

IST Amigo Project

Deliverable D3.1b

Detailed Design of the Amigo Middleware Core

Service Specification, Interoperable Middleware Core

Public

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

Deliverable Number	:	D3.1b
Title of Deliverable	:	Detailed Design of the Amigo Middleware Core – Service Specification, Interoperable Middleware Core
Nature of Deliverable	:	Public
Internal Document Number	:	Amigo_WP31b_v1.0.doc
Contractual Delivery Date	:	31 August 2005
Actual Delivery Date	:	12 September 2005
Contributing WPs	:	WP3
Editor	:	INRIA: Nikolaos Georgantas
Author(s)	:	INRIA: Sonia Ben Mokhtar, Yérom-David Bromberg, Nikolaos Georgantas, Noha Ibrahim, Valérie Issarny, Frédéric Le Mouél, Daniele Sacchetti FT: Anne Gerodolle ICCS-NTUA: Miltiades Anagnostou, Ioannis Papaioannou, Ioanna Roussaki, Dimitris Tsesmetzis IKER: Jorge Parra TELIN: Henk Eertink, Remco Poortinga, Andrew Tokmakoff TID: Sara Carro Martinez, José María Miranda, Alvaro Ramos, Johan Zuidweg VTT: Marko Karjalainen

Abstract

This report presents a refinement of the Amigo abstract middleware architecture focusing on the Amigo Interoperable Middleware Core and on Amigo-aware Service Specification. Regarding the middleware core, we provide design and first prototype implementation of service discovery interoperability and service interaction interoperability; we introduce a design for a programming and deployment framework for Amigo services; we provide an early design and implementation of domotic interoperability mechanisms; and we introduce a refined architecture for CE QoS interoperability. These advances in the middleware core have made part of an integrated prototype. Regarding Amigo-aware services, we define a declarative language for semantic service specification and outline a comprehensive conformance relation between services. Finally, we identify classes of networked services and related interoperability levels in the Amigo networked home environment.

Keyword list

ambient intelligence, networked home system, mobile/personal computing/consumer electronics/home automation domain, interoperability, semantic service specification, semantic reasoning, middleware, service discovery protocols, service interaction protocols, programming and deployment framework, context, quality of service, multimedia streaming

Table of Contents

Table of Contents.....	3
Figures	6
Tables.....	9
1 Introduction.....	10
2 Amigo-aware service specification and matching.....	11
2.1 Declarative language for semantic service specification.....	11
2.1.1 General properties of the language.....	12
2.1.2 Specification of service functional properties	13
2.1.2.1 Service capabilities	13
2.1.2.2 Service conversations	15
2.1.2.3 Underlying middleware	16
2.1.3 Specification of service context.....	17
2.1.3.1 Service context-awareness.....	18
2.1.3.2 Service context specification	21
2.1.3.3 Context parameter ontology	22
2.1.4 Specification of service QoS	24
2.1.4.1 QoS parameter ontology	25
2.2 Conformance relations on service specification.....	28
2.3 Tools for on-line semantic reasoning on conformance.....	29
2.3.1 Terminology	29
2.3.2 Semantic reasoning tools.....	30
2.3.3 Discussion	32
3 Service discovery and access in the Amigo networked home environment	33
3.1 The heterogeneous service-based Amigo networked home	33
3.1.1 The Amigo interoperable middleware core	33
3.1.2 The Amigo base middleware.....	35
3.1.3 Networked services integrated in the Amigo home environment	35
3.1.4 Interoperability levels in the Amigo networked home environment	36
3.2 Networked middleware-layer interoperable services.....	37
3.2.1 Interfacing with the middleware core.....	38
3.2.2 Achieving syntactic interoperability.....	39
3.3 Networked Amigo-aware services.....	41
3.3.1 Interfacing with the Amigo base middleware.....	41
3.3.2 Achieving semantic interoperability.....	41
3.4 Discussion	43

4	Amigo interoperable middleware core.....	44
4.1	Service discovery interoperability (SDI)	44
4.1.1	Design principles	44
4.1.2	Detailed design and implementation	47
4.1.2.1	Overview	47
4.1.2.2	Detailed description	48
4.1.3	Evaluation of implementation and performance	63
4.2	Service interaction interoperability (SII)	67
4.2.1	Design principles	67
4.2.2	Early design and implementation	69
4.2.2.1	Overview	69
4.2.2.2	Detailed description	72
4.3	Programming and deployment framework for Amigo services	77
4.3.1	Overview	77
4.3.2	Early programming interfaces design.....	78
4.3.2.1	Basic concepts: AmigoService, AmigoServiceDescription, AmigoAction	78
4.3.2.2	The Amigo enhanced lookup.....	79
4.3.3	Early OSGi-based deployment framework design	79
4.3.3.1	Amigo conformant bundles	81
4.3.3.2	Using OSGi-based Amigo middleware.....	82
4.3.3.3	Implementing drivers and publishers	85
4.4	Domotic interoperability.....	87
4.4.1	Design principles	87
4.4.2	Early design and implementation	88
4.4.2.1	Overview	89
4.4.2.2	Detailed description	91
4.5	Consumer Electronics interoperability	97
4.5.1	Overview	97
4.5.2	Refined Architecture	100
4.5.2.1	Background.....	101
4.5.2.2	QoS interoperable middleware architecture	107
5	Integrated Prototype.....	112
5.1	Scenario and integrated prototype infrastructure	112
5.2	Integrated prototype realization using the Amigo interoperable middleware core	115
5.2.1	Integration of SDI and SII	115
5.2.2	Integration of the OSGi-based framework	117
5.2.3	Integration of domotic interoperability	121
5.3	Integrated prototype visualization.....	122
6	Conclusion.....	124
	Acronyms	125
	References	127

Figures

Figure 2-1: OWL-S top level ontology	13
Figure 2-2: Specification of service capabilities	14
Figure 2-3: Specification of service conversation	16
Figure 2-4: Specification of underlying middleware	17
Figure 2-5: A general model for Context Aware Service Discovery	20
Figure 2-6: Specification of service context	21
Figure 2-7: The Context Parameter Ontology	24
Figure 2-8: Specification of service QoS	25
Figure 2-9: The QoS Parameter Ontology	27
Figure 3-1: Key functions of the Amigo interoperable middleware core	33
Figure 3-2: The Amigo interoperable middleware core and legacy APIs	34
Figure 3-3: Services networked in the Amigo home environment and related interoperability levels	36
Figure 4-1: SDP detection and interoperability mechanisms	45
Figure 4-2: SDP Unit configuration	46
Figure 4-3: SDP interoperability mechanisms	46
Figure 4-4: Detailed design of service discovery interoperability	48
Figure 4-5: Detailed design of event and message communication	48
Figure 4-6: SdpUnit composition configuration decided by Monitor for service discovery interoperability	49
Figure 4-7: SDP Units, SDP Unit Factories and Monitor class diagram	50
Figure 4-8: UML sequence diagram of the initialization process	51
Figure 4-9: SDPMsg, SDPEvent and list of Event types	53
Figure 4-10: Event and message connectors class diagram	54
Figure 4-11: Event publish/subscribe sequence diagram	55
Figure 4-12: Class diagram of the sockets provided by the middleware	56
Figure 4-13: Class diagram of the parsers provided by the middleware	58
Figure 4-14: Class diagram of the composers provided by the middleware	60
Figure 4-15: Unit class diagram	61
Figure 4-16: Extract from UPnP unit state machine	63
Figure 4-17: Native clients & services	65
Figure 4-18: Performance with Amigo located on the service side	65
Figure 4-19: Performance with Amigo located on the client side	66
Figure 4-20: Interaction protocol interoperability relying on event-based parsing	68

Figure 4-21: Interaction protocol interoperability with dynamic stub generation	68
Figure 4-22: Early design of service interaction interoperability components	70
Figure 4-23: Early design of service interaction interoperability	70
Figure 4-24: Diagram of classes used for interaction between service discovery and service interaction interoperability systems	73
Figure 4-25: Diagram of classes used for proxy generation	75
Figure 4-26: Example of generated classes for interfaces, service and proxy	75
Figure 4-27: Diagram of classes used by proxy provider	76
Figure 4-28: Class diagram of Amigo basic concepts: AmigoService, etc.....	79
Figure 4-29: Class diagram of AmigoServiceLookup	80
Figure 4-30: Sequence diagram showing an OSGi discovery example	82
Figure 4-31: An Amigo-aware client uses the OSGi lookup and active discovery to retrieve a Translation Service.....	83
Figure 4-32: An Amigo-aware client uses the OSGi lookup and passive discovery to retrieve a Translation Service.....	83
Figure 4-33: An Amigo-aware client using the Amigo enhanced lookup	84
Figure 4-34: Publication of a translation service according to a specific protocol	84
Figure 4-35: Example of implementing an AmigoService as an UPnPAmigoService.....	85
Figure 4-36: Registering UPnPService of an UPnPDevice as AmigoUPnPService.....	86
Figure 4-37: Lookup and access to the UPnPDevice through the AmigoUPnPService	86
Figure 4-38: Mapping of AmigoUPnPDevice and AmigoUPnPService with UPnPDevice and UPnPService.....	87
Figure 4-39 : Amigo domotic architecture.....	88
Figure 4-40: Amigo Legacy Device Architecture.....	89
Figure 4-41: BDF infrastructure.....	90
Figure 4-42: Amigo Base Device Architecture.....	90
Figure 4-43: RS232 Lamp.....	91
Figure 4-44: RS232 Lamp Class Diagram	91
Figure 4-45: Lamp discovery sequence diagram	92
Figure 4-46: Lamp removal sequence diagram.....	92
Figure 4-47: Action invocation sequence diagram	93
Figure 4-48: BDF Plug.....	93
Figure 4-49: Schuko receptacle.....	94
Figure 4-50: Class diagram for BDF support.....	94
Figure 4-51: Device presentation sequence diagram	96
Figure 4-52: Device removal sequence diagram.....	96

Figure 4-53: Action invocation sequence diagram	97
Figure 4-54: Amigo abstract Multimedia Streaming Architecture	101
Figure 4-55: UPnP QoS scenario (see [UPnPQoS])	102
Figure 4-56: RAPI Interaction.....	106
Figure 4-57: QoS interoperability: UPnP QoS and RSVP interoperability.	108
Figure 4-58: QoS interoperable middleware architecture refined for UPnP QoS and RSVP ...	109
Figure 4-59: Multiple QoS interoperability based on unit pairs.	109
Figure 4-60: Abstract QoS Unit Architecture	110
Figure 5-1: Integrated prototype infrastructure.....	112
Figure 5-2: SDI and SII integration into the prototype	115
Figure 5-3: First OSGi-based deployment architecture	118
Figure 5-4: Second OSGi-based deployment architecture.....	118
Figure 5-5: Sequence for discovery of a UPnP device.....	120
Figure 5-6: Preliminary version of the Amigo integrated prototype visualisation.	122

Tables

Table 2-1: Tools for semantic reasoning.....	32
Table 3-1: Legacy middleware cores experimented with in the Amigo project	38
Table 3-2: Legacy middleware cores and related Web sites for developers	39
Table 4-1: Footprint requirements in KBytes for known libraries and the Amigo middleware core	64
Table 4-2: Identified common quality of service events.....	110

1 Introduction

Ensuring interoperability between devices and applications in the networked home environment has been identified as the principal objective of the Amigo middleware elaborated in work package WP3 of the Amigo project. These devices and applications relate to the four application domains of the Amigo home, i.e., Personal Computing, Mobile, Consumer Electronics and Home Automation domains.

In Deliverable D2.1 [Amigo-D2.1], it was pointed out that interoperability shall be supported both at middleware- and at application service-level. In the present document D3.1b, which constitutes part of Deliverable D3.1 complemented by D3.1a and D3.1c, we address these two levels by focusing on the Amigo Interoperable Middleware Core and on Amigo-aware Service Specification. The middleware core supports interoperability between diverse service discovery and interaction protocols, while service technology-independent, semantic service specification enables integration of heterogeneous services.

More specifically, Deliverable D3.1b provides a considerable advance in the elaboration of the Amigo Interoperable Middleware Core (Chapter 4). We provide design and first prototype implementation of service discovery interoperability and service interaction interoperability. For the former, our design is detailed and covers the generic case, while, for the latter, we provide an early design covering some specific cases. Further, we introduce a design for a programming and deployment framework for Amigo services that aims at guiding and facilitating service development and at enabling dynamic configuration of the Amigo system. Our early design is based on the OSGi framework technology. In addition, we provide an early design and implementation of domotic interoperability mechanisms that enable integration of domotic devices (these are inherently based on low-level discovery and access mechanisms) in the Amigo service architecture. Finally, we introduce a refined architecture for CE interoperability, where we focus on QoS interoperability between the Amigo multimedia streaming architecture inside the home and multimedia streaming infrastructures outside the home. These advances in the middleware core have made part of an integrated prototype (Chapter 5).

We complement this work on the middleware core with work on Amigo-aware service specification and matching (Chapter 2). We elaborate a – informal at this stage – definition of a declarative language for semantic service specification covering both service functional properties and non-functional ones, specifically, context and QoS. Based on this language, we outline a comprehensive conformance relation between services that enables identifying the capacity of services to integrate and interoperate. We intend to realize at a later stage this conformance relation within an online tool reasoning on conformance; for the moment, we have surveyed related literature on available tools for semantic reasoning.

Finally, bridging our two efforts on middleware core and Amigo-aware services, we elaborate a study of service discovery and access in the Amigo networked home environment, identifying classes of networked services and related interoperability levels (Chapter 3). Interoperability may be based solely on middleware-level interoperability mechanisms, where it is syntactic, or both middleware-level and application-level interoperability mechanisms, where the enhanced service characteristics enabled by our declarative language are exploited.

2 Amigo-aware service specification and matching

Interoperability between heterogeneous services in the Amigo home will be realized based on the semantic specification of services. Semantic specification allows a common description of services at a higher, technology-independent, level, thus, enabling integration of service architectures that differ in the ways that services are natively specified and interact, and in the employed communication protocols supporting their interaction. We call 'Amigo-aware services' services that are provided with such semantic specification in addition to their native specification, as opposed to 'legacy services' that are only natively specified.

There are two tasks in WP3, Task 3.1 and Task 3.2, contributing to Amigo-aware service specification. Principal objective of Task 3.1 is to develop a set of ontologies modeling concepts/domains of interest, which can be used as a general-use vocabulary for describing services. Key objective of Task 3.2 is to develop a language (a set of ontologies, too) for semantically specifying services as a set of abstract attributes. The description of a service using this language will refer to the vocabulary of Task 3.1 for giving concrete values to the attributes of the service. Certainly, language and vocabulary shall be in accordance, e.g., addressing the same service attributes. Further, as both 'vocabulary' and 'language' – as we have defined them – are ontologies aimed at describing services, we need to specify the boundary between them. Targeting diverse, heterogeneous services and service architectures, we have opted to make the language as generic as possible, not taking any design decisions that would restrict its range of application. Then, the vocabulary may be seen as a complement to the language, not only for giving concrete values to the attributes of the language, but also for extending the language with new attributes that are appropriate for each specific case. The first results of the work being carried out in Task 3.1 are presented in Deliverable D3.1a [Amigo-D3.1a], while corresponding results of Task 3.2 are presented in this chapter.

More specifically, we introduce in this chapter a declarative language for semantic service specification using as starting point OWL-S, which is currently the most complete effort on semantic specification of services that follow the Web Services architecture (Section 2.1). Our language considerably generalizes OWL-S towards supporting diverse service architectures, as well as non-functional service properties. Then, based on this language, we outline a set of conformance relations on service specification, which aim at checking conformance (matching) between services for assessing their capacity to interoperate (Section 2.2). We finally carry out an initial survey on the state of the art of existing tools for semantic reasoning (Section 2.3). We plan to integrate such tools into an Amigo tool that will realize on-line the identified conformance relations towards dynamic service discovery and matching in the networked home.

2.1 Declarative language for semantic service specification

In Deliverable D2.1, we carried out initial work on the semantic specification of services. Thus, in Chapter 2 of D2.1, a generic ontology was elaborated for modeling services. Targeting interoperability between heterogeneous service architectures, we modeled both services as components and their underlying connectors over which they interact. Our focus was the functional attributes of services, while we identified the need to further model non-functional attributes. Then, in Chapter 3 of D2.1, we indicated a set of non-functional attributes of Amigo services, which shall make part of the specification of a service, collectively identified as quality of service (QoS) and context of the service. Finally, in Chapter 4 of D2.1, we presented initial work on the composition of multiple services based on their semantic specification, and particularly on the specification of their supported conversations.

Building on the base results of Deliverable D2.1, we introduce in this section a declarative language for the semantic specification of Amigo-aware services or simply Amigo services. Our language will use OWL-S as basis. The Ontology Web Language (OWL) is a recent recommendation by W3C supporting formal description of ontologies and reasoning on them. An ontology can represent concepts of any knowledge domain and relations between them. OWL-Services (OWL-S) is an OWL-based ontology for semantically specifying Web services, recently submitted to W3C for adoption. A more detailed report on OWL and OWL-S is provided in Chapter 2 of D2.1. In the same chapter, we identified a number of inadequacies of OWL-S for describing Amigo services, which we tried to cover in our generic modeling approach. Nevertheless, we pointed out that semantic Web services and OWL-S is an important paradigm for Amigo, and we actually employed it in Chapter 4 of D2.1 for composing multiple services. Thus, we have opted to elaborate our declarative language for semantic service specification building on OWL-S. OWL-S, as enabled by its specification, is extensible. A number of our elaborated extensions are authorized by OWL-S, which makes them acceptable as part of a semantic Web service specification. However, certain of our extensions are not authorized by OWL-S. Since we target any service architecture, our language is more general than OWL-S and not any more specifically tied to the Web services architecture. Thus, we have designed our language as a *superset* of OWL-S.

In our elaboration of the language, we take the following approach. Based on the initial results of Deliverable D2.1, we identify a set of required features for the language. Then, for each required feature, we discuss the solution provided or not provided by OWL-S, and present our extension towards specification of Amigo services. At this stage, we define our language in an informal manner, identifying included classes and properties between them. We reuse OWL-S classes and describe extensions to them in a free manner, without specifying the exact way in which we are going to incorporate them into our language. We will provide a formal specification of our language at a later stage. Finally, we illustrate our presentation of the language with a number of ontology diagrams elaborated using the *Protégé* open source ontology editor along with the *OntoViz* visualization plug-in (the latter employs AT&T's *Graphviz* open source graph visualization software). Our definition of the language comprises setting its general properties (Section 2.1.1), and addressing the specification of service functional properties (Section 2.1.2), service context (Section 2.1.3) and service QoS (Section 2.1.4).

2.1.1 General properties of the language

In this section, we establish a set of general properties for our language, which affect several design choices in the following sections, and we indicate how these properties shall be realized. The specific realization for each one of these properties is then detailed in the following sections.

