL license.

Language

Java (OSGi), .NET languages (C#, VBNET)

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

This component is not an executable, but a library.

Platform

Java Platform for Java developers

.NET Platform for .NET developers

Tools

Java development tools

.NET development tools

Files

Source code files are currently available only on [Amigo-OSS-SCM] under the mdwcore/domotics structure.

Documents

Documentation (only developer's guide, because it's not a user-oriented component) will be available on [Amigo-OSS-Pub]:

- Developer's guide: design principles and UML diagrams

Tasks

The final release is scheduled for M30.

There will be some intermediate releases.

Bugs

None yet, but will be initially reported on [Amigo-OSS-SCM] under the mdwcore/domotics structure.

Patches

None yet, but will be initially reported on [Amigo-OSS-SCM] under the mdwcore/domotics structure.

6.3 BDF Driver (low-level driver)

Provider

IKERLAN

Introduction

A low-level driver is a base technology-dependent driver (in this case, BDF) that generates and instantiates proxies for the devices that it supports (a BDF washing-machine, BDF oven, BDF plug…) in a generic (base technology-independent) way.

Development status

Development was started in Q1 of 2006.

Intended audience

Low-level driver developers. This component will be a sample implementation of a low-level driver. New low-level drivers for other domotic base technologies (EIB, EHS…) can be developed following the principles described by this module.

License

The software developed will be released under a LGPL license.

The base technology employed (BDF native driver) is under a proprietary license.

Language

Java (OSGi)

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

Hardware: BDF domotic devices, BDF bridge to RS232

Software: JVM, OSGi implementation, BDF native driver (OSGi bundles), OSGi Domotic Service Model component

Platform

Java 1.4.2

OSGi

Tools

Java development tools

Files

Source code files are currently available only on [Amigo-OSS-SCM] under the mdwcore/domotics structure.

Documents

Documentation (only developer's guide, because it's not a user-oriented component) will be available on [Amigo-OSS-Pub]:

- Developer's guide: describing the design principles and documentation.

Tasks

The final release is scheduled for M30.

There will be some intermediate releases.

Bugs

None yet, but will be initially reported on [Amigo-OSS-SCM] under the mdwcore/domotics structure.

Patches

None yet, but will be initially reported on [Amigo-OSS-SCM] under the mdwcore/domotics structure.

6.4 UPnP Device Builder (high-level driver)

Provider

IKERLAN

Introduction

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

Development status

Development was started in Q1 of 2006.

Intended audience

High-level driver developers. This component will be a sample implementation of a high-level driver. New high-level drivers (WS, SLP, Jini…) can be developed following the principles described by this module.

Domotic service clients (UPnP clients).

License

The module will be released under a LGPL license.

Language

Java (OSGi)

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

Software: JVM, OSGi implementation, OSGi Domotic Service Model component

Platform

Java 1.4.2

OSGi

Tools

Java development tools

Files

Source code files are currently available only on [Amigo-OSS-SCM] under the mdwcore/domotics structure.

Documents

Documentation will be available on [Amigo-OSS-Pub]:

- User's guide: UPnP device and service description XML files.
- Developer's guide: describing the design principles and documentation.

Tasks

The final release is scheduled for M30.

There will be some intermediate releases.

Bugs

None yet, but will be initially reported on [Amigo-OSS-SCM] under the mdwcore/domotics structure.

Patches

None yet, but will be initially reported on [Amigo-OSS-SCM] under the mdwcore/domotics structure.

7 Security & Privacy

Introduction

The Amigo security service is the central instance to handle authentication and authorization in the Amigo home. It forms the primary (and typically only) interface of the Amigo security framework with the Amigo user, where user and device access to the home can be controlled based on a role-based authorization scheme.

This component is currently in a very early, proof-of-concept stage. Most aspects of the user interface are expected to be revised in coming releases of this component, which will be tied more strongly into other services provided by the Amigo middleware. Thus, the information in this chapter is necessarily preliminary and incomplete.

7.1 Security Framework

Provider

Microsoft, IMS

Introduction

This component provides access to the authentication and authorization service of Amigo (see Security Services, Section 7.2). It encapsulates the communication and cryptographic primitives that are used for device/user registration, authentication, and authorization with the centralized Amigo security service, which is released as a separate component.