A key property of the language is that it shall be technology-independent, i.e., it shall support any service-oriented architecture, e.g., Web services, RMI, etc. As already discussed, OWL-S is specific to Web services. Thus, we will extend OWL-S by removing its features that are specific to Web services. Reviewing the specification of OWL-S (see Figure 2-1), we observe that the OWL-S Service Profile constitutes a generic ontology class enabling the semantic description of high-level capabilities of a service by reference to general, possibly existing, ontologies that may represent concepts from a number of knowledge domains. Further, the OWL-S Service Model is also a generic class enabling the semantic description of the conversation of a service by employing general workflow structures and by – this one, too – referring to external ontologies. Hence, both the OWL-S Service Profile and Service Model are independent of the Web services architecture and can very well be used to describe services from different service architectures. What makes OWL-S Web services-specific is the OWL-S Service Grounding, which maps the Service Profile and Service Model onto concrete Web service interfaces and interaction protocols (these two constitute the underlying middleware) as represented by the WSDL and SOAP specifications, respectively. Thus, our intervention on

OWL-S will be to remove the uniqueness of the Web services-specific grounding and enable different groundings, towards supporting different service-oriented architectures. This is a non-authorized extension to OWL-S, as OWL-S allows different groundings for Web services, however, it does not allow another service architecture.

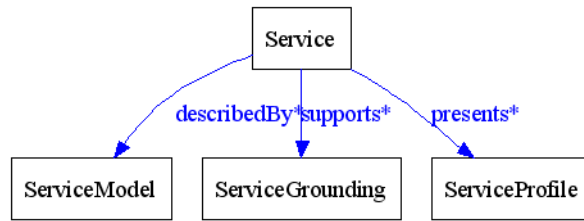


Figure 2-1: OWL-S top level ontology

Supporting the specification of any service-oriented architecture aims at realizing interoperability between these heterogeneous service architectures. Since, each one of these architectures employs native application service specification and interaction schemes over a native middleware infrastructure, both application-level and middleware-level interoperability shall be enabled by our language. This, more specifically, requires that the language provide at both levels the appropriate specification abstractions on which interoperability mechanisms may be built. These abstractions will be detailed in the following sections. OWL-S supports only application-level interoperability, while it imposes a single middleware infrastructure, i.e., Web services. Allowing – as discussed above – different middleware infrastructures (groundings), calls further for their adequate specification that will enable interoperability between them.

Finally, in the ambient, user-centric, intelligent Amigo environment, supporting rich service functionalities by integrating multiple, heterogeneous devices is not enough. The user's actual experience is very much dependent on non-functional properties of service provision, such as context and QoS. Hence, our language shall specify both functional and non-functional properties of services. As discussed in the following sections, such specification spans both the application and the middleware level. OWL-S only supports specification of functional properties at application level; further, the OWL-S Service Profile includes some standard non-functional information of a service. Nevertheless, the Service Profile can be easily extended to describe any non-functional property of a service at application level; we will exploit this feature in our language, which constitutes an authorized extension to OWL-S. For the middleware level, our language will include non-functional properties in the corresponding middleware specification.

2.1.2 Specification of service functional properties

Our specification of service functional properties comprises the specification of service capabilities (Section 2.1.2.1), service conversations (Section 2.1.2.2) and underlying middleware (Section 2.1.2.3). In the following, the supported features and inadequacies of the respective OWL-S Service Profile, Service Model and Service Grounding are discussed, and appropriate extensions are identified for our language.

2.1.2.1 Service capabilities

The main functional service attribute that shall be modeled by our language is a service *capability*, i.e., a specific functionality offered by the service. The OWL-S Service Profile models a service as both:

- A semantic concept, e.g., by specifying the service category on the basis of some external, possibly existing, service taxonomy; and

- A set of semantic IOPEs specifying the data Inputs and Outputs of the service, as well as the Preconditions that need to be fulfilled for the execution of the service and the Effects (Results) produced to the world, e.g., the environment of the service, by the execution of the service.

We adopt this approach in our language. However, we assume that a service may offer a number of capabilities, and we explicitly model capabilities supported by a service. Actually, OWL-S supports multiple profiles for a service; nevertheless, using a different profile for each capability of a service does not allow capabilities to share a set of common attributes, which may globally characterize the service. In our language, each such capability will be defined as both a semantic concept and a set of semantic IOPEs. This enables specifying richer services supporting several capabilities that may be functionally dependent or independent. Further, we explicitly model *provided capabilities* as capabilities supported by a service, and *required capabilities* as capabilities needed by a service, which will be sought on other networked services. This enables support for any service composition scheme, such as a peer-to-peer scheme or a centrally coordinated scheme. Figure 2-2 depicts the specification of service capabilities in our language.

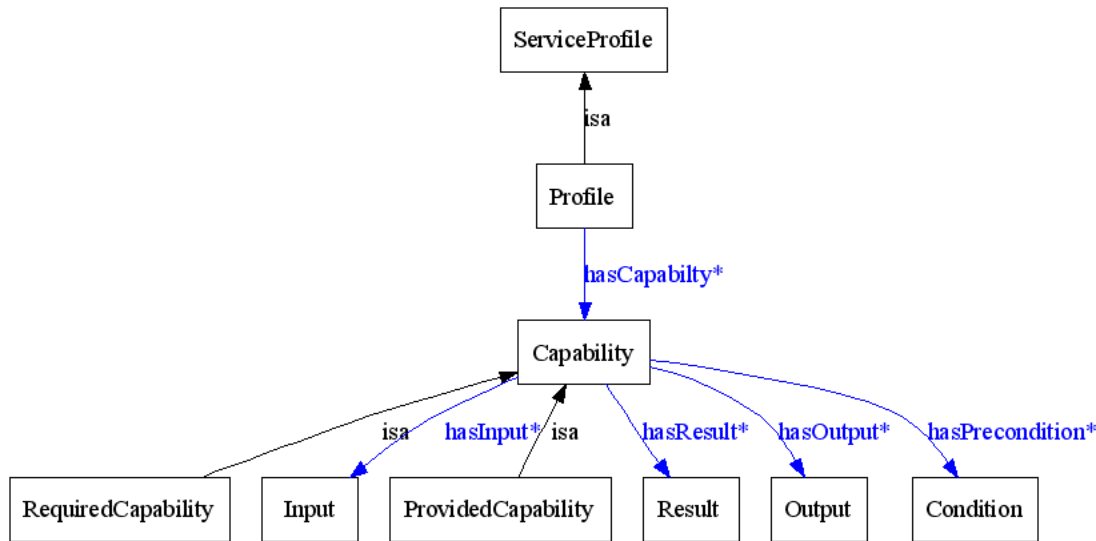


Figure 2-2: Specification of service capabilities

Furthermore, to be able to support any service-oriented architecture, the data types of the Inputs and Outputs of a service capability shall be independent of a specific type system. In service-oriented architectures, this type system is determined either by the programming language (when a single language is used, e.g., Java data type system for RMI), or by the middleware (when language-independent, e.g., XML Schema data type system for Web services). As discussed above, the OWL-S Service Profile and Service Model are independent of the Web services architecture; thus, they employ ontologies to represent data types of Inputs and Outputs. Our language incorporates this feature of the higher-level classes of OWL-S.

Preconditions and Effects are not communicated over the network upon service invocation. Thus, one may select a language for expressing logical formulas, such as SWRL [SWRL], KIF [KIF] or PDDL [PDDL], independently of the service-oriented architecture in use. OWL-S enables the encapsulation of logical formulas expressed in different languages. Our language adopts this handling of Preconditions and Effects.

2.1.2.2 Service conversations

A service capability specified by our language shall further be associated to a service *conversation*. OWL-S describes the conversation supported by a service by employing a single OWL-S Service Model. We incorporate in our language the conversation description approach of OWL-S. Further, following our extension of multiple service capabilities in the previous section, we enable multiple conversation scenarios of a service, one per capability, which, however, make part of one single conversation. This may be enabled by defining several entry points in the conversation workflow or several branches of workflow all starting at a single entry point. Figure 2-3 depicts the specification of service conversation in our language.

Following from our objective to support heterogeneous service-oriented architectures, we identify two more required features for conversation modeling to be supported by our language.

First, our language shall model application-level interaction with a service independently of the interaction model realized by the underlying middleware, such as RPC or event-based. In OWL-S, atomic processes are the elementary units of application-level interaction. An atomic process is defined as a set of IOPEs. Via the Service Grounding, an atomic process is mapped on a WSDL invocation of a Web service operation, which can be a two-way synchronous operation (RPC) or an one-way operation (notification), as prescribed by the WSDL 1.1 specification. WSDL 2.0 will offer a richer interaction model, e.g., it will additionally support two-way asynchronous interactions. Atomic processes of OWL-S are generic enough to be mapped on any interaction model, e.g., an IOPE could be realized by an underlying event-based middleware. Thus, we incorporate the notion of atomic processes in our language.

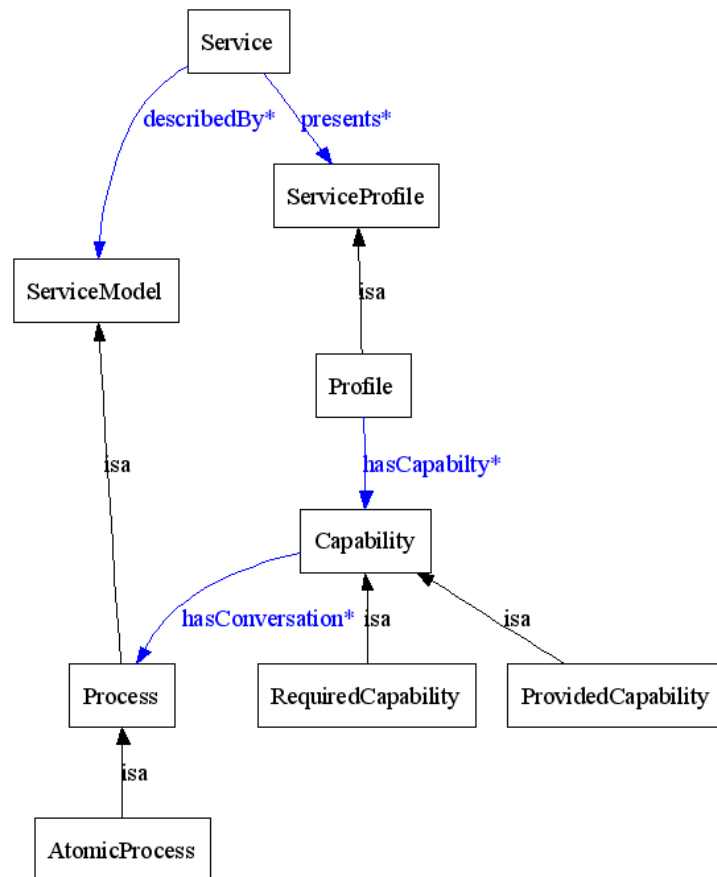


Figure 2-3: Specification of service conversation

Second, similarly to what was identified above for a service capability, the data types of the Inputs and Outputs of an atomic process shall be independent of a specific type system. As indicated above, the OWL-S Service Model is data type system-independent, employing ontologies to represent data types of Inputs and Outputs. Further, Preconditions and Effects are not dependent on a specific type system. Our language incorporates this feature supported by OWL-S.

Finally, to enable automated invocation of a service, the conversation description of a service shall be dynamically interpretable and executable by a specialized execution engine. Automated service execution makes part of service interoperability mechanisms, which we aim to elaborate at a later stage. OWL-S is not directly executable; nevertheless, execution semantics have been proposed in the literature for OWL-S, and there exist implementations of corresponding execution engines [PASS03]. We aim to investigate further execution semantics for our language based on existing efforts addressing OWL-S.

2.1.2.3 Underlying middleware

As pointed out in Section 2.1.1, to support different service architectures, our language shall allow and explicitly provide for different groundings, i.e., different middleware infrastructures. Further, to enable interoperability between them, our language shall provide adequate abstractions for specifying these infrastructures. OWL-S may support different groundings, however, only for Web services, and employs WSDL for specifying concrete bindings to Web service interfaces and deployed interaction protocols like SOAP. Towards multiple middleware, our language will include specification of the underlying middleware in the service specification.

As already discussed in Deliverable D2.1, and as further made evident in the following chapters of the present document, in the diverse, dynamic Aml environment of the Amigo networked home, service discovery is equally important and necessarily precedes service interaction. Thus, we consider both service discovery and interaction as indispensable elements of middleware.

In a first, but major, step, we aim to support well-known middleware platforms integrating specific service interaction protocols, such as Web services/SOAP and Java RMI, coupled with also well-known service discovery protocols, such as UPnP and SLP, respectively. In Chapter 5 of Deliverable D2.1, we provided an abstract architecture for middleware-layer interoperability methods (both for discovery and interaction) based on low-level semantic abstractions of relevant protocols. In Chapter 4 of the present document, design and prototype implementation of these interoperability mechanisms, and, more specifically, support of UPnP/SOAP and SLP/RMI, are provided. Based on this work, the specification of underlying middleware supported by our language may simply be a reference by name to a well-known middleware or to the employed discovery and interaction protocols. Then, interoperability between two services referencing each a native middleware in its specification may directly be assessed on the basis of the availability of appropriate interoperability mechanisms between the referenced protocols.

A second step will be to enable services to be deployed over any (discovery or interaction) connector, possibly associated to a specific capability/conversation of the service. Then, our language shall provide a complete specification of the connector, possibly incorporating external ontologies providing taxonomies of connector attributes. Based on such specification, interoperability between two connectors shall be assessed and potentially dynamically realized. Initial results in this direction were presented in Chapter 2 of D2.1; we aim to investigate further this approach later in the Amigo project. Figure 2-4 depicts the specification of underlying middleware in our language.

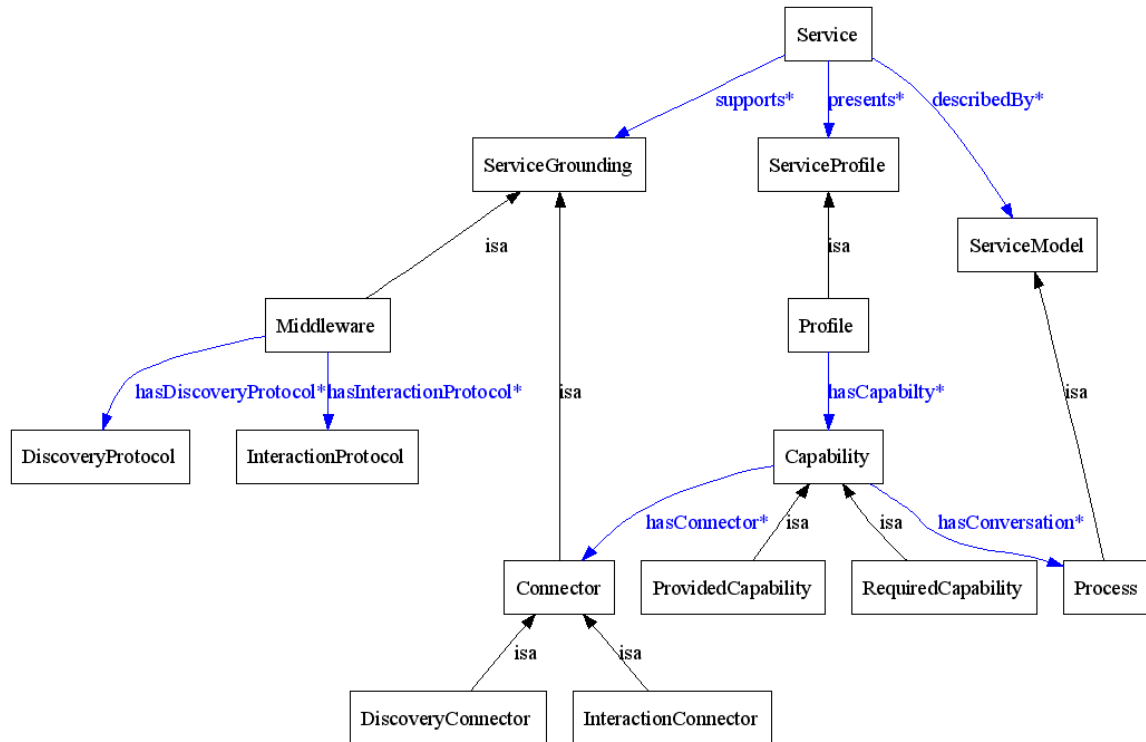


Figure 2-4: Specification of underlying middleware

Further, one of the features characterizing a middleware infrastructure or a connector is the supported interaction model, such as RPC or event-based. As discussed in Section 2.1.2.2, OWL-S adopts the interaction model of WSDL/SOAP, which is quite limited in WSDL 1.1 and will be extended in WSDL 2.0. We aim to support different interaction models, and are particularly interested in event-based middleware or, more abstractly, in event-based connectors. The event-based interaction model is particularly suited for the dynamic, asynchronous Aml environment.

Finally, the data type system-independency as discussed in Sections 2.1.2.1 and 2.1.2.2 shall be complemented here by appropriate middleware-level support. Thus, our language shall allow different type systems employed by middleware. In OWL-S, when associating the Service Model to a concrete Service Grounding, OWL ontologies representing data types of Inputs and Outputs are mapped on XML Schema data types. Our language will enable mapping semantic data types to different syntactic data types, i.e., to different middleware type systems.

2.1.3 Specification of service context

Context is a key notion in the Amigo environment. In Deliverable D2.1, an initial discussion on context in the Amigo home was presented. There, three context categories were identified: (i) *Device Context* that represents the information related to devices, including their characteristics and capabilities, (ii) *User Context* that represents all information that describes individuals, and finally (iii) *Physical Context* that represents the physical environment's specific information. Work on a context model and context management mechanism for the Amigo system is being carried out in WP4.

In this section, we seek to identify what kind of context information is relative to a service, and what context information shall be included in the service specification. In the following, we discuss context-awareness related to services (Section 2.1.3.1), from which we arrive at a

generic specification of service context included in our language (Section 2.1.3.2). We then refine the fundamental *ContextParameter* class of our context specification elaborating a detailed ontology (Section 2.1.3.3).

2.1.3.1 Service context-awareness

Context awareness can be used during service discovery in a number of ways:

- In the discovery phase. This allows applications and requesting services to refine their requests with context parameters, e.g. to request 'the *nearest* printer that has *sufficient* A3-paper available', or 'the display now *in use by Betty*'.
- In the registration phase. The support for context-aware discovery means that services register/publish not only a functional description of their capabilities, but also a collection of context types and corresponding context-information sources that are able to provide actual contextual information of the registered type. These context types must correspond with the context vocabulary ontologies specified in Deliverable D3.1a; the architectural design of context-information sources will be described in more detail in Deliverable D4.1.
- In the discovery/invoke phase. We envision a shift from 'normal' to context-aware services. These services have context inputs in addition to primary inputs, which may not be explicitly provided by the user. The matching process should be able to adapt the rating of services according to the availability of these context inputs (possibly by automatic context source discovery).

However, to enable context awareness in service discovery, the following issues need to be addressed:

- 1) *Expressiveness of the service description language* through which the requested services are described. This language will need extensions to support the formulation of additional contextual constraints, possibly with QoC (Quality of Context) characteristics. Examples of the latter are accuracy and timestamp.
- 2) *Distribution of functionality*. Query-resolution is dependent on the current context of both the requestor and the services that the requestor is trying to discover. For example, 'nearest printer with sufficient A3 paper' depends on the actual location of the requestor and the current amount of A3 paper. How should this be done? Is the requestor responsible for supplying their context information? Is the service provider responsible for supplying the current context of their service? Is the service discovery service (if not peer-to-peer) responsible for obtaining the context of the requestor and/or the provider?
- 3) *Security and privacy*. The service discovery infrastructure (if not completely peer-to-peer) is some kind of man-in-the-middle with respect to context-information exchange between service user and service provider. The privacy-considerations must therefore be carefully considered.

In the following subsections, we analyze these open issues, and suggest some solutions.

Context-aware service specification

Adding context awareness to service discovery means that the *service description formalism* must be extended with features that allow client applications to specify their own context and their context-constraints on the returned service discovery results. Additionally, there is a need for service providers to be able to specify the types of contextual information they can supply on the services that they provide. These context information items are used to optimize selection of the services that are returned to the requesting client application [PoKW03].

The service description formalism must support the following additional aspects:

- *Context constraints.* These are elements that are provided by client applications, and are used to describe additional conditions on the context of the requested service. The basic elements of a constraint is a tuple <context parameter name, condition>, that can be combined using AND/OR boolean operators. Multiple constraints can be specified, and subsequently policy rules (similar to the MIDAS system [BCMS03]) can be used to optimize the results.
- *Context profile.* This profile is specified by the service provider together with its service description. The context profile contains context-parameters, as tuples <context parameter name, context information type, context-information provider>. This context-information provider is either a basic value, or can also be a reference to a service that can be invoked to return the current value of the context parameter.

Each of the types and names must correspond to the context vocabulary ontologies defined in Deliverable D3.1a.

Context-aware service discovery

A general model for service discovery is shown in Figure 2-5, where a Client Application, the Service Discovery Service (SDS) and two Services are shown. This model allows Services to either advertise themselves to the SDS or to be discovered actively by the SDS (e.g. a Multiprotocol Service Discovery Service [Rals05]). In either case, the Services make both their Service specification and their Context information known to the SDS.

Due to its inherently dynamic nature, this context information is likely to change over time. To allow for this, it is possible for a Service to provide a reference to its Service Context Service (which may be located elsewhere and could be a proxy for (a set of) context sources), rather than providing static context information directly to the SDS. The Service Context Service is responsible for making the Service's contextual information available via interaction **(A)** and **(A')**, according to the context vocabulary ontologies defined in Deliverable D3.1a.

By taking advantage of Amigo Awareness and Notification services (as will be discussed in Deliverable D4.1), it is possible for the SDS to “subscribe” to context changes that occur in the Service and which are expressed by the Service's Context Service. In this way, changes in Service context can be “pushed” toward the SDS so that it is always kept “up to date”. Note that there are performance considerations with regard to the tradeoff between push and pull models; this depends on the expected frequency of context changes and also whether all (or just some) of these changes need to be signaled to the SDS.

In general, usage of the SDS involves client applications interacting with the SDS when they wish to discover a certain service (interaction **(C)**). One basic piece of information that needs to be provided by the client to the SDS is the service specification of the Service they require. Beyond this, they may also provide their client context to the SDS (or a reference to their Client Context Service, as discussed earlier for the Services themselves) and also a specification of the desired context of the Service they are searching for, as shown in interaction **(B)**.

From these, the SDS is able to determine matching services based upon the client-supplied Service Specification, which may be either cached service references or discovered dynamically. From this list of matching services, the SDS is then able to apply a filter to the set of matches, applying the client-requested Service context along with the Client context which may be obtained through interaction **(B)**. From this, a final list of matching services is returned to the client in interaction **(C')**. Note that as part of this process, the SDS may choose to actively retrieve a Service's context from its Service Context Service. An example of context aware service matching is given in [BPSK04].

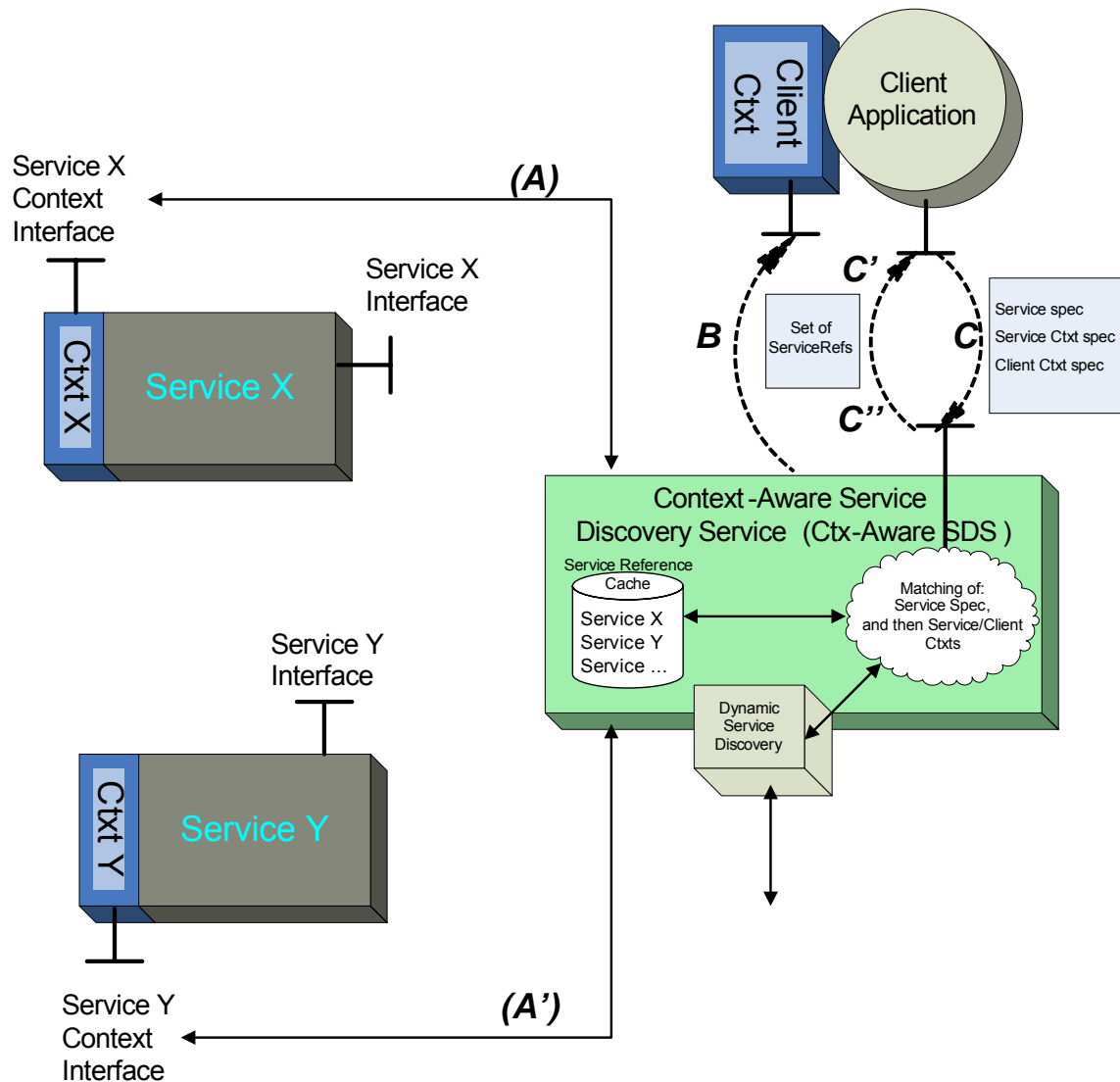


Figure 2-5: A general model for Context Aware Service Discovery

Assuming that a client has interacted with the SDS and obtained a service reference, the client may go ahead and start to interact with the discovered service. However, over time, the context of the client and perhaps also the service may change. In this case, there are two options open to the client application, with regard to optimizing the service binding.

- 1) Issue a new request to the SDS, requesting the same service but making use of its most recent context information (and also that of the available Services) to result in a new set of matching Services, or
- 2) Make use of a "Persistent Request" to the SDS. Such a request can be considered to be a request that has a specified lifetime. If the SDS discovers that a better match than the Service previously suggested to the client is available, then it uses a client callback (interaction **(C'')**) to return an "updated" set of results to the query. This functionality means that client applications can issue a single query and then continue to be updated when more appropriate services become available, either through changes in their own context, or those of the Services which are available. Note also that this feature depends upon the Amigo Awareness and Notification functionality. The

decision to switch to a more relevant service is made by the client. A typical application that could benefit from this functionality is (mobile) multimedia streaming and, in general, applications that have long-lived sessions.

2.1.3.2 Service context specification

Based on the discussion of the previous section, the context specification of a service, as supported by our language, shall include:

- Service context parameters and sources;
- Client (of the service) context parameters and sources;
- Context required by the client; resolution of this requirement may be dependent on both the above; and
- Context inputs of services; these may not be explicitly provided by the client.

Aiming at a generic context specification not posing any restrictions on what context can be in the Amigo environment, we do not define the specific context information that may be associated to an Amigo service in the service context specification included in our language. As we have already indicated, WP4 will provide a context model for the Amigo networked home. Nevertheless, work on context-related vocabulary carried out in Task 3.1 and presented in Deliverable D3.1a attempts to complement the language with more concrete context information. Concluding, the fundamental class of our service context specification is the generic *ContextParameter* class, which represents any context attribute.

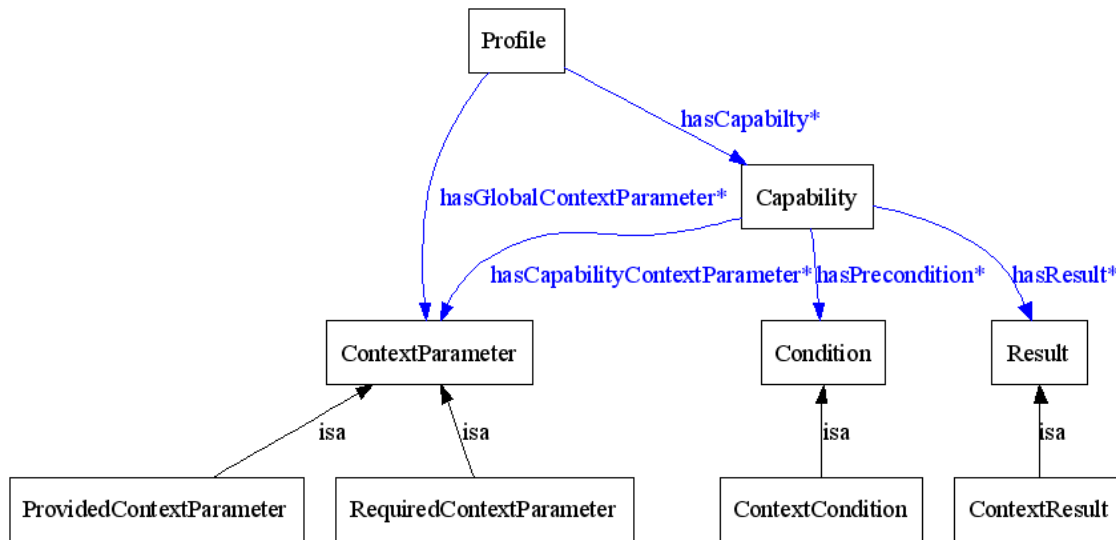


Figure 2-6: Specification of service context

Following this generic approach and according to the above list, we extend the OWL-S Service Profile to include a set of service context attributes. These attributes may be global context attributes of the service or may be associated with a specific service capability. Further, both provided and required context attributes can be specified, thus, enabling – see the above list – service and client context and context required by the client. Additionally, the adopted IOPE scheme describing a specific capability allows us to specify contextual Preconditions and Effects for each service capability. This can be further extended to include atomic processes, which are also described as IOPEs in the Service Model. Nevertheless, contextual Preconditions and Effects included in the Service Profile are generally more visible than the

ones in the Service Model with regard to service discovery. Finally, as indicated in the above list, besides inputs (or outputs) provided by (to) its client, a service may have a number of context inputs (outputs). We consider this as part of the functional inputs/outputs of a service capability. Figure 2-6 depicts the specification of service context in our language.

Our generic service context specification has so far only identified the association of the generic ContextParameter class to specific elements of the Service Profile and Service Model. In the following section, we go one step further refining ContextParameter. We elaborate an ontology that provides a detailed common specification of any context parameter/attribute of a service, among which the source attribute identified in the above list.

2.1.3.3 Context parameter ontology

We introduce a context parameter ontology that provides a standard generic modeling of arbitrary context information originating from various domains. The designed context parameter ontology is depicted in Figure 2-7. It consists of the following classes, most of which are interconnected via object properties:

- *ContextParameter*. The ContextParameter is the centric class of the context ontology and represents a piece of context information, aggregating various object and datatype properties.
- *Type*. The Type is a class introduced to indicate the concrete context attribute represented by a specific instantiation of the ContextParameter (e.g., "User", "AbsoluteLocation", "Device", "Time", "InterfacePreferences", etc.). It is associated with the ContextParameter class through the hasType object property (ContextParameter → domain, Type → range).
- *Metric*. This class defines the way each context parameter is assigned with a value. It is associated with the ContextParameter class through the hasMetric object property (ContextParameter → domain, Metric → range). Each Metric object consists of a MetricType and a Value, which are modeled as datatype properties having xsd:string values. The MetricType datatype property is an enumerated string (xsd:enumeration) that represents the ContextParameter's data type, e.g., int, long, string, boolean, etc. Value is a datatype property that formulates the ContextParameter's value as a string. Together with the MetricType property, the system can easily extract the semantic of this information. The Metric class is also related with the Unit class via the hasUnit object property (Metric → domain, Unit → range) that defines the units used to measure the contextual parameter's quantity. Of course, each context parameter can either be measurable or unmeasurable. In the latter case, the Unit is set to null. As there are various ways to express a physical quantity in terms of units, the Unit class holds a relationship with the ConversionFormula class that is introduced to enable the transformation from one unit to another. Thus, each Unit object is related to a ConversionFormula object via the hasConversionFormula object property (Unit → domain, ConversionFormula → range), while the ConversionFormula class holds a convertsTo object property (ConversionFormula → domain, Unit → range).
- *Domain*. The Domain is a datatype property of the ContextParameter class that represents the domain where the ContextParameter resides. It is an enumerated string (xsd:string (xsd:enumeration)). Potential values are user domain, physical domain (further refined in environmental, space and time domains), device domain, application domain, network domain, object domain and non-human beings domain (see work on context vocabulary ontologies in Deliverable D3.1a).
- *Nature*. The Nature is also a datatype property of the ContextParameter class that is used to distinguish between static and dynamic context information. Static parameters are considered to be those that are not modified in time, while the values of dynamic

parameters may be constantly changing. Obviously, the values of this enumerated datatype property are: "Static" and "Dynamic".

- *Quality*. It is the class that is related to the ContextParameter via the hasQuality object property (ContextParameter → domain, Quality → range) and is used to represent the quality aspects of the context information. Parameters such as accuracy, timeliness, confidence, lifetime, min/max/mean error, etc, are examples of potential datatype properties of the Quality class. Each of these factors can be useful when the system is required to select among various context providers or multiple value measurements of the same context parameter.
- *Source*. The Source is a class that is related to the ContextParameter via the hasSource object property (ContextParameter → domain, Source → range) and is used to represent the source which provided the value of the context parameter. This source can be a device/sensor that performed the relevant context value measurement, a context provider that sold the relevant context information, or even the user that defined some static context data concerning himself/herself.
- *RetrievalMechanism*. It is a datatype property of the ContextParameter class, which represents the mechanism used to retrieve the specific context information. The range of this property is xsd:string (xsd:enumeration), and its enumerated values are: "Sensed", "Inferred", and "Profiled".
- *Status*: The Status datatype property of the ContextParameter class defines the current status of the context parameter, i.e. active or inactive. It is a quite useful feature of the context information, especially for the static context parameters, as it enables for example the user to have multiple preferences and activate/deactivate them at will. The value range of the Status datatype property is xsd:string (xsd:enumeration) and its enumerated values are: "Active" and "Inactive".
- *Timestamp*: The Timestamp datatype property of the ContextParameter class is very important, as it enables the context parameter ontology to capture the history of context information and check its validity. It represents the time when the value of the ContextParameter object was last updated. It is more critical for the dynamic context information, where the various context parameters have different updating demands and valid lifetime, but it is also used for the static context data. Depending on the application requirements, the system may maintain only the current value of each context parameter, or select to monitor the values of some pieces of context information in time, storing pairs of timestamps and values. The later is very useful in context inference mechanisms. The data type used for the Timestamp property is xsd:dateTime that captures the date and the exact time of the parameter's value update.
- *Relationship*. This class represents the way a ContextParameter is correlated with others. It is related to the ContextParameter class via a hasRelationship optional object property (ContextParameter → domain, Relationship → range). In order to interrelate two ContextParameter objects, we introduced an influentialParameter mandatory object property (Relationship → domain, ContextParameter → range) that indicates (i.e. has range) the ContextParameter that has an impact on the "owner" ContextParameter (i.e. the domain of the specific hasRelationship property) of the Relationship. This approach may also handle the case of asymmetric interdependencies between Context parameters. The Relationship may be Proportional, InverselyProportional, Equal, etc, and these factors are modeled using the IFTYPE datatype property that has an xsd:string (xsd:enumeration) value range. The Relationship may also be Strong, Medium or Weak. This information is captured by the ValidityLevel datatype property that also has an xsd:string (xsd:enumeration) value range. The ImpactFactor class is introduced to encapsulate the two properties above (i.e. IFTYPE & ValidityLevel) that

characterize the Relationship. The `hasImpactFactor` object property (Relationship \rightarrow domain, ImpactFactor \rightarrow range) is used to bind a Relationship to an ImpactFactor object.