The Amigo security system is based on a centralized Security service, which may be replicated to achieve higher system reliability. The employed protocol is a simplified, web-service version of Kerberos: shared secrets are established during registration and are subsequently used for mutual authentication. Authorization by the security service is granted following a role-base authorization scheme, and transmitted securely using encrypted tickets.

The current framework provides convenient abstractions of this underlying protocol, and enables programmers to participate in the security scheme without having to understand the details of the security mechanism. It includes a discovery mechanism that allows automatic fail-over in case of the unavailability of a particular instance of the security service, based on WS-Discovery.

Development status

The first prototype version was distributed to the Amigo partners in M18. An external release to the public is planned in M24. The software will be available from a download page off of the Microsoft EMIC web site at www.microsoft.com/EMIC (also referenced on [Amigo-OSS-Pub]).

The Java implementation is still under development; a first prototype version will be available in M24 on [Amigo-OSS-Pub].

Intended audience

Service and application developers that need to control access to their service/application.

License

.NET version: See Appendix A.

Java version: The Java libraries will be made available under the LGPL license terms.

Language

C# / Java

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

The security framework will support/employ:

- Hardware: PC/Laptop/PDA/Smartphone
- OS: Windows XP / Windows Server 2003 / Pocket PC 2003 / Smartphone 2003 / Linux
- Software: .Net for Windows / .NetCF for Windows / OSGi / JRE 1.5

Platform

Microsoft .Net 2.0 / Microsoft .NetCF 2.0 or JVM

Tools

Generic .Net tools, Visual Studio 2005

Eclipse

Documents

The developer's and user's guide are available on [Amigo-OSS-Pub].

Tasks

For .NET, there will be an intermediate release in M24 and a full release in M30.

The first Java version will be available in M24.

Bugs

None so far

Patches

None so far

7.2 Security Service

Provider

Microsoft

Introduction

The Amigo security system is based on a centralized Security service, which may be replicated to achieve higher system reliability. The employed protocol is a simplified, web-service version of Kerberos: shared secrets are established during registration and are subsequently used for mutual authentication. Authorization by the security service is granted following a role-base authorization scheme, and transmitted securely using encrypted tickets.

The role-based authorization scheme works by assigning each registered device/user/service to a specific class, like domotic, admin, mobile, etc. Access to a service of a specific class is granted based on an access matrix, which captures which service class may be used by which device and/or user class.

Development status

The first prototype version was distributed to the Amigo partners in M18 (as a minimal implementation of a security service). An external release to the public is planned in M24. The software will be available from a download page off of the Microsoft EMIC web site at www.microsoft.com/EMIC (also referenced on [Amigo-OSS-Pub]).

Intended audience

Service developers as well as application developers that need to control access to their service/application.

License

See Appendix A.

Language

C#

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

Security services will support/employ:

- Hardware: PC / Laptop
- OS: Windows XP / Windows Server 2003
- Software: .Net for Windows

Platform

Microsoft .Net 2.0

Tools

Generic .Net tools, Visual Studio 2005

Documents

The developer's and user's guide are available on [Amigo-OSS-Pub].

Tasks

Intermediate release in M24 and full release in M30

Bugs

None so far

Patches

One so far

8 Content Delivery

Provider

Microsoft and TID

Introduction

The Content Delivery service will provide available content in the Amigo home to Amigo services and applications. This is achieved by aggregating content meta data (description) from UPnP Digital Media Servers (like Windows Media Connect, etc.). The actual content will not be copied for performance reasons. Aggregated content will be again visible through a standard UPnP Digital Media Server for seamless integration in standard UPnP environments.

Moreover, Content Delivery has the ability to provide content in a format which suits the renderer's capabilities in the best possible way. For this, copying content and adapting it becomes necessary. How adaptation is applied to some content is decided by Content Selection with the help of device capabilities gathered from UPnP device descriptions and other sources like CCPP profiles. Adaptation itself is handled by a subcomponent (see Section 8.1). This is an extension to the UPnP AV Architecture is therefore not useable by standard UPnP devices.