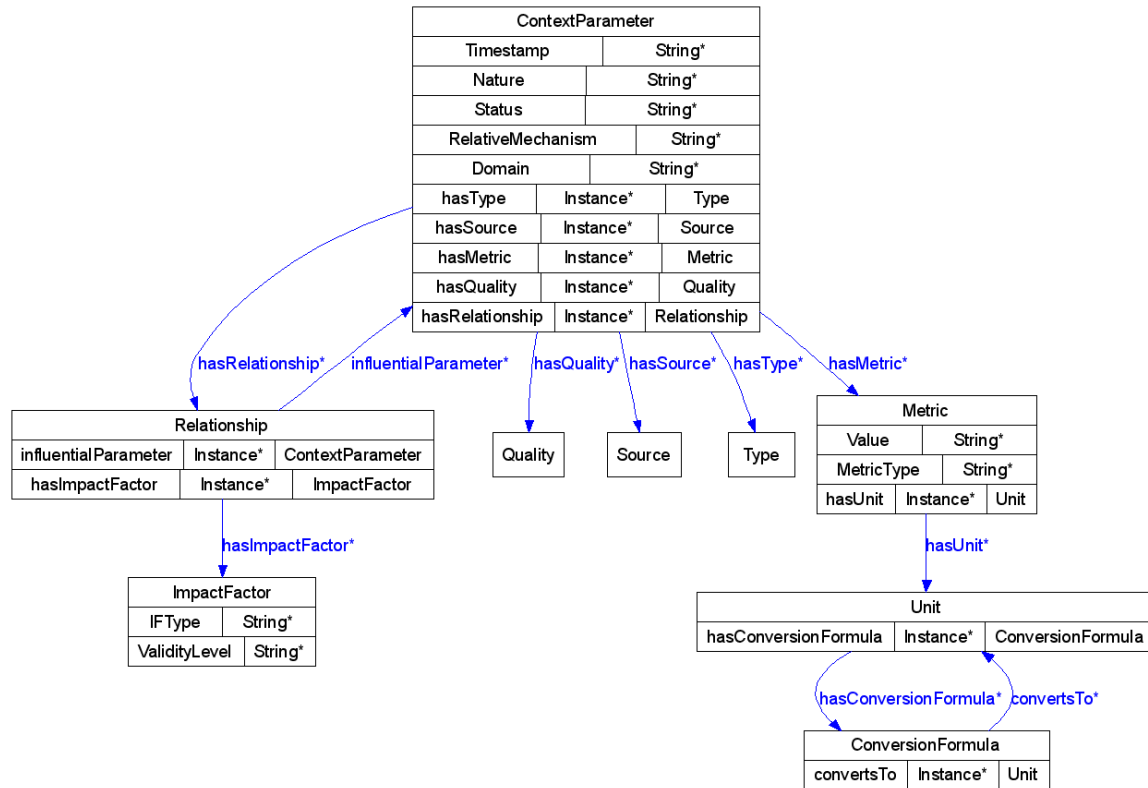


Figure 2-7: The Context Parameter Ontology

2.1.4 Specification of service QoS

QoS assurance is an essential requirement in the Amigo networked home, as it decisively affects user's experience of the supported Aml functionalities. In Deliverable D2.1, an initial discussion on QoS in the Amigo home was presented. There, four main categories of QoS information were identified: (i) runtime-related QoS, (ii) transaction-support QoS, (iii) configuration- & cost-related QoS, and (iv) security-related QoS.

Following the same approach as for context, in this section, we do not attempt to identify the concrete QoS information that may be associated with Amigo services. We have opted to include in our language a generic service QoS specification, where the fundamental *QoSParameter* class representing any QoS attribute is introduced; no concrete QoS attributes are further identified. Complementary work on defining QoS-related vocabulary is being carried out in Task 3.1 and presented in Deliverable D3.1a. Our current results on QoS will be further refined in the course of WP3, possibly towards a QoS model for the Amigo system.

In our generic approach, we initially introduce a general classification for QoS related to a service, in which we identify the different system levels responsible for ensuring such QoS:

- *Application-level QoS*. This is the QoS ensured by the service itself, e.g., response time (performance) of a service is dependent on its computation efficiency, possibly based on programming optimizations.
- *Platform/system-level QoS*. QoS depends also on the platform on which the service executes, e.g., service response time is also dependent on the system OS, CPU, memory, etc.

- *Middleware-level QoS.* Mechanisms deployed by middleware can affect QoS, e.g., middleware may support distributed replication of a server transparently for clients in order to improve service response time in case of high load.
- *Network-level QoS.* QoS certainly depends on the network connection between a service and its client, e.g., reservation of resources along the network path can guarantee sufficient bandwidth for timely delivery of a data stream.

Based on this classification, we identify the diffusion of QoS attributes in the service specification. Figure 2-8 depicts the specification of service QoS in our language.

For specifying application-level and platform/system-level QoS, we extend the OWL-S Service Profile to include a set of QoS attributes, which may be either global QoS attributes of the service or associated with a specific service capability. QoS attributes may be either provided or required, associated, respectively, to provided or required capabilities.

For specifying middleware-level and network-level QoS, related (provided or required) QoS attributes shall be added to the middleware (or connector) specification included in the service specification. In the case of network-level QoS, the middleware (or connector) specification may be extended to include underlying network protocols. These QoS attributes may imply – however, not specify – employment of appropriate middleware-level or network-level mechanisms, e.g., server replication or resource reservation, respectively. Network-level QoS mainly concerns multimedia streaming services of the CE domain in the Amigo networked home. Our approach to CE QoS is detailed in Section 4.5 of Chapter 4.

Finally, we include security & privacy in the QoS properties that are to be ensured by the Amigo system. More specifically, we consider security & privacy features as a set of middleware-level QoS attributes that reflect the Amigo security & privacy architecture [Amigo-D2.1].

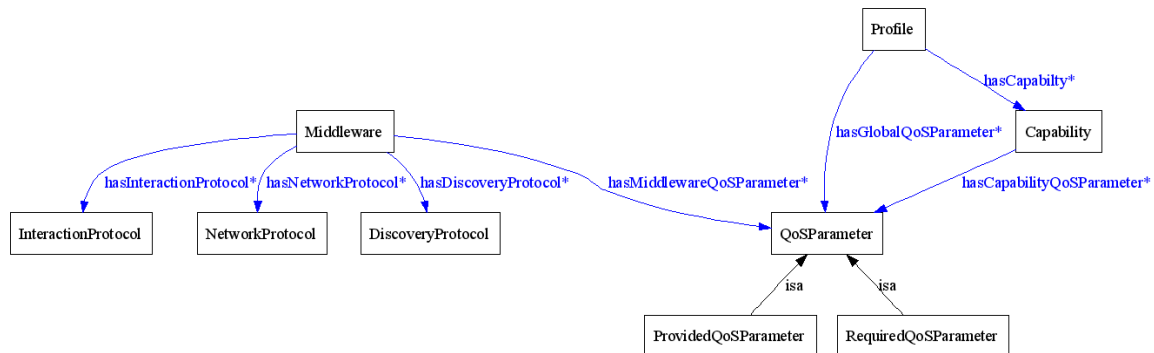


Figure 2-8: Specification of service QoS

In all identified QoS categories, QoS is represented by one or more generic QoS parameters/attributes. In the following section, we refine the fundamental *QoSParameter* class of our QoS specification. As in the case of context, we elaborate an ontology that provides a detailed common specification of any QoS parameter/attribute of a service.

2.1.4.1 QoS parameter ontology

We introduce a QoS parameter ontology that 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 designed QoS parameter ontology is depicted in Figure 2-9. In this ontology, each QoS attribute is described by the following classes:

- *QoSParameter*. QoS parameter represents a non-functional property of the service within a specific domain. These properties may be measurable or not and may hold relationships to each other.
- *Metric*. This class defines the way each QoS parameter is assigned with a value. It is associated with the QoSParameter class through the hasMetric object property (QoSParameter → domain, Metric → range). Each Metric object consists of a MetricType and a Value, which are modeled as datatype properties having xsd:string values. The MetricType datatype property is an enumerated string (xsd:enumeration) that represents the QoSParameter's data type, e.g., int, long, string, boolean, etc. Value is a datatype property that formulates the QoSParameter's value as a string. Together with the MetricType property, the system can easily extract the semantic of this information. The Metric class is also related with the Unit class via the hasUnit object property (Metric → domain, Unit → range) that defines the units used to measure the QoS parameter's quantity. Of course, each QoS parameter can either be measurable or unmeasurable. In the latter case, the Unit is set to null. As there are various ways to express a physical quantity in terms of units, the Unit class holds a relationship with the ConversionFormula class that is introduced to enable the transformation from one unit to another. Thus, each Unit object is related to a ConversionFormula object via the hasConversionFormula object property (Unit → domain, ConversionFormula → range), while the ConversionFormula class holds a convertsTo object property (ConversionFormula → domain, Unit → range). The QoS ontology also supports statistical analysis elements over the monitored QoS parameters. This functionality is provided by the Statistics subclass of Metric that includes various statistical functions.
- *QoSImpact*. The QoSImpact object property represents the way the QoSParameter value contributes to the service quality perceived by the user. For instance, a reduction on the service latency is expected to increase the quality utility for the user. The QoSImpact property enables the system to estimate the degree of user satisfaction with regards to a given QoS parameter measurement.
- *Type*. The Type is a class introduced to indicate the concrete QoS attribute represented by a specific instantiation of the QoSParameter (e.g. "Bandwidth", "Scalability", "SupportedStandards"). It is associated with the QoSParameter class through the hasType object property (QoSParameter → domain, Type → range).
- *Nature*. This datatype property of the QoS parameter represents its static or dynamic nature. A QoSParameter that is defined a priori and does not change during the entire duration of the service session is a Static QoSParameter. On the other hand, QoSParameters that may vary during the service execution time are Dynamic. The values of the Nature datatype property are defined by the Service Provider and are periodically confirmed in the user domain. The Nature property is formulated as enumerated string (xsd:string (xsd:enumeration)), and its enumerated values are: "Static" and "Dynamic". An example of Static QoSParameter is the security protocols supported by the service, while a Dynamic QoSParameter is the service response time.
- *Aggregated*. The QoSParameter that is composed by two or more defined QoSParameters has the object property of aggregation. For example, the service response time is composed by the latency parameter and the request process time by the server.
- *Node*. The Node datatype property of the QoSParameter identifies the network node that may have an impact on its value. Thus, each QoSParameter may depend on the Server node attributes, the Client node attributes or both. It is formulated as an

xsd:string (xsd:enumeration) data type, while its enumerated values are: “client” and “server”.

- Relationship.** This class represents the way a QoSParameter is correlated with others. It is related to the QoSParameter class via a hasRelationship optional object property (QoSParameter → domain, Relationship → range). In order to interrelate two QoSParameter objects the influentialParameter mandatory object property (Relationship → domain, QoSParameter → range) has been introduced, which indicates (i.e. has range) the QoSParameter that has an impact on the “owner” QoSParameter (i.e. the domain of the specific hasRelationship property) of the Relationship. This approach may also handle the case of asymmetric interdependencies between QoS parameters. The Relationship may be Proportional or InverselyProportional. This feature is modeled by the IFTYPE datatype property that has an xsd:string (xsd:enumeration) value range and {“Proportional”, “InverselyProportional”} enumerated values. For example, the service response time and the throughput are InverselyProportional parameters. The Relationship may also be Strong, Medium or Weak. This information is captured by the ValidityLevel datatype property that also has an xsd:string (xsd:enumeration) value range. The ImpactFactor class is introduced to encapsulate the two properties above (i.e. IFTYPE & ValidityLevel) that characterize the Relationship. The hasImpactFactor object property (Relationship → domain, ImpactFactor → range) is used to bind a Relationship to an ImpactFactor object.

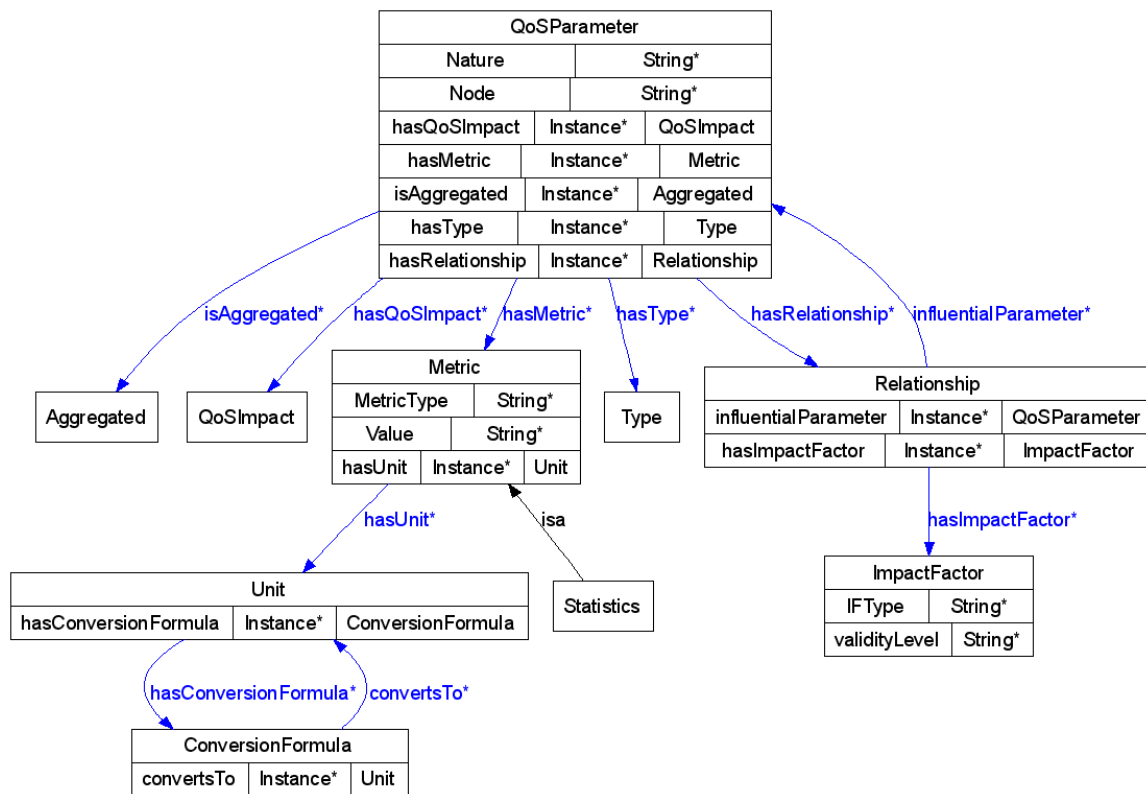


Figure 2-9: The QoS Parameter Ontology.

2.2 Conformance relations on service specification

Conformance relations exploit the semantic specification of services as elaborated in Section 2.1 to assess the capacity of heterogeneous services to integrate and interoperate. In Deliverable D2.1, we established the basis for conformance relations on service specification, and outlined or elaborated a couple of conformance relations for Amigo services. In this section, we review this base work and extend it building on the service specification language of Section 2.1.

Seeking to establish interoperability between heterogeneous service architectures, our conformance relations shall check conformance both at application and at middleware level. Since, in the diverse, heterogeneous Amigo environment, the case of services conforming *totally* (or strongly) to each other will rather be the exception than the rule, our conformance relations shall enable identifying *partial* (or weak) conformance between services. Then, the definition of partial conformance depends on the capacity to deploy an adequate interoperability method to compensate for the non-conforming part. These interoperability methods shall certainly be employed both at application and at middleware level.

In Chapter 2 of Deliverable D2.1, we outlined an application-level conformance relation, which enables checking functional conformance between services and is based on service capabilities. According to this conformance relation, two services may be composed if they require and provide in a complementary way semantically conforming capabilities. Capabilities are compared as semantic concepts and/or as semantic IOPEs. A popular, base capability matching algorithm has been introduced in [PKPS02].

In Chapter 4 of Deliverable D2.1, we elaborated a second application-level conformance relation in the context of composition of multiple services, which also targets functional conformance and is based on service conversations. This conformance relation enforces a stronger conformance: it imposes semantically conforming atomic processes and workflow conformance between service conversations. Workflow conformance is required in certain cases where a service needs to manage its own internal state transitions during the interaction with a remote service. In contrast, in the interaction enabled by the previous conformance relation, a service shall follow the state transitions of the contacted remote service.

Regarding middleware-level conformance, we outlined in Chapter 2 of D2.1 a functional conformance relation for generic connectors, where a set of features are compared to assess interoperability between two connectors. Further, as discussed in Section 2.1.2.3, the availability of middleware-layer interoperability mechanisms for well-known middleware infrastructures allows us to directly infer middleware-level functional conformance between two services.

Based on our service specification language, we now outline a comprehensive conformance relation which covers both application-level and middleware-level conformance and addresses both functional and non-functional service properties. All the base results discussed above are incorporated in our comprehensive conformance relation. Then, depending on the specific case, a subset of this relation will be applied. Thus, our comprehensive conformance relation shall be able to check conformance between two or more services in terms of:

- *Service capabilities.* Provided and required capabilities will be matched as semantic concepts and/or as semantic IOPEs.
- *Service conversations.* Atomic processes will be semantically matched. Workflows will be matched in terms of structure.
- *Underlying middleware.* Well-known middleware platforms may directly be matched based on available interoperability mechanisms. Generic connectors will be matched in terms of a set of features, which is under study.

- *Context attributes of services/service capabilities.* Context attributes will be matched in terms of their values. Contextual Preconditions and Effects of service capabilities will be evaluated as logical formulas.
- *QoS attributes of services/service capabilities.* QoS attributes will be matched in terms of their values.
- *QoS attributes of underlying middleware/network.* Such attributes will trigger deployment of appropriate middleware/network mechanisms.
- *Security & privacy attributes of middleware.* Such attributes indicate the availability of appropriate middleware mechanisms.

2.3 Tools for on-line semantic reasoning on conformance

This section surveys existing software tools that could be used for online (dynamic) semantic reasoning. Main focus is on open source tools that can provide support for the Web Ontology Language (OWL). The implementation language and software license are also studied.

2.3.1 Terminology

OWL

The OWL Web Ontology Language (OWL) is a language for defining and instantiating Web ontologies. An *OWL ontology* may include descriptions of *classes*, *properties* and their instances. OWL specification includes three sublanguages, OWL Lite, OWL-DL and OWL Full¹.

Semantic reasoning

The Web Ontology Language (OWL) enables (via software tools) semantic reasoning of the data. For example if $x \rightarrow y$ and $y \rightarrow z$ then $x \rightarrow z$. In the web services it enables of discovering services that semantically identical (using sameAs construct). So reasoning means finding information that may not be explicitly stated in the knowledge base but can be “reasoned” from or is entailed in the data. OWL enables merging of ontologies and creating possibly complex ontologies so writing reasoning tools is not an easy thing to do. It also depends on what OWL sublanguage is used, for OWL Full it is unlikely that any reasoning software will be able to support every feature of the language.

OWL Lite

OWL Lite supports those users primarily needing a classification hierarchy and simple constraint features. For example, while OWL Lite supports cardinality constraints, it only permits cardinality values of 0 or 1. It should be simpler to provide tool support for OWL Lite than its more expressive relatives, and provide a quick migration path for thesauri and other taxonomies.

OWL-DL

OWL-DL is one of three species of OWL sublanguages. *OWL DL* supports those users who want the maximum expressiveness without losing computational completeness (all entailments

¹ See <http://www.w3.org/TR/owl-features/> for a more comprehensive overview. In this section, we briefly describe the different variants of OWL.

are guaranteed to be computed) and decidability (all computations will finish in finite time) of reasoning systems. OWL DL includes all OWL language constructs with restrictions such as type separation (a class can not also be an individual or property, a property can not also be an individual or class). OWL DL is so named due to its correspondence with *description logics*, a field of research that has studied a particular decidable fragment of first order logic. OWL DL was designed to support the existing Description Logic business segment and has desirable computational properties for reasoning systems.