Content Delivery will be able to render content to UPnP Digital Media Renderers (DMR). When content is subject to adaptation some delay might be expected, otherwise rendering will start directly. For non-UPnP DMRs like Windows Media Player etc. some application needs to take care of transferring content to the rendering device via http-get. The same is applicable for offline consumption.

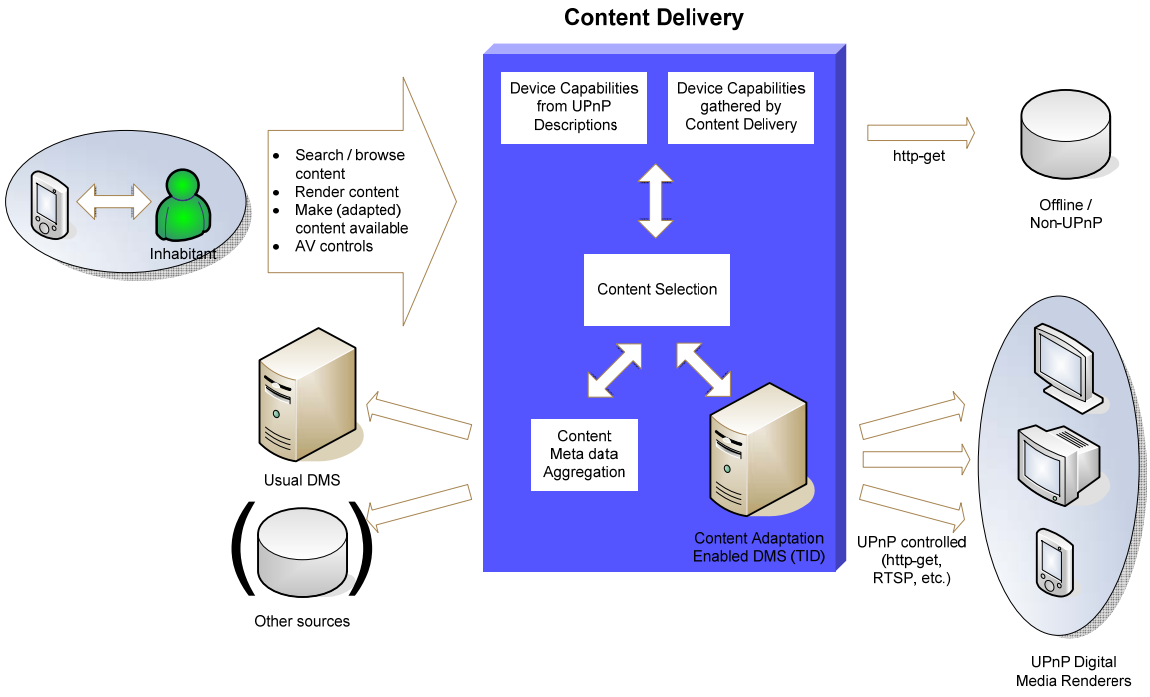


Figure 8-1: Content Delivery

Development status

There will be an initial release in M24. Final release will be in M30. The software will be available from a download page off of the Microsoft EMIC web site at www.microsoft.com/EMIC

Intended audience

Service and application developers that need some (entertaining) content to be rendered or delivered to some device in the Amigo home.

License

See Appendix A

Language

C#

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

The content distribution service will support/employ:

- Hardware: PC/Laptop
- OS: Windows XP / Windows Server 2003
- Software: .Net for Windows

Platform

Microsoft .Net 2.0

Tools

Generic .Net tools, Visual Studio 2005

Documents

The developer's and user's guide are available on [Amigo-OSS-Pub].

Tasks

Development has started in Q1 2006. Subsequent releases will be available as listed below:

- M20: early prototype, basic content delivery, no adaptation
- M23: first prototype, basic adaptation like mpeg2-to-mpeg4
- M30: final prototype.

Bugs

None so far

Patches

None so far

8.1 Subcomponent: Content Adaptation

Provider

TID

Introduction

The broad range of consumer devices requires an adaptation engine to deliver the content with compatible properties. This engine needs to be versatile enough not only to provide needed transformations to render content in a wide variety of devices, but also to cover the evolutions of devices and media formats. Many conversion tools provide a wide variety of content transformations but most of them do not implement the whole transformation matrix consisting of source/target format pairs. Usually, powerful tools are able to convert one or two input formats into a wide variety of output formats. In a limited environment such as the home where powerful transcoding software with a wide and complete transformation matrix will not be available, an adaptation system that is able to deal with a limited set of conversion tools to provide the maximum number of transformations is desirable.

By expressing, together with the complex ones, the simple or *atomic* transformations that a conversion tool is capable of performing in the tool's description, an adaptation system may take advantage of these descriptions to multiply the number of possible transformations. The concept behind this statement is that none of the single tools of the set of present ones might be able to perform certain complex transformation, even though it would be possible to perform it by composing simple (atomic) transformation capabilities offered by the same set. In the first case the target format would be considered unreachable by the adaptation system while in the second it would be considered reachable, and through some kind of composition process, obtained.

Semantic specification of these capabilities enables managing content transformation concepts independently of the underlying implementation or grounding; in other words, allows a common description of these capabilities at a higher, technology-independent, level.

As a result a Content Adaptation Enabled DMS (CADMS) will be provided as subcomponent of the Content Delivery subsystem. It publishes and offers adaptation services for content resources stored in it, which will mainly serve the Content Delivery components to convey appropriate content resources to selected renderers. Furthermore, the semantic framework for integrating plugins is additionally provided in the CADMS, based on the semantic specification of content conversion capabilities and a plugin registration mechanism.

Development status

Development was started in Q1 2006. The final prototype will be provided at the end of M30.

Intended audience

This is a subcomponent and as such only interesting to the Content Delivery component or developers/integrators of transcoding plugins for Amigo.

License

Content adaptation will be made available under the LGPL license terms.

Language

Java

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

Content adaptation will employ/support:

- Hardware: PC/Laptop
- OS: Any Java-enabled OS
- Software: Partial UPnP Cyberlink DMS implementation, Jena2

Platform

Any system capable of running JRE 1.5

Tools

None so far

Documents

The developer's and user's guide are available on [Amigo-OSS-Pub].

Tasks

Development has started in Q1 2006. Subsequent releases will be available as listed below:

- M23: first prototype, basic adaptation like mpeg2-to-mpeg4
- M30: final prototype.

Bugs

None so far

Patches

None so far

9 Data Store

Provider

Microsoft

Introduction

This component offers a generic storage capability to other components and applications inside an Amigo System. There is no restriction on the kind of content that can be stored and each component or application can open and control access to a sub-store inside the Data Store. The data store supports notifications on changes in a sub-store. Data is automatically backed up and restored when necessary.

The data store uses a concept of individual data stores (SSDS – Service Specific Data Store) that are created on behalf of an owner. The owner of a SSDS specifies:

- The structure of each data element in an SSDS
- The user group and their access rights to an SSDS
- The events that are generated when elements in an SSDS are modified
- Whether versioning needs to be applied to modifications (history of changes to allow retrieval of historical data)

Operations on an SSDS include addition, deletion, modification and querying of data elements.

The data store is a centralized solution, performing automatic backup and restoration functions when needed to allow a maintenance free operation.

Development status

There will be an initial release in M24. Final release will be in M30.

Intended audience

Service and application developers that need a reliable storage for their data.

License

See Appendix A.

Language

C#

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

Data store will support/employ:

- Hardware: PC/Laptop
- OS: Windows XP / Windows Server 2003
- Software: .Net for Windows / Microsoft SQL Server 2005 Express Edition

Platform

Microsoft .Net 2.0 runtime (http://download.microsoft.com)

Tools

Generic .Net tools, Visual Studio 2005

Documents

The developer's and user's guide are available on [Amigo-OSS-Pub].

Tasks

There will be an initial release in M24. Full release will be in M30.

Bugs

None yet

Patches

None yet

10 Conclusion

D3.2 is the first deliverable on the prototype implementation and associated documentation of the Amigo middleware. It comprises the present document and a multitude of other delivered material:

- Developed source code of essential Amigo middleware components;
- Developed ontologies in OWL constituting the service description vocabulary and language;
- User's guide and developer's guide documents for components and ontologies; and
- Accompanying Javadoc-style and OWLDoc electronic documentation.