SWRL

Semantic Web Rule Language (SWRL) is a combination of OWL-DL and OWL Lite sublanguages with the Unary/Binary Datalog RuleML sublanguages of the Rule Markup Language. The SWRL proposal extends the set of OWL axioms to include Horn-like rules.

2.3.2 Semantic reasoning tools

This section describes the semantic reasoning tools. Supported formats, programming languages, and software licenses are described. Descriptions also contain information whether the tool is open source or not. A list of tools and their descriptions can be found in Table 2-1.

Tool Name	Description
Jena	Open source Java framework for building semantic web applications. Supports OWL reasoning and provides framework for developing with RDF, RDFS, and OWL. Jena is available under BSD License. http://jena.sourceforge.net/
KAON	Extends RFDS with symmetric, transitive and inverse relations, relation cardinality, meta-modeling, etc. KAON (version 1.x.x) is open source software but it does not support OWL. Online demo available (requires Java Web start). LGPL License. http://kaon.semanticweb.org/
KAON2	Reasoner for OWL-DL and SWRL. It also provides an OWL API. KAON2 is not open source software. KAON2 is a commercial product. A precompiled binary distribution is free of charge for research and academic purposes. http://kaon2.semanticweb.org/
Pellet OWL Reasoner	Pellet is an open source Java based OWL DL reasoner and available under MIT license. Online demo available. Pellet is available under the MIT License. http://www.mindswap.org/2003/pellet/index.shtml
WonderWeb OWL API	Online OWL Ontology validator. Online version: http://phoebus.cs.man.ac.uk:9999/OWL/Validator WonderWeb OWL API provides programmatic (Java) access to data structures representing OWL ontologies. OWL API is open source software and it supports OWL Lite and OWL DL. According to the web site this is work in progress and should at best be considered alpha quality code. The API has been developed as part of the EU IST project WonderWeb . OWL API is available under GNU Lesser General Public License. http://owl.man.ac.uk/api.shtml
FaCT++	FaCT++ is an (C++) implementation of an OWL-Lite reasoner. It is new generation of the FaCT reasoner. It will support OWL DL language in the

	<p>future. FaCT++ is open source software published under GNU General Public License.</p> <p>http://owl.man.ac.uk/factplusplus/</p>
OWLJessKB	<p>OWLJessKB is an open source description logic reasoner for the W3C's Ontology Web Language written in Java programming language. The semantics of the language is implemented using Jess, The Java Expert System Shell. It also uses Jena class library. Includes most of the common features of OWL lite, plus some and minus some. It is released under GNU General Public License.</p> <p>http://edge.cs.drexel.edu/assemblies/software/owljesskb/</p>
SOFA	<p>SOFA (Simple Ontology Framework API) is open source project aimed for development of an integral software infrastructure and a common development platform for various ontology-oriented and ontology-based software applications. It is implemented using Java programming language. The SOFA ontology model is independent from specific languages, but it includes Ontology serialization packages for OWL, DAML+OIL and RDF/RDF Schema. SOFA is available under the terms of GNU Lesser General Public License (LGPL).</p> <p>http://sofa.projects.semwebcentral.org/</p>
Sesame	<p>Sesame is an open source Java framework for storing, querying and reasoning with RDF and RDF Schema. It can be used as a database for RDF and RDF Schema, or as a Java library for applications that need to work with RDF internally. However, it does not support OWL. It is released under LGPL license.</p> <p>http://www.openrdf.org</p>
Java RDF (JRDF)	<p>JRDF is an attempt to create a standard set of APIs and base implementations to RDF (Resource Description Framework) using Java. JRDF has no support for the OWL. The Apache Software License, Version 1.1.</p> <p>http://jrdf.sourceforge.net/</p>
Redland RDF Application Framework	<p>Redland is a set of free (open source) software packages that provide support for the Resource Description Framework (RDF). It is written in C and provides bindings for several programming languages including C#, Java and Python. Closely related software include Rasqal RDF Query Library and Raptor RDF Parser toolkit.</p> <p>It is available under LGPL Version 2.1, GPL 2 or Apache License Version 2.0.</p> <p>http://librdf.org/</p>
OWL-S Matcher	<p>The OWL-S Matcher is a Java implementation of a matchmaking algorithm for matching OWL-S descriptions. The matching algorithm is presented in [JRGL05]. Source code is available under LGPL 2.1.</p> <p>http://owlsm.projects.semwebcentral.org/</p>
OWL-S Editor	<p>Java-based tool for creating, validating and visualizing OWL-S models.</p> <p>http://staff.um.edu.mt/cabe2/supervising/undergraduate/owlseeditFYP/OwlSEdit.html</p>
RacerPro	<p>RACER stands for Renamed ABox and Concept Expression Reasoner. RacerPro is the commercial name of the software. RacerPro supports RDF</p>

	<p>RacerPro is the commercial name of the software. RacerPro supports RDF and OWL. RacerPro is commercial product, but trial version can be downloaded (test for 30 days). According to the web site Educational license can be obtained for free and can be used for 180 days.</p> <p>http://www.franz.com/products/racer/</p>
--	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Table 2-1: Tools for semantic reasoning

2.3.3 Discussion

The previous section gave a brief overview of the semantic reasoning tools. Based on the needs of Amigo project, perhaps the most complete open source tool available is Jena. Jena is used in many projects and also it has the most downloads in survey done in [BiWe]. Jena supports RDF, RFDS and OWL. It is open source and it is written in Java programming language. Many other open source tools are not mature enough or support only RDF and not OWL. Of the commercial products, RacerPro would be suitable for the Amigo because it supports OWL and it is implemented as a server and so can be easily accessed from internet/intranet. However, RacerPro is not an open source product. RacerPro evaluation license can be used only for 30 (trial version) or 180 (educational license) days.

3 Service discovery and access in the Amigo networked home environment

3.1 The heterogeneous service-based Amigo networked home

As presented in Deliverable D2.1 [Amigo-D2.1] on the specification of the Amigo abstract middleware architecture and as further detailed in the next chapter, the Amigo middleware allows integrating services based on heterogeneous middleware technologies, in the networked home environment. Such a feature is enabled by the Amigo interoperable middleware core that implements middleware-layer interoperability methods so that services of the networked home environment may be discovered and accessed by the other networked services, and conversely, independent of the service-oriented middleware technology the various networked services are implemented upon.

3.1.1 The Amigo interoperable middleware core

Key functions of the Amigo interoperable middleware core are depicted in Figure 3-1, which illustrates interoperability between a JINI and a UPnP service. Functions of the Amigo interoperable middleware core lie in:

- *SDP detection and interoperability* for service discovery independent of the specific service discovery protocols used by networked services for advertising and requesting services (e.g., JINI and SSDP in our example), and
- *Service interaction interoperability (SII)* for enabling interaction between services, independent of the specific brokers (interaction protocols) used by networked services for being accessed and/or accessing remote services (e.g., RMI and SOAP in our example).

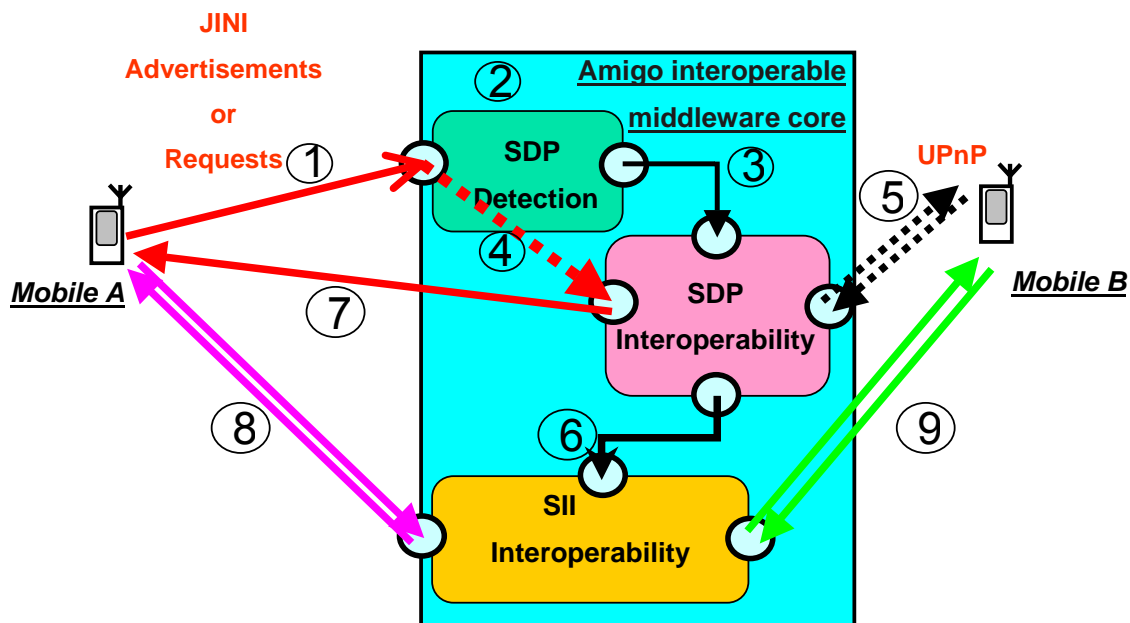


Figure 3-1: Key functions of the Amigo interoperable middleware core

The Amigo interoperable middleware core may be deployed on one of the following three types of nodes of the networked home environment: service client, service provider or

gateway, as most convenient according to the specific architecture of the networked home environment, and capabilities and usage of the networked devices. For instance, it is most convenient to embed the Amigo interoperable middleware in mobile devices, as this will allow them to integrate in any networked home environment, as they move from one network to another. However, the interoperability methods implemented by the middleware introduce processing overhead, which should be accounted for, in particular in the case of resource-constrained, wireless devices. Also, it is much convenient to deploy the Amigo interoperable middleware core on the home gateway – if/when available – since it eases integration of devices as they join the Amigo networked home environment, hence making the environment open and highly adaptive.

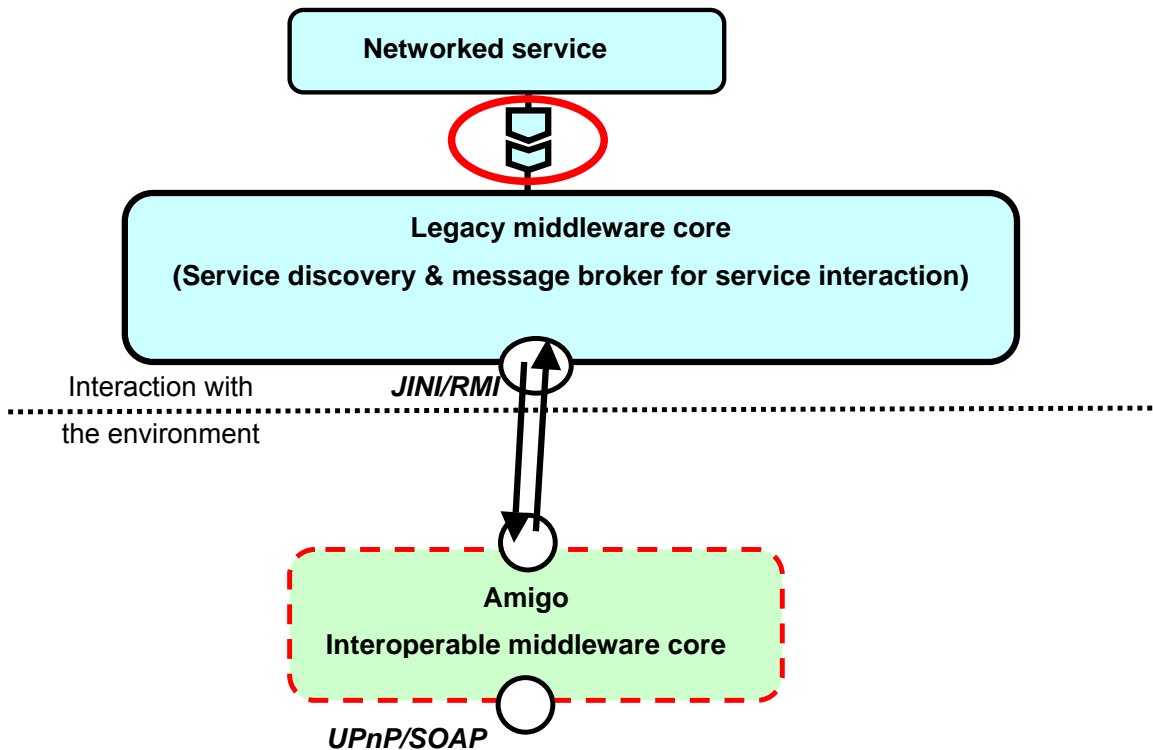


Figure 3-2: The Amigo interoperable middleware core and legacy APIs

One key feature of the Amigo interoperable middleware core is that it is transparent to applications. As depicted in Figure 3-2, services that are networked in the Amigo home environment use legacy middleware core (e.g., JINI in our example) to interact with the environment, and in particular with remote networked services. The Amigo interoperable middleware core then interposes at the network layer to ensure interoperability with services based on distinct middleware technologies (e.g., interaction with a UPnP service in our example). In general, the Amigo middleware does not introduce any new API for base functions of the middleware core, as there is a significant number of such APIs that have already been proven quite successful for the networking of services in various environments (e.g., UPnP for home networks whether ad hoc or infrastructure-based, JINI for Intranet/PC networks, Web services and .NET for Internet/Intranet networks). Instead, the Amigo middleware supports the development of services using the legacy middleware technologies that are the most appropriate to the services being developed, both in terms of networking features and developer skills. The Amigo middleware then takes in charge the integration of the service in today's, rich heterogeneous networked home environment. Hence, services that are networked in the Amigo home environment, whether legacy or Amigo-aware, are all

developed using legacy middleware technologies for functionalities relevant to the middleware core.

3.1.2 The Amigo base middleware

In addition to the above interoperable middleware core, the Amigo base middleware for the networked home environment introduces a number of value-added middleware functions that may be exploited by developers when developing Amigo-aware services. These functions include support for semantic service specification and related conformance checking, as presented in the previous chapter, which is the corner stone of application-layer service interoperability. Additional functions relate to:

- Amigo-aware service discovery that enables semantic-level, context-aware service discovery,
- service composition,
- content interoperability,
- storage & distribution,
- security & privacy,
- accounting & billing, and
- mobility management.

These functions are partly presented in companion Deliverable D3.1c [Amigo-D3.1c]; related APIs will be defined in the next phase of the project and be presented in Deliverable D3.3.

3.1.3 Networked services integrated in the Amigo home environment

Figure 3-3 provides an overview of the various types of networked services that may be integrated in the Amigo home environment, together with the interactions that may effectively take place among networked services (depicted with arrows), according to the interoperability methods embedded on the devices. We distinguish between service client and provider (termed service), noticing that the underlying legacy middleware instance differs on the client and provider side, as the client side is lighter-weight and the service provider side embeds client capabilities in addition to server ones. However, in general, we assume rich services that act both as service client and provider, networking opportunistically (using various communication paradigms such as, e.g., client-server, peer-to-peer or event-based communication) with other services, as service requester and/or provider, in the home environment. Three types of services (acting as client and/or provider) are then distinguished in the Amigo networked home environment:

- **Legacy services:** These are legacy services that do not have any knowledge about Amigo networks and are implemented on top of some legacy (service-oriented) middleware.
- **Middleware-layer interoperable services:** These are services that integrate the Amigo interoperable middleware core, and as above, are implemented using the API of some legacy (service-oriented) middleware.
- **Amigo-aware services:** These are services exploiting the overall functionalities of the Amigo base middleware, i.e., integrating both application- and middleware-layer interoperability and possibly part of the aforementioned advanced middleware functions although not depicted in our figure.

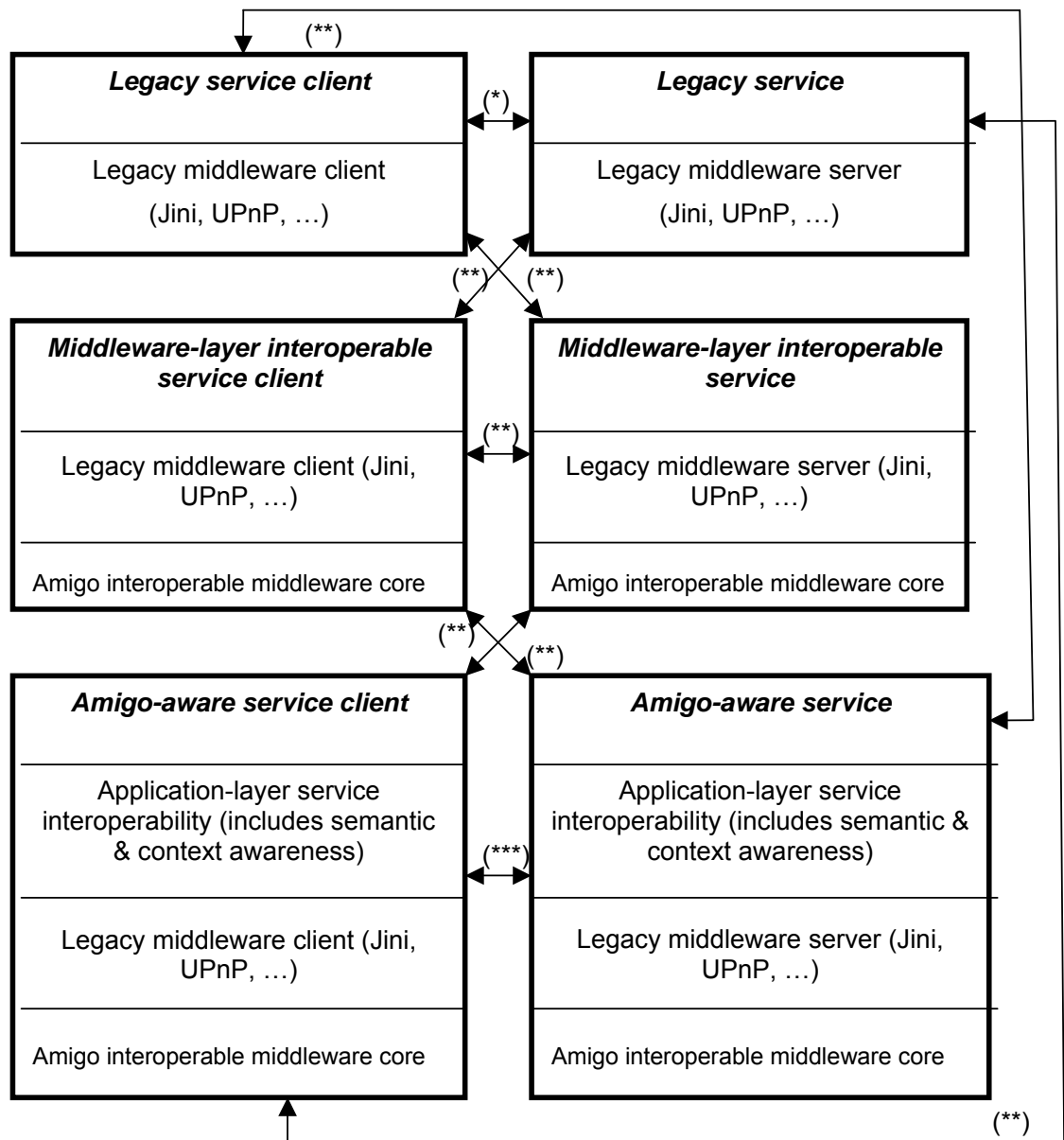


Figure 3-3: Services networked in the Amigo home environment and related interoperability levels

3.1.4 Interoperability levels in the Amigo networked home environment

Following the above classification, we identify three levels of interoperability (denoted by arrows in Figure 3-3) between networked service clients and providers, from (*) to (***):

- **(*)-Network-dependent interoperability** is the weakest interoperability enabled in the network, which is when neither the service client nor the service provider hosts the Amigo interoperable middleware core. Then, interoperability is only possible if either:

- the service client and provider are implemented upon the same or compatible legacy middleware technologies (e.g., a Web service client may interact with a UPnP service, assuming the client knows the service instance), or
- there is a gateway node in the network hosting the Amigo interoperable middleware core that may act as a bridge between the service client and provider for them to be interoperable, hence actually enabling (**)syntactic interoperability discussed below.