Delivered material besides the present document can be accessed – in a restricted way – on the Amigo OSS Repository - Public Web site [Amigo-OSS-Pub]. While we already deliver the first version of several Amigo middleware components, we also report on ongoing conceptual and design work for other middleware components.

D3.2 addresses the Amigo programming and deployment framework (first implementation available), service description vocabulary and language (specification in OWL available), aspects of service discovery (conceptual and experimental work), service discovery and interaction interoperability (detailed design and evaluation work), domotic infrastructure (under development), security (first implementation available), content delivery (under development), and data store (under development).

In our development of the Amigo middleware components, we follow the component overview and associated timeline stated in the Intermediate Amigo OSS Report [Amigo-OSSReport]. Our prototype implementation of the Amigo middleware is currently at a satisfactory stage, enabling developers of Amigo intelligent user services (WP4) and applications (WP5, WP6, WP7) to already employ essential middleware functionalities in their developments. We have organized and carried out two internal tutorial workshops for all Amigo developers, and especially the ones coming from WP4-WP7, on the development of an Amigo service on top of the basic discovery and interaction middleware functionalities; our focus was on presenting principles and practices established by our programming and development framework on both the OSGi and .NET platforms. Amigo developers from WP4-WP7 have already declared their preferences for one of the two development platforms. We are currently rapidly enhancing the basic development capabilities covered by these tutorials with advanced ones integrating additional middleware functionalities.

Appendix A

MICROSOFT EMIC AMIGO SHARED SOURCE LICENSE FOR NONCOMMERCIAL USE

"The Amigo partners are licensed to use the Deliverable in accordance with the Amigo Consortium Agreement and EU Contract. If and when the Deliverable is released for use by the general public on the terms of the licence below, the Amigo partners (as well as the general public) may also use the Deliverable upon the terms of such licence. However, their use of the Deliverable upon the terms of such licence shall not limit their rights under the Amigo Consortium Agreement or EU Contract."

This License governs use of the accompanying Software (including source code), and your use of the Software constitutes acceptance of this license. If you do not accept all the terms of this license, you must not use the Software.

You may use this Software for any non-commercial purpose, subject to the restrictions in this License. Some purposes which can be non-commercial are teaching, academic research, and personal experimentation. You may also distribute this Software with books or other teaching materials, or publish the Software on websites, that are intended to teach the use of the Software.

You may not use or distribute this Software or any derivative works in any form for commercial purposes. Examples of commercial purposes would be running business operations, licensing, leasing, or selling the Software, or distributing the Software for use with commercial products.

You may modify this Software and distribute the modified Software for non-commercial purposes, however, you may not grant rights to the Software or derivative works that are broader than those provided by this License. For example, you may not distribute modifications of the Software under terms that would permit commercial use, or under terms that purport to require the Software or derivative works to be sublicensed to others.

You may use any information in intangible form that you remember after accessing the Software. However, this right does not grant you a license to any of Microsoft's copyrights or patents for anything you might create using such information.

In return, you agree:

1. Not to remove any copyright or other notices from the Software.

2. That if you distribute the Software in source or object form, you will include a verbatim copy of this License.

3. That if you distribute derivative works of the Software in source code form you do so only under a license that includes all of the provisions of this License, and if you distribute derivative works of the Software solely in object form you shall do so only under a license that complies with this License.

4. That if you have modified the Software or created derivative works, and distribute such modifications or derivative works, you will cause the modified files to carry prominent notices so that recipients know that they are not receiving the original Software. Such

notices must state: (i) that you have changed the Software; and (ii) the date of any changes.

5. **THAT THE SOFTWARE COMES "AS IS", WITH NO REPRESENTATIONS, WARRANTIES OR CONDITIONS. THIS MEANS NO EXPRESS, IMPLIED OR STATUTORY REPRESENTATION, WARRANTY OR CONDITION, INCLUDING (WITHOUT LIMITING THE SCOPE OF THIS EXCLUSION), WARRANTIES OR CONDITIONS CONCERNING THE QUALITY OF OR FITNESS FOR ANY PURPOSE OF THE SOFTWARE OR ANY REPRESENTATION OR WARRANTY OF TITLE OR THAT THE USE OF THE SOFTWARE WILL NOT RESULT IN THE INFRINGEMENT OF ANY PERSON'S RIGHTS. ALSO, YOU MUST PASS THIS DISCLAIMER ON WHENEVER YOU DISTRIBUTE THE SOFTWARE OR DERIVATIVE WORKS.**

6. **THAT NEITHER MICROSOFT NOR ANY PERSON OR CORPORATION CONNECETD WITH IT WILL BE LIABLE FOR ANY LOSS OR DAMAGE RELATED TO THE SOFTWARE OR THIS LICENSE. THIS MEANS NO LIABILITY FOR ANY DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL OR INCIDENTAL LOSS OR DAMAGE, NO MATTER WHAT LEGAL THEORY IT IS BASED ON, TO THE MAXIMUM EXTENT THE LAW PERMITSTHIS EXCLUSION. ALSO, YOU MUST PASS THIS LIMITATION OF LIABILITY ON WHENEVER YOU DISTRIBUTE THE SOFTWARE OR DERIVATIVE WORKS.**

7. **THAT THE EXCLUSIONS IN PARAGRAPHS 5 AND 6 ABOVE ARE REASONABLE IN THE CIRCUMSTANCES. IN PARTICULAR, YOU ACKNOWLEDGE (1) THAT THIS SOFTWARE HAS BEEN MADE AVAILABLE TO YOU FREE OF CHARGE, (2) THAT THIS SOFTWARE IS NOT "PRODUCT" QUALITY, BUT HAS BEEN PRODUCED BY A RESEARCH GROUP WHO DESIRE TO MAKE THIS SOFTWARE FREELY AVAILABLE TO PEOPLE WHO WISH TO USE IT FOR NONCOMMERCIAL PURPOSES ONLY, AND (3) THAT BECAUSE THIS SOFTWARE IS NOT OF "PRODUCT" QUALITY (BUT IS THE RESULT OF BASIC RESEARCH), IT IS INEVITABLE THAT THERE WILL BE BUGS AND ERRORS, AND POSSIBLY MORE SERIOUS FAULTS, IN THIS SOFTWARE.**

8. That no technical support will be provided in relation to the Software.

9. That if you sue anyone over patents that you think may apply to the Software or anyone's use of the Software, your license to use the Software under the terms of this License shall end automatically.

10. That your rights under this License shall end automatically if you breach it in any way.

11. That Microsoft reserves all rights not expressly granted to you in this License.

12. That, except to the extent that local laws necessarily apply, this license shall be governed and construed in all respects in accordance with the laws of England and Wales.

References

[Amigo-D2.1] Amigo Consortium. Deliverable D2.1: Specification of the Amigo Abstract Middleware Architecture. April 2005.

[Amigo-D3.1a] Amigo Consortium. Deliverable D3.1a: Detailed Design of the Amigo Middleware Core Service Modelling for Composability. September 2005.

[Amigo-D3.1b] Amigo Consortium. Deliverable D3.1b: Detailed Design of the Amigo Middleware Core – Service Specification, Interoperable Middleware Core. September 2005.

[Amigo-D3.1c] Amigo Consortium. Deliverable D3.1c: Detailed Design of the Amigo Middleware Core – Security & Privacy, Content Distribution, Data Storage. September 2005.

[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

[Amigo-OSSReport] Amigo Consortium. Intermediate Report: Open Source Software Components Planned in Amigo. March 2006.

[Amigo-OSS-SCM] Amigo Consortium. Amigo OSS Repository - Source Code Management (SCM). http://gforge.inria.fr/projects/amigo/

[B2004] Dave Beckett. RDF/XML Syntax Specification (Revised). Technical report, W3C, http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/ , February 2004

[BHS98] N. Bhatti, M. Hiltunen, R. Schlichting, W. Chiu. “Coyote: A system for constructing fine-grain configurable communication services”. ACM Trans. On Computer Systems, 16(4):321-366, 1998.