The former case is no longer discussed since this corresponds to a homogeneous networked home environment.

- **(**)-Syntactic interoperability** corresponds to middleware-layer interoperability, as achieved by the Amigo interoperable middleware core, which is transparent to services. Specifically, services are able to integrate and compose within the networked home, regarding both the service discovery and interaction protocols they are running, using the API of the legacy middleware they are implemented upon, thanks to the middleware-layer interoperability methods. However, this only solves possible protocol mismatch, it does not solve possible semantic and syntactic mismatch at the application layer. Indeed, effective interoperability between middleware-layer interoperable services may only be achieved if service clients and providers use common interfaces to characterize services, which may differ however in terms of description languages used (i.e., interfaces match syntactically, assuming underlying semantic matching of the respective service models).
- **(**)-Semantic interoperability** corresponds to application-layer interoperability, as achieved by the Amigo base middleware, using semantic description of service behavior. This enables integration and composition of services in the networked home based on semantic knowledge about services. Services may further benefit from advanced middleware functions like context-and QoS-awareness.

Note that the above does not consider the case of *semantic-aware services* that integrate Amigo application-layer interoperability methods but not the Amigo interoperable middleware core. Such a class of services may indeed be developed using a subset of the Amigo middleware. In this case, the level of interoperability that may be achieved depends on the middleware core hosted by the networked services between which the interactions take place, ranging from the network-dependent to semantic interoperability levels.

The two next sections detail how to develop services that enable syntactic, respectively semantic, interoperability in the Amigo networked home environment. Section 3.4 concludes this chapter with an overview of its contribution and area with future work.

3.2 Networked middleware-layer interoperable services

As already stressed, the development of middleware-layer interoperable services is similar to that of legacy services, thanks to the transparency of the Amigo interoperable middleware core. In the time frame of the Amigo project, due to obvious time and budget constraints, software prototypes of the Amigo middleware core will be developed for experimenting interoperability with a fixed set of legacy middleware technologies. Specifically, we will focus on the legacy middleware technologies that are the most popular in the domains integrated by the Amigo networked home environment, i.e.:

- UPnP for the home network that is the core of the DLNA architecture integrating the CE, PC and mobile domains,
- Java technologies for PC, mobile and intranet networks, and
- Web services for the PC domain and Intranet/Internet networks.

The following section provides references to relevant Web sites providing support for the development of services based on those technologies. Section 3.2.2 then addresses how to achieve syntactic interoperability in the Amigo networked home environment, as it is a key requirement for effective use of the Amigo base middleware. Indeed, it cannot be assumed (nor enforced) that all networked services will be Amigo-aware: legacy services will (and shall) play a significant role in Amigo networks.

3.2.1 Interfacing with the middleware core

Middleware cores for open networked environments like the Amigo home integrate two key functions for enabling effective integration of services within the network, i.e.:

- a service discovery protocol, so that networked services can discover, and be discovered by, others, and
- a message broker, also referred to as service interaction, so that networked services may access, and be accessed by, others.

From a design perspective, the Amigo middleware core enables interoperability with any legacy middleware that is based on the service-oriented architecture paradigm. However, from a pragmatic standpoint, service-oriented middleware technologies with which interoperability is enabled by the software prototypes of the Amigo middleware core depend on the interoperability units that are (will be) actually developed/available (see next chapter). As stated above, in the time frame of the Amigo project, we concentrate on the middleware technologies that are the most representative of the Amigo application domains, i.e., UPnP, JINI and, more largely, Java-based middleware technologies, and Web services.

Middleware core	Service discovery protocol	Message broker (service interaction protocols)
UPnP	SSDP	SOAP
JINI	JINI	RMI
Java-based	SLP	RMI
Web services in the home	SLP	SOAP

Table 3-1: Legacy middleware cores experimented with in the Amigo project

Considering the above middleware technologies, all of them provide a message broker but do not necessarily integrate a service discovery protocol, as not all of them are specifically aimed at open networked home environments. Specifically, the following middleware cores target open home networks: UPnP integrates SSDP for service discovery and SOAP for service interaction, and Java-based JINI integrates JINI proprietary service discovery protocol and RMI for service access. Then, other Java-based middleware platforms offer necessary functions for service interaction *via* RMI, but do not prescribe any specific service discovery protocol. Also, there exist Java-based middleware like JXTA that offer a service discovery protocol but for networks different than the networked home environment (e.g., JXTA targets peer-to-peer computing on the Internet). Regarding Web services, the supporting middleware defined by W3C offers SOAP for service interaction. In addition, UDDI is often put forward as the standard service discovery protocol for the discovery of services on the Internet, although defined by the OASIS consortium instead of W3C. Also, service discovery protocols that are not bound to a specific message broker have been introduced. This is in particular the case of the SLP protocol, which may be coupled with, e.g., SOAP or RMI. According to the specifics of the Amigo networked home environment and features of the above middleware technologies, we will primarily focus on the integration of the legacy middleware cores listed in Table 3-1, hence developing related units, as further discussed in the next chapter. Additional legacy

middleware cores may be considered as the project evolves, and in particular as prototype applications developed in WP5-WP7 will be refined.

Development of services using the middleware cores listed in Table 3-1 has to adhere to the developer guides of the respective service discovery protocols and message brokers. The interested reader is thus referred to related Web sites for detail about related APIs and implementation guidelines (see Table 3-2).

Middleware technology	Web site for developers
UPnP	http://www.upnp.org/
JINI	http://www.jini.org/ ; http://www.sun.com/software/jini/
RMI	http://java.sun.com/products/jdk/rmi/
SOAP	http://www.w3.org/TR/soap/
SLP	http://www.openslp.org/doc/html/IntroductionToSLP/

Table 3-2: Legacy middleware cores and related Web sites for developers

3.2.2 Achieving syntactic interoperability

The Amigo interoperable middleware core solves the mismatch of the service discovery and interaction protocols, which occurs between the service client and provider's protocols when the client and provider are implemented upon distinct legacy middleware technologies. Specifically, this solves the mismatch occurring at the architectural connector level in terms of protocol behavior and message format. However, this does not solve the mismatch of service description, as provided at the application layer (architectural component level), which provides the semantic of the service solely based on its syntactic description. As discussed in Deliverable D2.1 for the specific case of mapping between enriched service description and legacy SDPs (see §3.2.1.3 of D2.1), mismatch between service descriptions may be solved using *standard taxonomies for service description*, providing standard terms for naming services, operations, and parameters, as, e.g., investigated by the UPnP forum. For instance, in the integrated prototype presented in Chapter 5, we use the interfaces of the standardized UPnP AV Architecture² to elicit taxonomies for the description of both UPnP and Java-based Audio-Video services. While the definition of standard taxonomies for all the application services relevant to the networked home environment is beyond the scope of the Amigo project, the Amigo project will provide taxonomies for service interfaces relevant to the prototype applications that will be developed in WP5-7, building upon the Amigo work on ontologies for the networked home environment carried out in Task 3.1 and relevant standards like standardized device and service descriptions from the UPnP forum.

Standard taxonomies for service description may be exploited to:

- define standard interfaces using the description language of the specific legacy middleware being used (e.g., UPnP XML schema for service description, Java language for Java-based middleware, WSDL for Web services), and
- reason about the matching of service descriptions.

However, this further requires defining the mapping between the service models and data types defined by the middleware languages for service description. Indeed, service description

² <http://www.upnp.org/standardizeddcps/>

languages from legacy service-oriented middleware define different service models and type systems.

Regarding the mapping of type systems of service description languages, definitions exist in the literature (e.g., mapping of WSDL and Java data types, mapping of UPnP and Java data types), in particular thanks to the mapping between data types of service description languages and programming languages, which are defined by legacy middleware. Then, existing solutions may be exploited to elicit standard mapping between type systems of the various service description languages, as, e.g., illustrated in the next chapter.

With respect to the mapping of service models of service description languages, we should distinguish between models used for service discovery and those used for service access, since not all middleware tightly integrate service discovery and service interaction protocols. Consider first the case of service discovery, focusing on the service discovery protocols that are listed in Table 3-1, since these are representative of most legacy protocols in use today for pervasive computing environments. Basically, those protocols enable searching a service, based on the provision of the name of the service type. UPnP further defines a containment relationship for services, using the notion of devices, where a physical device may embed a number of logical devices, each embedding a set of services. Then, it is possible to also seek devices implementing a given device type in UPnP networks. As for SLP, it is further possible to qualify the set of services of interest, by providing values for given known properties. All service discovery protocols use the provided service type name to seek service instances implementing a matching type (based on the equality of type names), and then return the address of those instances. Note that in the case of UPnP, the discovery process is in two steps, as the URL of the service/device description is first returned to then get the address of the service instance(s) from this description. Basically, the notion of services matches in all the above protocols; services are described as providing a set of operations and are sought based on the name of the type they (exactly) implement. The notion of device may further be considered as a special kind of service, which embeds more primitive services. Then, assuming that services in the Amigo networks:

- enable a containment relationship, and
- are *minimally* described as providing a set of operations,

service models from the aforementioned protocols directly map to the Amigo service model. However, constraining service search using properties, as in SLP, is not possible. We consider that is this not a major issue for the Amigo networks because more advanced service discovery and selection may be performed with Amigo-aware services, and the attributes/properties introduced by SLP lack precise semantics and standard definition, although some attempts have been made for specific service types (e.g., printer).

Once the address of a service instance matching a service request is discovered, the service may be either accessed using the access protocol tightly coupled with the service discovery protocol (e.g., UPnP-SSDP-SOAP, JINI-RMI) or using some legacy middleware interaction protocol (e.g., SLP-SOAP, SLP-RMI). In the former case, the same service model is used for both service discovery and access. In the latter case, the service models for service discovery and access may differ. However, the service models used for service access by the legacy interaction protocols listed in Table 3-1, on which we concentrate in a first step, obviously map directly to the above Amigo service model.

Following the above discussion, syntactic interoperability may be achieved among networked middleware-layer interoperable services within Amigo networks, by providing standard ontologies of service interfaces according to the service model defined above. Two approaches are possible and may actually be combined to define those ontologies, either using the Amigo declarative language for service description, which was introduced in the previous chapter, or using the description language of some legacy middleware. We will undertake a pragmatic approach in the time frame of the Amigo project, i.e., we will use

existing standard interfaces like the ones of the UPnP forum whenever available and introduce new standard interfaces described using the Amigo declarative language, when required by the application prototypes. Those standard interfaces may then be automatically translated into middleware-specific interface descriptions, using the middleware service description language, thanks to the mapping of respective type systems.

3.3 Networked Amigo-aware services

Amigo-aware services are developed using all the advanced features of the Amigo middleware, i.e., they are described using the language introduced in the previous chapter and are implemented on top of the Amigo base middleware. Interfacing with the Amigo base middleware is addressed in the following section, while Section 3.3.2 discusses semantic interoperability that is achieved among networked Amigo-aware services.

3.3.1 Interfacing with the Amigo base middleware

In the same way as for other networked services, Amigo-aware services use APIs of legacy middleware for accessing to functionalities of the middleware core, i.e., syntactic service discovery and service interaction. Then, relevant APIs for the development of Amigo-aware services in the course of the Amigo project are those listed in Table 3-2.

Amigo-aware services may further exploit advanced functionalities embedded in the Amigo middleware, relating to:

- Legacy middleware services like transaction processing,
- Intelligent user services investigated in WP4, and
- Base middleware functions investigated in WP3, which include various functionalities relevant to Aml systems, like security and privacy, in addition to the ones of the interoperable middleware core.

Relevant APIs for legacy middleware services may be found in the literature, while APIs for Amigo middleware services will be defined in the course of the Amigo project, as part of WP3 and WP4 work.

3.3.2 Achieving semantic interoperability

Two types of services of the Amigo base middleware are key to achieving semantic interoperability among Amigo-aware services, i.e., those dedicated to (semantic) service interoperability and Amigo-aware service discovery. The former relates to semantic service description and matching, as introduced in the previous chapter. The latter relates to *enhanced service discovery* exploiting semantic service description, further enriched with QoS and context information for service discovery (see Section 3.2 and Figure 3.4 of Deliverable D2.1).

Two approaches may be considered for the development of the Amigo solution to enhanced service discovery:

- developing a new service discovery protocol to be integrated in the middleware core,
- developing Amigo-aware service discovery as a middleware service on top of a legacy middleware core.

As suggested above, we adopt the latter approach. Basically, the former approach has the advantage of being potentially more efficient, in particular not paying the cost of service access in service discovery. On the other hand, this approach does not enable to benefit from the various legacy service discovery protocols that are already in place and optimized for the targeted networking environment. Also, this would lead to duplicate discovery processes in the

case of service discovery protocols that are tightly integrated with the middleware access protocol, like the UPnP middleware. Then, as a first approach, we will develop the Amigo-aware service discovery solution as an Amigo-aware service on top of legacy middleware core. Such an approach was in particular experimented in the Ozone project, with the WSAMI³ middleware for Aml, which introduces a middleware service for service discovery that builds on top of SLP. Performance evaluation showed that the cost of service discovery is, in this case, comparable to that of service access and efficiency then depends on the ratio between service access and service discovery.

The Amigo-aware service discovery process is depicted in Figure 3-4. Basically, the Amigo service for enhanced service discovery builds on top of a legacy middleware core, which is in particular exploited for discovering peer services for Amigo-aware service discovery in the network (Step 1 in the figure). Then, both semantic and syntactic service discovery may be performed. The former relies on the distributed service discovery protocol performed by the Amigo-aware service discovery (Step 2 in the figure). The latter relies on service discovery using the legacy middleware (Step 3 in the figure), translating the Amigo semantic service specification in a more primitive standard service interface, based on standard ontologies for services. Note that the latter step enables both an Amigo-aware service to be discovered by a networked legacy service (using the push scheme on the Amigo-aware device) and a networked legacy service to be discovered by an Amigo-aware service (using either the push scheme on the legacy device or the pull scheme on the Amigo-aware device).

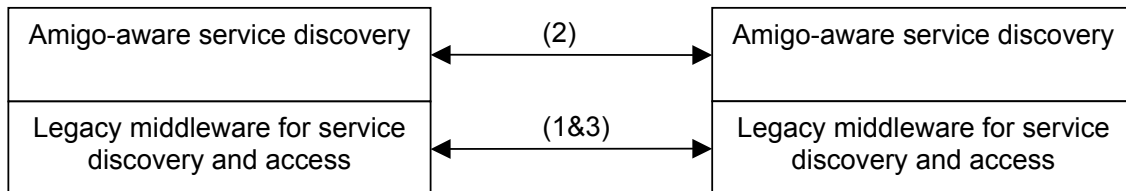


Figure 3-4: Amigo-aware service discovery

Detailed design of the Amigo-aware service discovery will be undertaken in the next project phase, addressing in particular:

- Integration of tool for efficient (both in resource and time dimensions) on-line semantic reasoning on semantic matching,
- Definition of the service discovery protocol, whether pull-based, push-based or both, whether centralized, semi-distributed or fully distributed.
- Dealing with QoS and context-awareness.

Also, we have mentioned that the Amigo-aware service discovery will build on a legacy middleware core without specifying which middleware among, e.g., those listed in Table 3-1. Our view is that the developer should be able to use the legacy middleware technology that is the most appropriate, according to his/her skill and main target application domain. This would further allow refining the Amigo-aware service discovery into domain-specific service discovery. This suggests not fixing a priori the underlying legacy middleware core for Amigo-aware service discovery. This issue is going to be investigated further in the next phase of the project. In addition, it is important to note that while the underlying legacy middleware

³ <http://www-rocq.inria.fr/arles/download/ozone/index.htm>

technology may be left open from a design perspective, the software prototypes implementing Amigo-aware service discovery that will be developed in the time frame of the project will be for a limited set of legacy middleware cores to be specified in accordance with, in particular, WP5-WP7 legacy software platforms.

3.4 Discussion

Building upon Amigo solutions to application-layer and middleware-layer interoperability that were introduced in Deliverable D2.1 and that are further refined in this deliverable, this chapter has investigated service discovery and access in the Amigo, open networked home environment, which may integrate:

- *legacy services* that are totally Amigo-unaware and do not use any of the Amigo solutions to interoperability;
- *middleware-layer interoperable services* that integrate Amigo middleware-layer interoperability solutions, i.e., the Amigo interoperable middleware core whose detailed design is addressed in the next chapter, and
- *Amigo-aware services* that are built on top of the Amigo middleware, which integrates solutions to application-layer and middleware-layer interoperability.

Then, depending on the services, interoperability levels may be:

- *network-dependent* if involved services are all legacy,
- *syntactic* if there are middleware-layer interoperable services involved, and
- *semantic* if all the services involved are Amigo-aware.

Syntactic interoperability requires defining ontologies of standard service interfaces, while semantic interoperability exploits semantic service description and matching introduced in the previous chapter.

Key feature of the Amigo middleware with respect to service discovery and access in the open Amigo network is that it allows effectively integrating legacy services without requiring any change to them, while enabling discovery of, and access to, services based on heterogeneous middleware technologies. In addition, advanced service discovery and access, exploiting semantic knowledge about the services' functional and non-functional (QoS- and context-related) properties may be performed, thanks to the Amigo application-layer interoperability methods and advanced middleware functions. A key challenge for the Amigo middleware is then to provide such features at low cost in terms of resource usage and further offering satisfying response time to end-users. The next chapter investigates this issue, introducing the detailed design of the Amigo interoperable middleware core and early performance results based on first prototype implementation.