[BI05] Y.-D. Bromberg, V. Issarny. “INDISS: Interoperable Discovery System for Networked Services”. In Proc. of Middleware 2005.

[CBMEG02] L. Capra, G. Blair, C. Mascolo, W. Emmerich,, P. Grace. “Exploiting reflection in mobile computing middleware”. ACM SIGMOBILE Mobile Computing and Communications Review, 6(4):34-44, 2002.

[DIG] DIG Interface. http://dl.kr.org/dig/

[DL] Description Logics. http://dl.kr.org/

[DOM] Document Object Model (DOM). http://www.w3.org/DOM/

[EG02] R. van Engelen, K. Gallivan. The gSOAP toolkit for web services and peer-to-peer computing networks. In Proc. IEEE International Symposium on Cluster Computing and the Grid, 2002.

[FaCT] Ian Horrocks. The FaCT System. http://www.cs.man.ac.uk/~horrocks/FaCT/

[FaCT++] Ian Horrocks. The FaCT++ System. http://owl.man.ac.uk/factplusplus/

[GBS032] P. Grace, G. Blair, S. Samuel. "ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability". In Proc. of International Symposium on Distributed Objects and Applications (DOA), 2003.

[Grounding] OWL-S Service Grounding ontology.http://www.daml.org/services/owl-s/1.1/Grounding.owl

[GTB01] Javier G, David Trastour, Claudio B. Description Logics for Matchmaking of Services. www.hpl.hp.com/techreports/2001/HPL-2001-265.pdf

[H2003] Ian Horrocks. From /SHIQ/ and RDF to OWL: The making of a web ontology language". *J. of Web Semantics*,1(1):7-26, 2003.

[HK] Matthew Horridge, Holger Knublauch et. Al. A Practical Guide To Building OWL Ontologies Using The Prot´eg´e-OWL Plugin and CO-ODE Tools Edition 1.0

[HR97] M. Hayden, R. van Renesse. “Optimizing Layered Communication Protocols”. In proc. IEEE International Symposium on High Performance Distributed Computing, 1997.

[J2ME] J2ME - Java 2 Micro Edition. http://java.sun.com/j2me/

[Jena] http://jena.sourceforge.net/

[JENA] Jena – A Semantic Web framework for Java.http://jena.sourceforge.net/

[JESS] Java Expert System Shell. http://herzberg.ca.sandia.gov/jess/

[jUDDI] jUDDI Homepage. http://ws.apache.org/juddi/

[KIF] Knowledge Interchange Format: Draft proposed American National Standard (dpans). Technical Report 2/98-004, ANS, 1998. http://logic.stanford.edu/kif/dpans.html

[KKS-MX] Mattthias Klusch, B. Fries, M Khalid, K Sycara. OWL-MX Matcher. http://projects.semwebcentral.org/frs/?group_id=90

[KKS-MXb] Mattthias Klusch, B. Fries, M Khalid, K Sycara. OWLS-MX: Hybrid OWL-S Service Matchmaking www.dfki.de/~klusch/papers/owlsmx-aaai.pdf

[KR03] Joseph Kopena and William Regli. DAMLJessKB: A tool for reasoning with the semantic web. In IEEE Intelligent Systems, volume 18, pages 74-77, May/June 2003.

[KWSS03] D. Kurzyniec, T. Wrzosek, V. Sunderam, A. Slominski. “RMIX: A multiprotocol RMI framework for Java”. In Proc. of the International Parallel and Distributed Processing Symposium (IPDPS), 2003.

[MH-OWL] Deborah L. McGuinness ,Frank van Harmelen. OWL Web Ontology Language Overview. http://www.w3.org/TR/owl-features/

[MT90] S. Martello and P. Toth (1990), “Knapsack Problems: Algorithms and Computer Implementations”, Wiley, Chichester, England.

[MySQL] MySQL Homepage. http://www.mysql.com/

[NAICS] North American Industry Classification System. http://www.census.gov/epcd/www/naics.html

[OJKB] OWLJessKB: A Semantic Web Reasoning Tool http://edge.cs.drexel.edu/assemblies/software/owljesskb/

[OWLPlugin] The Protégé OWL Plug-in. http://protege.stanford.edu/overview/protege-owl.html

[OWL-S] OWL-S 1.1 Release, http://www.daml.org/services/owl-s/1.1/

[PDDL] M. Ghallab et al. PDDL-The Planning Domain Definition Language V.2. Technical Report, report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[Pellet] Pellet OWL reasoner, http://www.mindswap.org/2003/pellet/

[Pizza.owl] http://www.co-ode.org/ontologies/pizza/2005/05/16/pizza.owl

[PKPS02] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic Matching of Web Service Capabilities. LNCS 2342, 2002. www.springerlink.com/index/9LWN9FKCLNGM6TRL.pdf

[Process] OWL-S Service Process ontology.http://www.daml.org/services/owl-s/1.1/Process.owl

[Profile] OWL-S Service Profile ontology. http://www.daml.org/services/owl-s/1.1/Profile.owl

[Protege] http://protege.sourceforge.net/

[Racer] RACER (Renamed ABox and Concept Expression Reasoner). http://www.sts.tu-harburg.de/~r.f.moeller/racer/

[RBFHK95] R. van Renesse, K. Birman, R. Friedman, M.Hayden, D. Karr. “Framework for Protocol Composition in Horus”. In Proc. of International ACM Symposium on Principles of Distributed Computing (PODC), 1995.

[RBM96] R. van Renesse, K. Birman, S. Maffeis. “Horus: a flexible group communication system”. Communications of the ACM, 39(4):76-83, 1996.

[RDQL] RDQL - A Query Language for RDF, W3C Member Submission 9 January 2004, http://www.w3.org/Submission/RDQL/

[Rec03] Modularisation of Domain Ontologies Implemented in Description Logics and related formalisms including OWL, Alan Rector, K-CAP'03, October 23-25,2003. Sanibel Island, Florida, USA. pp 121-128.

[SAX] Simple API for XML (SAX) http://www.saxproject.org/

[S-FOL] Semantic Web Rule Language (SWRL) First Order Logic (FOL) language, The Joint US/EU ad hoc Agent Markup Language Committee, 2 November 2004. http://www.daml.org/2004/11/fol/

[SGGB01] A. Slominski, M. Govindaraju, D. Gannon, R. Bramley. “Design of an XML based Interoperable RMI System: SoapRMI C++/Java 1.1”. In Proc. of Parallel and Distributed Processing Techniques and Applications Conference, 2001.

[SK05] Alex Sinner and Thomas Kleemann. KRHyper - In Your Pocket System Description. CADE 2005, LNAI 3632, pp. 452–457, 2005.

[SPARQL] SPARQL Query Language for RDF, W3C Working Draft, 2006. http://www.w3.org/TR/rdf-sparql-query/

[SPS05] Naveen Srinivasan, Massimo Paolucci, Katia Sycara. An Efficient Algorithm for OWL-S based Semantic Search in UDDI. LNCS 3387, 2005. www.daml.ri.cmu.edu/matchmaker/download/cr-swsws-paper.pdf

[StAX] Streaming API for XML (StAX) http://stax.codehaus.org/

[SUDDI] Naveen Srinivasan. OWL-S UDDI Matchmaker. http://projects.semwebcentral.org/projects/owl-s-uddi-mm/

[SWRL] I. Horrocks, P.F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. Swrl: A semantic web rule language combining owl and ruleml, 2003. http://www.daml.org/2003/11/swrl/

[TTUBa] Stefan Tang. The TUB OWL-S Matcher. http://owlsm.projects.semwebcentral.org/

[TTUBb] Stefan Tang. The TUB OWL-S Matcher Report. http://owlsm.projects.semwebcentral.org/resources/thesis_steftang.pdf

[UDDI] Universal Description, Discovery and Integration (UDDI) http://www.uddi.org/about.html

[UNSPSC] UNSPSC Home Page. http://www.unspsc.org/

[XML] Extensible Markup Language (XML) http://www.w3.org/xml

[ZW97] Amy Moormann Zaremski and Jeannette M. Wing. Specification matching software components. ACM Transactions on Software Engineering and Methodology, 1997.