4 Amigo interoperable middleware core

The Amigo interoperable middleware aims at enabling ambient intelligence for the networked home environment by addressing the integration of devices and related application services available within the networked home system (i.e., devices from the Consumer Electronics (CE), home automation, mobile and PC domains). The Amigo interoperable middleware architecture is specifically designed to realize an open networked home system that dynamically integrates heterogeneous devices as they join the network.

In this chapter, we present the design and implementation of the Amigo interoperable middleware *core*, which comprises essential middleware functions, such as service discovery, service interaction and QoS support, building upon the Amigo abstract middleware architecture presented in Deliverable D2.1 [Amigo-D2.1]. More specifically, we elaborate detailed design and first prototype implementation for certain functionalities of the middleware core, while for other functionalities, we provide at this stage early design and implementation, or a refined architecture with respect to the abstract middleware architecture. All parts of the middleware core will be further elaborated in next deliverables towards the final prototype implementation.

The main functionality offered by the Amigo interoperable middleware core is interoperability between services that employ different discovery (e.g., SSDP, Jini, SLP) and interaction (e.g., RMI, SOAP) protocols. Our solution to middleware interoperability integrates two tightly related subsystems: the service discovery interoperability subsystem (elaborated in Section 4.1) and the service interaction interoperability subsystem (elaborated in Section 4.2). Further, we introduce a programming and deployment framework for the Amigo system, which enables modular development and configurability of Amigo middleware components as well as application services (Section 4.3). We finally elaborate solutions to domotic domain interoperability (Section 4.4) and CE domain interoperability (Section 4.5).

4.1 Service discovery interoperability (SDI)

This section presents our detailed design and implementation of service discovery interoperability. Initially, we recall from Deliverable D2.1 the design principles on which SDI is based (Section 4.1.1). Then, we present the detailed design and first prototype implementation of SDI, using the UML language (Section 4.1.2). We finally provide evaluation of our prototype in terms of implementation footprint and performance (Section 4.1.3).

4.1.1 Design principles

The majority of service discovery protocols (SDPs) support the concepts of *client*, *service* and *repository*. In order to find needed services, clients may perform two types of request: *unicast* or *multicast*. The former implies the use of a repository, equivalent to a centralized lookup service, which aggregates information on services from services' advertisements. The latter is used when either the repository's location is not known or there exists no repository in the environment. Similarly, services may announce themselves with either unicast or multicast advertisement, depending on whether a repository is present or not. Two SDP models are then identified, irrespectively of the repository's existence: the passive discovery model and the active discovery model. When a repository exists in an environment, the main challenge for clients and services is to discover the location of the repository, which acts as a mandatory intermediary between clients and services. In this context, when using the passive discovery model, clients and services are passively listening on a multicast group address specific to the SDP used, and are waiting for a repository multicast advertisement. On the contrary, in an active discovery model, clients and services send multicast requests to discover a repository, which sends back a unicast response to the requester to indicate its presence. In a "repository-less" context, a passive discovery model means that the client is listening on a

multicast group address that is specific to the SDP used to discover services. Obviously, the latter periodically send out a multicast announcement of their existence to the same multicast group address. In contrast, with a repository-less active discovery model, the roles are exchanged. Thereby, clients perform periodically multicast requests to discover needed services, and the latter are listening to these requests. Furthermore, services reply unicast responses directly to the requester only if they match the requested service.

Basically, all SDPs use a multicast group address and a UDP/TCP port that must and have been assigned by the Internet Assigned Numbers Authority (IANA). Thus, assigned ports and multicast group addresses are reserved, without any ambiguity, to only one type of use. Furthermore, it is important to notice that an entity may subscribe to several multicast groups, and so may be simultaneously a member of different types of multicast groups. These two characteristics only are sufficient to provide simple but efficient environmental SDP detection. To achieve SDP detection, a component called *monitor component* embeds these two major behaviors: (i) the ability to subscribe to several SDP multicast groups, irrespectively of their technologies; and (ii) the ability to listen to all their respective ports. The monitor component is able to determine the current SDP(s) that is (are) used in the environment upon the arrival of the data at the monitored ports without doing any computation, data interpretation or data transformation. It does not matter what SDP model is used (i.e., active or passive), as the detection is not based on the data content but on the data arrival at the specified UDP/TCP ports inside the corresponding groups.

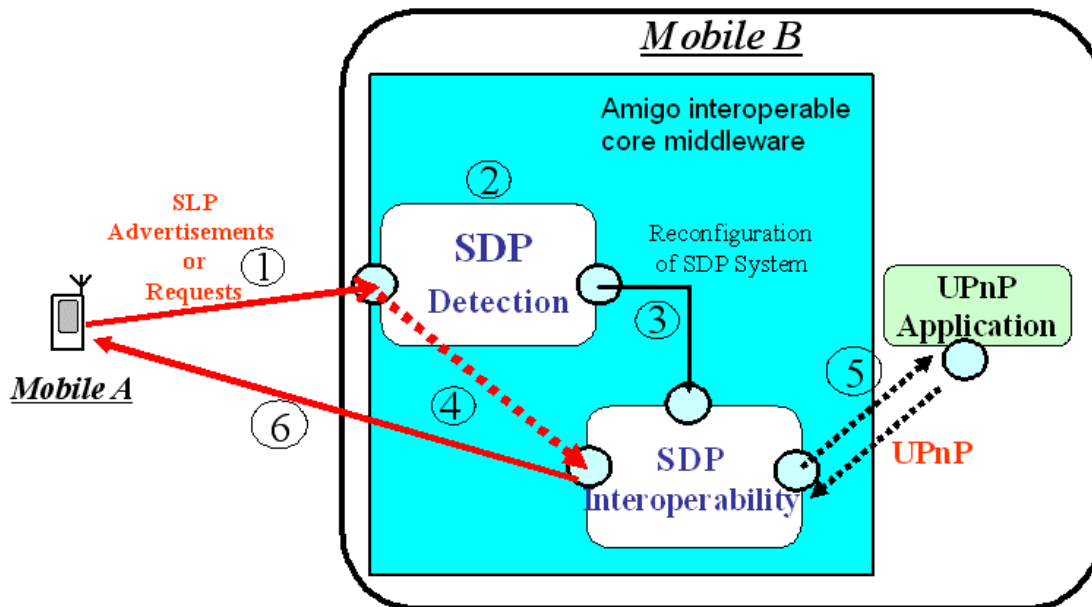


Figure 4-1: SDP detection and interoperability mechanisms

SDP detection is just a first step towards SDP interoperability and represents a primary component. The main issue is still unresolved: the incoming raw data flow that comes to the monitor component needs to be correctly interpreted to deliver the services' descriptions to the application components. To support such functionality, we introduce an SDP interoperability component based on event-based parsing concepts. Figure 4-1 shows the relation between the SDP detection component and the SDP interoperability component of the Amigo interoperable middleware core in an environment with an SLP (Service Location Protocol) based application and a UPnP (Universal Plug and Play) based application.

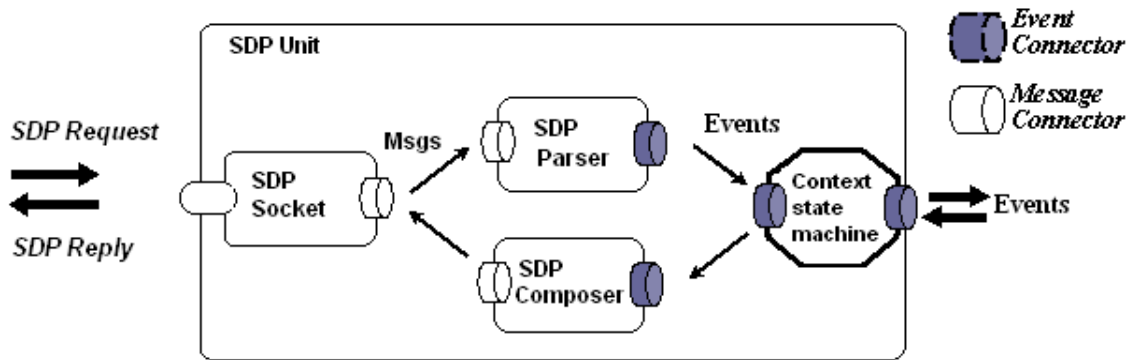


Figure 4-2: SDP Unit configuration

Specifically, upon the arrival of raw data at monitored ports, the monitor component detects the SDP that is used, and sends a corresponding event to the appropriate *parser* to successfully transform the raw data flow into a series of events. The parser extracts semantic concepts as events from syntactic details of the SDP detected. Then, the generated events are delivered to the local components' *composers*. The communication between the parser and the composer does not depend on any syntactic detail of any protocol. They communicate at semantic level through the use of events. Parsers and composers are dedicated to a specific SDP protocol. Then, to support more than one SDP, several parsers and composers must be embedded into the system. Parsers and composers are further decoupled from the transport protocol used for the receipt/sending of messages by enabling various types of *socket* components, which may further be changed at runtime.

In general, SDP functions are complex distributed processes that require coordination between the actors of the specific service discovery function. This may be realized by embedding the parser and composer within a *unit* that runs coordination processes associated with the functions of the given SDP. The unit is further self-configurable in that it manages the evolution of its configuration as needed by the SDP specifics and the evolution of the environment. The behavior of the unit may easily be specified using finite state machines.

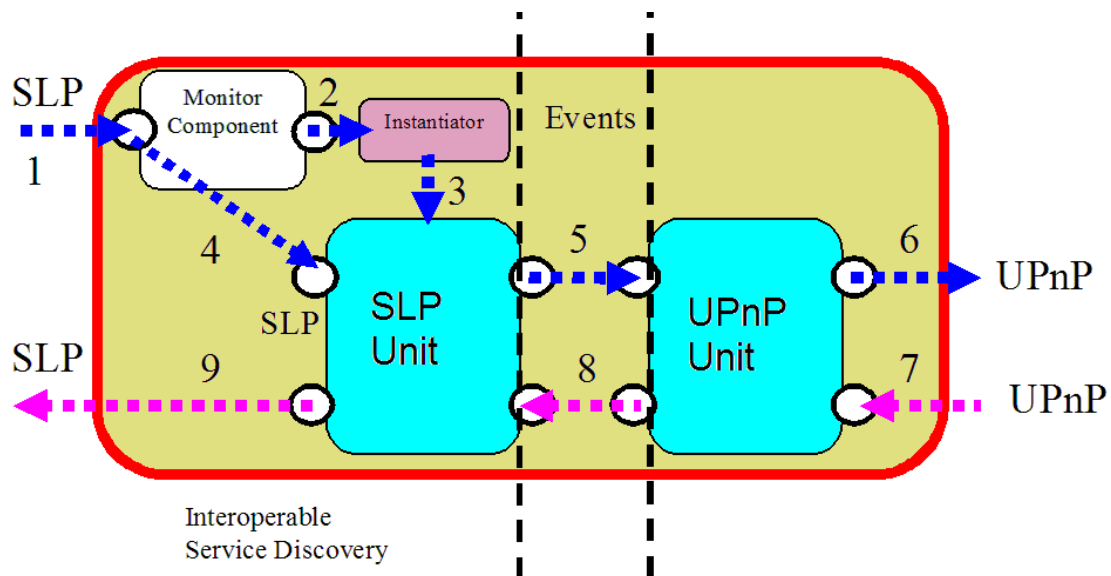


Figure 4-3: SDP interoperability mechanisms

The unit is in charge of dispatching event notifications to its registered listeners through event connectors. Message-oriented connectors enable the interaction among components that are not event-oriented. Parsers are endowed with both event- and message-oriented connectors. Thus, inside the units, parsers' input ports are bound to message-oriented connectors, whereas parsers' output ports are bound to an event connector controlled through the unit's state machine. Conversely, composers' output ports are bound to message-oriented connectors, whereas composers' input ports are bound to the unit's event bus (see Figure 4-2).

SDP interoperability is achieved through the correct composition of a number of units. As depicted in Figure 4-3, the translation from SLP to UPnP discovery corresponds to the composition of an SLP unit with a UPnP unit. At this level, units are only considered as computational elements that transform messages to events and vice versa. The units' internal mechanisms are totally hidden.

4.1.2 Detailed design and implementation

In this section, we present our detailed design and implementation of the service discovery interoperability. In Section 4.1.2.1, we provide an overview of the design and the relations among the classes that constitute the prototype, using UML class diagram descriptions. Then, in Section 4.1.2.2, we detail our implementation using UML class and sequence diagrams: the internals of each class and its relations with the other classes are surveyed to provide a better understanding of the technical details of the prototype implementation.

4.1.2.1 Overview

Figure 4-4 depicts the components that realize the service discovery interoperability mechanisms.

SdpSocket is the abstract class that provides the common interface and the basic functionalities for all the sockets supported by the middleware. *TcpSocket*, *UdpSocket*, *UdpMulticastSocket* and *HttpSocket* inherit from this base class and respectively provide implementations of sockets for TCP, unicast UDP, multicast UDP and HTTP. *SdpParser* is the abstract class that provides the common interface and the basic functionalities for all the parsers supported by the middleware. *SlpParser*, *SSDPParser*, *HttpParser* inherit from this base class and respectively provide an implementation of SLP, SSDP and HTTP parsing. *ServiceDescrParser* and *DeviceDescrParser* are respectively used to parse UPnP device and service descriptions. *SdpComposer* is the abstract class that provides the common interface and the basic functionalities for all the composers supported by the middleware. *SlpComposer* and *UpnpComposer* inherit from this base class and respectively provide an implementation of a composer for SLP and UPnP.

Units are represented by the abstract class *SdpUnit*. The abstract class *SdpUnitFactory* is in charge of the creation of instances of *SdpUnit* objects. Two units are currently supported by the middleware: *SlpUnit* and *UpnpUnit*, which are respectively the implementations of SLP and UPnP protocols. For each of them, we provide a class that inherits from the *SdpUnitFactory* and is in charge of the creation of instances of the unit (*SlpUnitFactory* and *UpnpUnitFactory*). The context state machine shown in Figure 4-2 is represented by *SlpUnitContext* and *UpnpUnitContext*.

The definition of a unit must include at least one component of each type (socket, parser and composer). Thus, *SdpUnit* contains a *SdpSocket* object list, a *SdpParser* object list and a *SdpComposer* object list. *SlpUnit* and *UpnpUnit* will define the parsers, composers and sockets used by instantiating the values of these three lists.

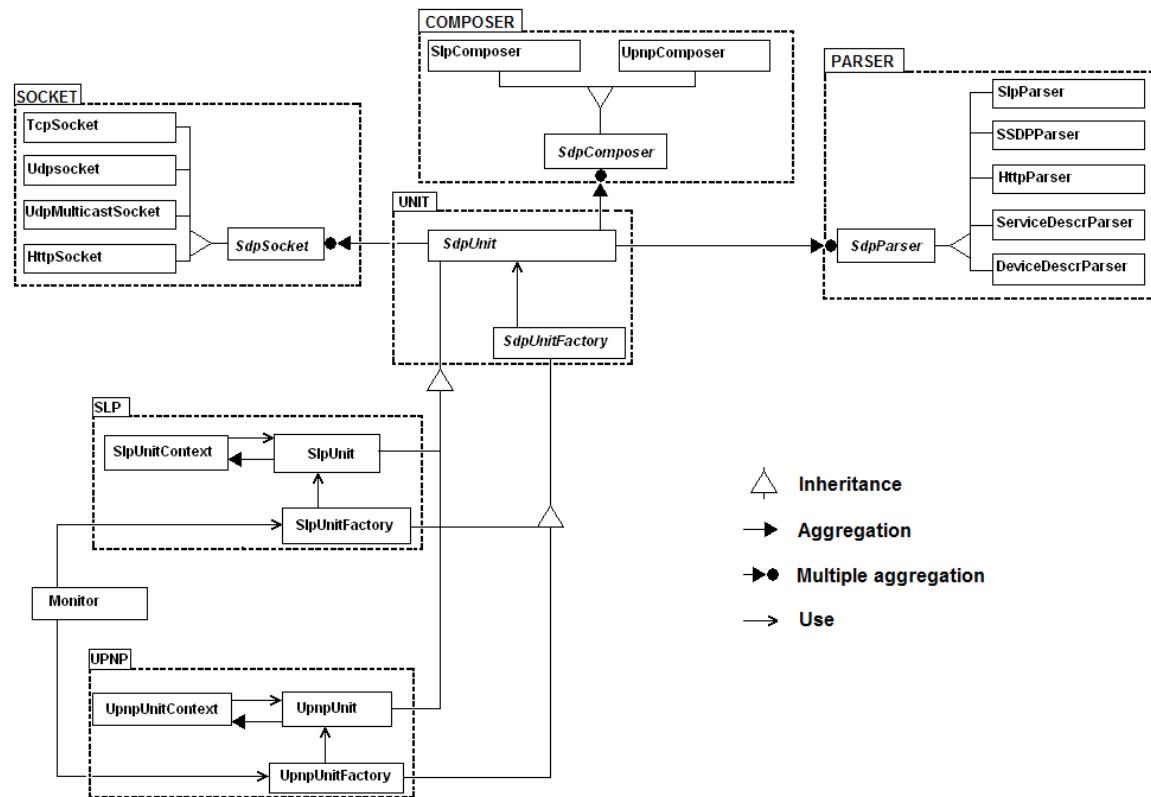


Figure 4-4: Detailed design of service discovery interoperability

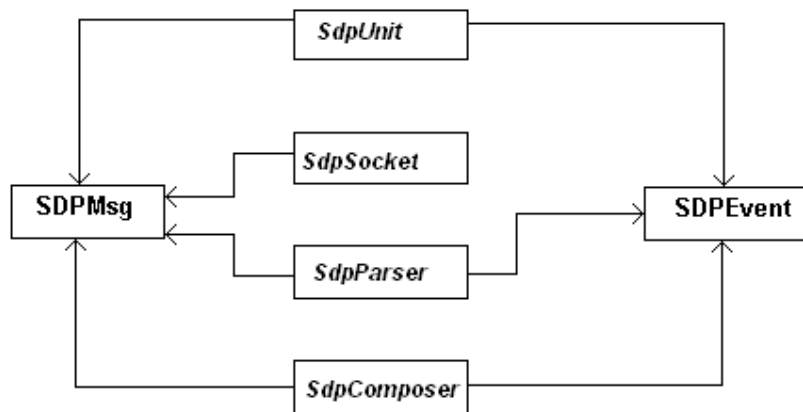


Figure 4-5: Detailed design of event and message communication

Events and messages for internal communication inside the unit are respectively implemented by *SDPEvent* and *SDPMsg* (Figure 4-5). The implementation of connectors for transmission of events and messages is detailed in the following section.

4.1.2.2 Detailed description

This section describes the different components and mechanisms that implement the service discovery interoperability: the monitor, the event and message connectors, the sockets, the parsers, the composers and the units. The details of the implementation of each component and mechanism are described using UML class and sequence diagrams.

Monitor

When the system is initialized, it is the responsibility of the *Monitor* to set up the correct configuration of unit composition to achieve service discovery interoperability. In the current prototype of service discovery interoperability, we provide a *SlpUnit* and a *UpnpUnit*. When the system is initialized, the Monitor defines the configuration of unit composition for service discovery interoperability as shown in Figure 4-6. In Figure 4-6a, a SLP unit listens to SLP network messages and generates events that will be dispatched to a UPnP unit. The latter will translate these events into UPnP messages delivered to the UPnP application. In the opposite direction, the UPnP unit will translate UPnP messages coming from the UPnP application into related events that will be redirected to the SLP unit to create an SLP reply. In Figure 4-6b, the Monitor defines another configuration, with a UPnP unit listening to UPnP network messages and an SLP unit delivering to an SLP application.

As Figure 4-6 shows, the SLP unit includes a SLP composer, a SLP parser and two UDP sockets, one for unicast and one for multicast, while the UPnP unit includes a UPnP composer, four different parsers (*SSDPParser*, *HttpParser*, *ServiceDescrParser*, *DeviceDescrParser*) and four different sockets (*TcpSocket*, *UdpSocket*, *UdpMulticastSocket*, *HttpSocket*).

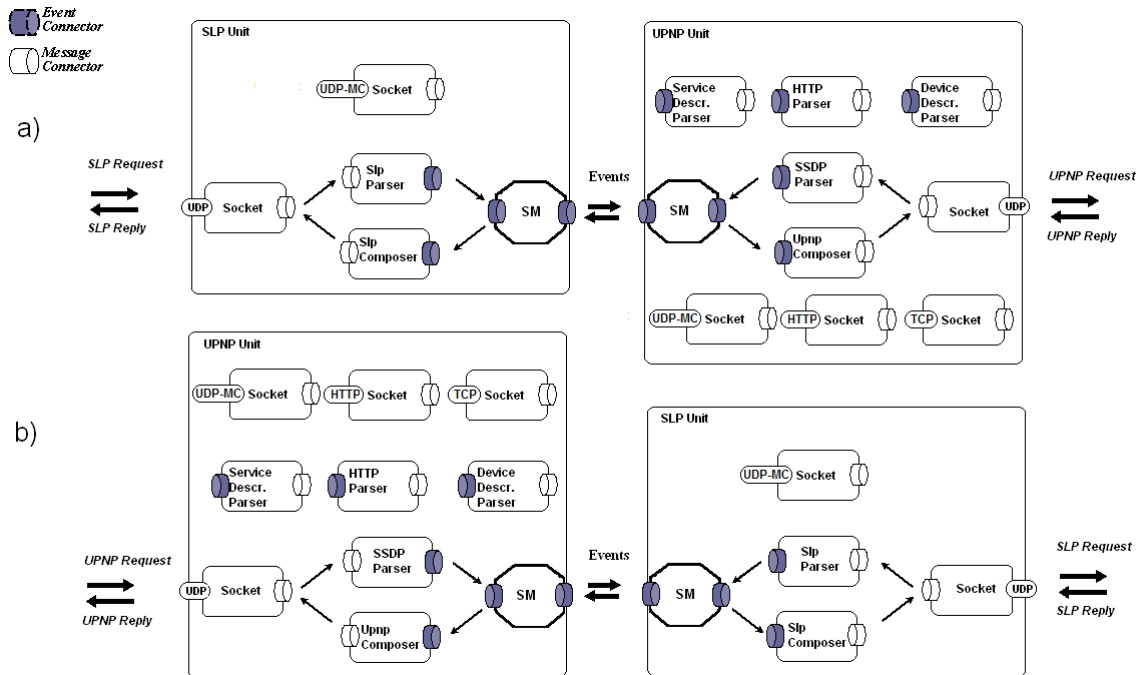


Figure 4-6: *SdpUnit* composition configuration decided by Monitor for service discovery interoperability

Figure 4-7 depicts the diagram of the classes involved in the initialization process coordinated by the Monitor leading to the composition configuration depicted in Figure 4-6. The base class *SdpUnit* represents an SDP unit, and each specific SDP is implemented by a concrete subclass of *SdpUnit* (e.g., *UpnpUnit*, *SlpUnit*). *SdpUnitFactory* is the class factory able to create instances of *SdpUnit*. The abstract method *createNewSdpUnit* of *SdpUnitFactory* must be implemented by its concrete subclasses, and, when executed, it creates a corresponding concrete subclass of *SdpUnit*.

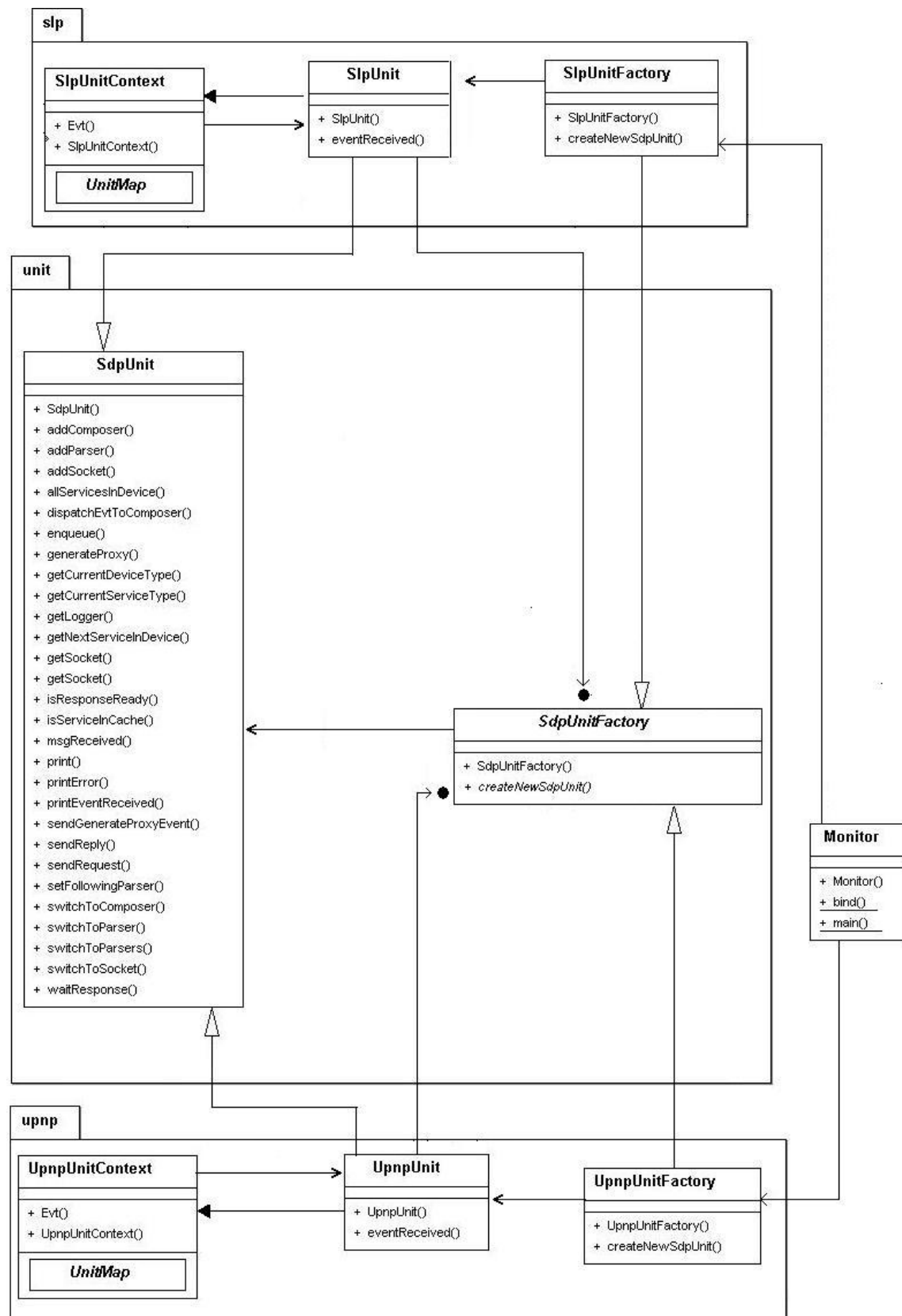


Figure 4-7: SDP Units, SDP Unit Factories and Monitor class diagram

The sequence diagram in Figure 4-8 shows the initialization process aiming to configure the unit composition for interoperability between SLP and UPnP depicted in Figure 4-6a. For each supported discovery protocol other than SLP, a new *SdpUnitFactory* object is created (actually, in the example, one *UpnpUnitFactory* object is created) and passed as a parameter to the new generated class *SlpUnit*. The *SlpUnit* object, when created, will store a reference to the *SdpUnitFactory* object and make a call to the method *createNewSdpUnit* of all the *SdpUnitFactory* objects received as constructor parameters. This method, as already seen above, will create an instance of the corresponding SDP unit (in the example, *UpnpUnitFactory* will create an instance of *UpnpUnit*). The unit generated will be registered as an event listener of the *SlpUnit* and vice versa (in the example, the *UpnpUnit* will be registered as an event listener of the *SlpUnit* and vice versa). More details about the event connector model will be provided in the following. The initialization process to configure the unit composition for interoperability between UPnP and SLP depicted in Figure 4-6b is similar; only the roles of SLP and UPnP classes are inverted.

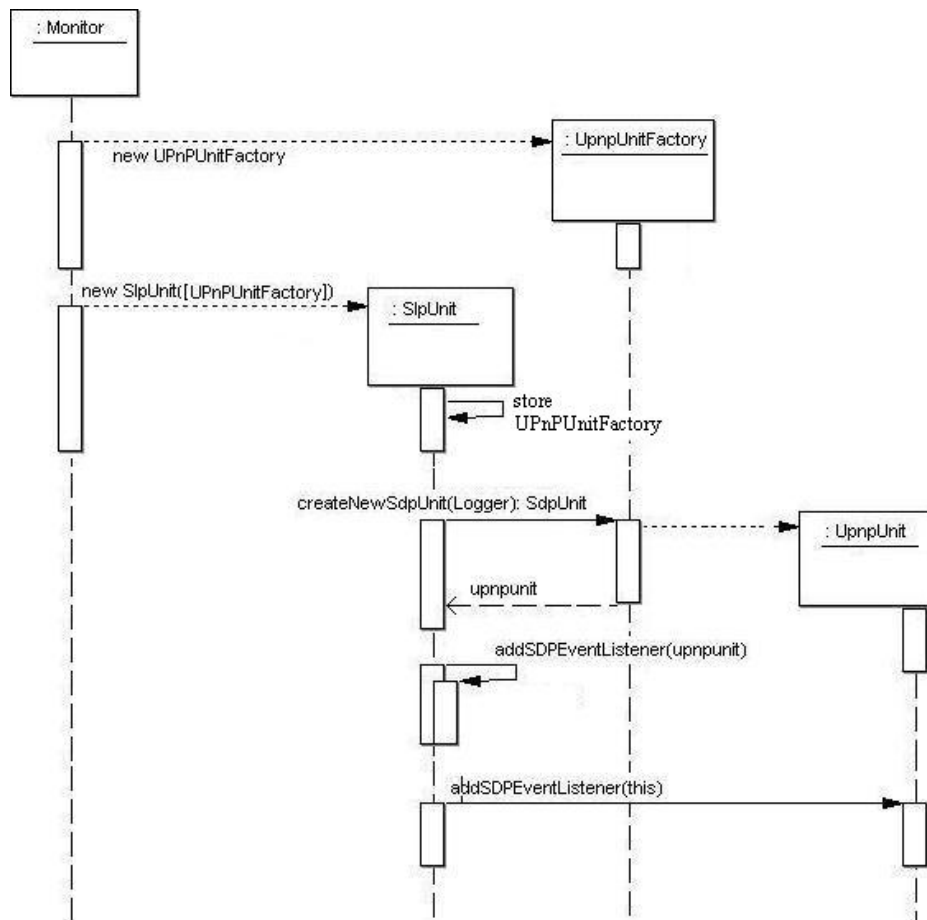


Figure 4-8: UML sequence diagram of the initialization process

After this initialization step, the service discovery interoperability system is configured. Each unit has been connected through event connectors to other units, and is able to handle the network messages received by the socket component that is listening on the assigned address and port. As Figure 4-8 shows, this configuration consists in associating a list of *SdpUnitFactory* objects to the unit, one *SdpUnitFactory* object for each protocol supported other than the one concerned by the configuration. For each new message received on the socket, the correct unit composition is instantiated using the list of *SdpUnitFactory* associated.

For example, if an SLP message is received, the composition of Figure 4-6a will be instantiated by *SlpUnit* by creating an instance of *UpnpUnit* using the *createNewSdpUnit* method of *UpnpUnitFactory*. On the other hand, if the message is a UPnP message, the composition of Figure 4-6b will be instantiated.

Event and Message connectors

As seen in Section 4.1.2.1, the internal communication inside each unit is based on events and messages. Each component that wants to communicate through events provides an event connector; if it wants to communicate through messages, it provides a message connector.

Events that are carried through the connector are represented by the class *SDPEvent* (Figure 4-9). An *SDPEvent* object consists of two parts: an event type and a payload containing some data. In Deliverable D2.1 [Amigo-D2.1], we defined the minimal set of event types that is common to all SDPs and the sets of events that are specific to some SDPs. The minimal set includes, for example, the events that may be generated by all SDP components in order to notify their listeners of their internal states. It also includes the events that describe the common functions provided by the different SDPs: service search request, service search response, service advertisements and the type of the service searched.

All the possible values of the event type field are listed in the *Event* class in Figure 4-9. The data payload carried by an event can contain different types of values: numbers, strings, arrays of bytes or a *java.lang.Object*. The types of these values depend on the event type. For example an event of type *Event.SDP_REQ_SERVICE_TYPE* will carry a value of type *String* containing the required service identifier.

For communication with external components, a unit makes use of sockets supporting the different network transport protocols. We introduce message connectors to separate the internal component communication from network protocols. Thanks to message connectors that carry protocol-independent messages, the unit components are not aware of the protocol used to send or receive a message. Messages carried through the connectors are represented by the class *SDPMsg* (Figure 4-9). Each *SDPMsg* object contains the message source and destination IP addresses and ports and the content (an array of bytes) received (or to be sent) on the socket (can be either on UDP or TCP or HTTP connection).

Figure 4-10 represents the complete class diagram hierarchy used in the middleware to support the event and message connector model. Every component (socket, parser and composer) is a publisher and/or a subscriber of events and/or messages.

The base interfaces *EvtPublisher*, *EvtSubscriber* are respectively provided to handle the dispatching and reception of events. A class that wants to listen to events and be notified when an event *SDPEvent* is raised will implement the interface *EvtSubscriber*, while a class that wants to produce events and notify event listeners will implement the interface *EvtPublished*. The interface *EvtPublisher* provides a method to add and a method to remove a listener for events on the publisher class; and a method *dispatchEvents* to send an event to all the listeners registered on the publisher. An event subscriber must implement the interface *EvtSubscriber*. The subscriber class must use the *EvtPublisher*'s method *addSDPEventListener* to subscribe for events on a specific event publisher class. When the publisher raises an event, the subscriber method *eventReceived* will be invoked with the event description associated with the method call.

For *SDPMsg* messages, the middleware makes use of the same solution as for events: the two interfaces *MsgPublisher* and *MsgSubscriber* are the corresponding interfaces that have the same functions as the two interfaces for events explained above.

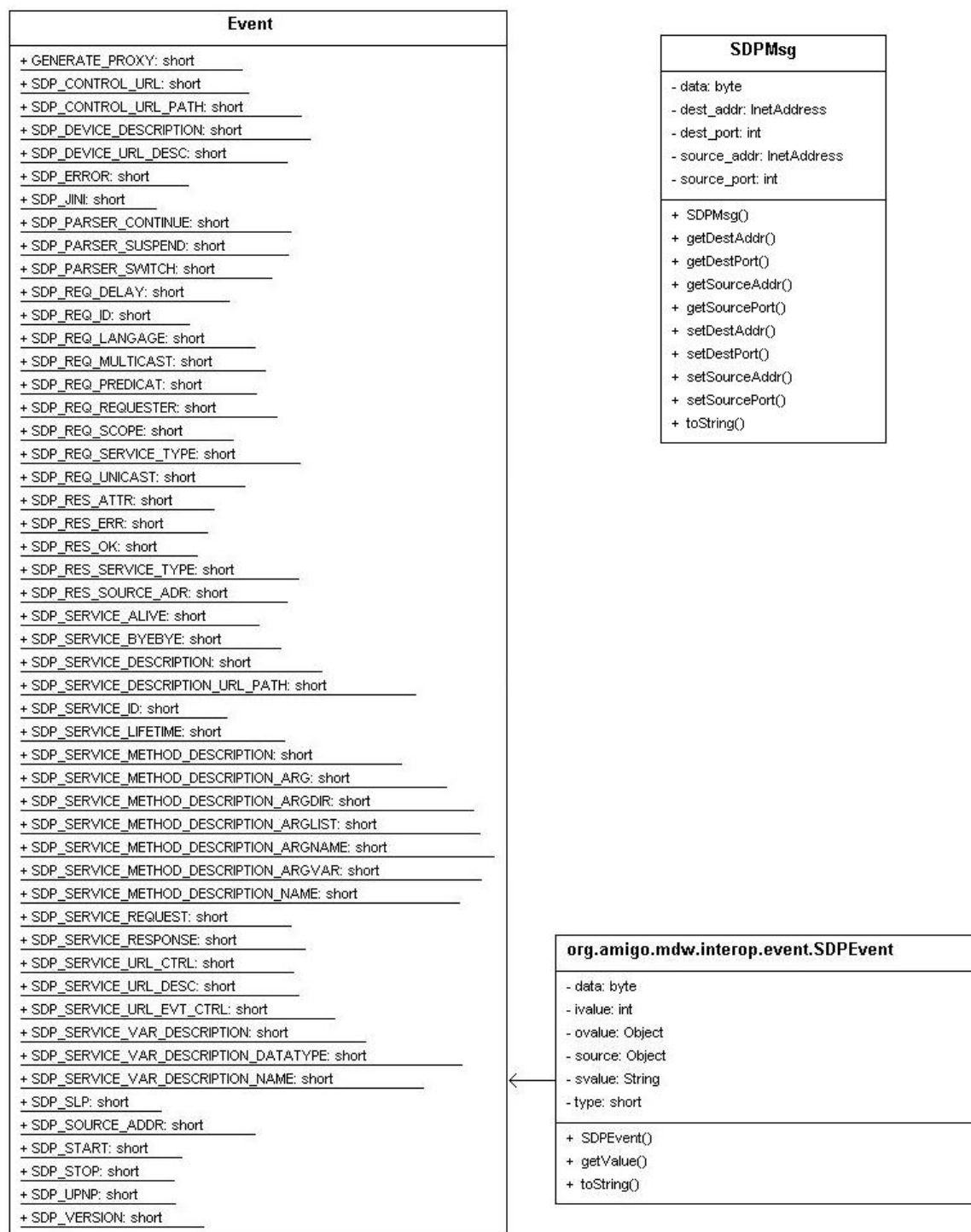


Figure 4-9: SDPMsg, SDPEvent and list of Event types

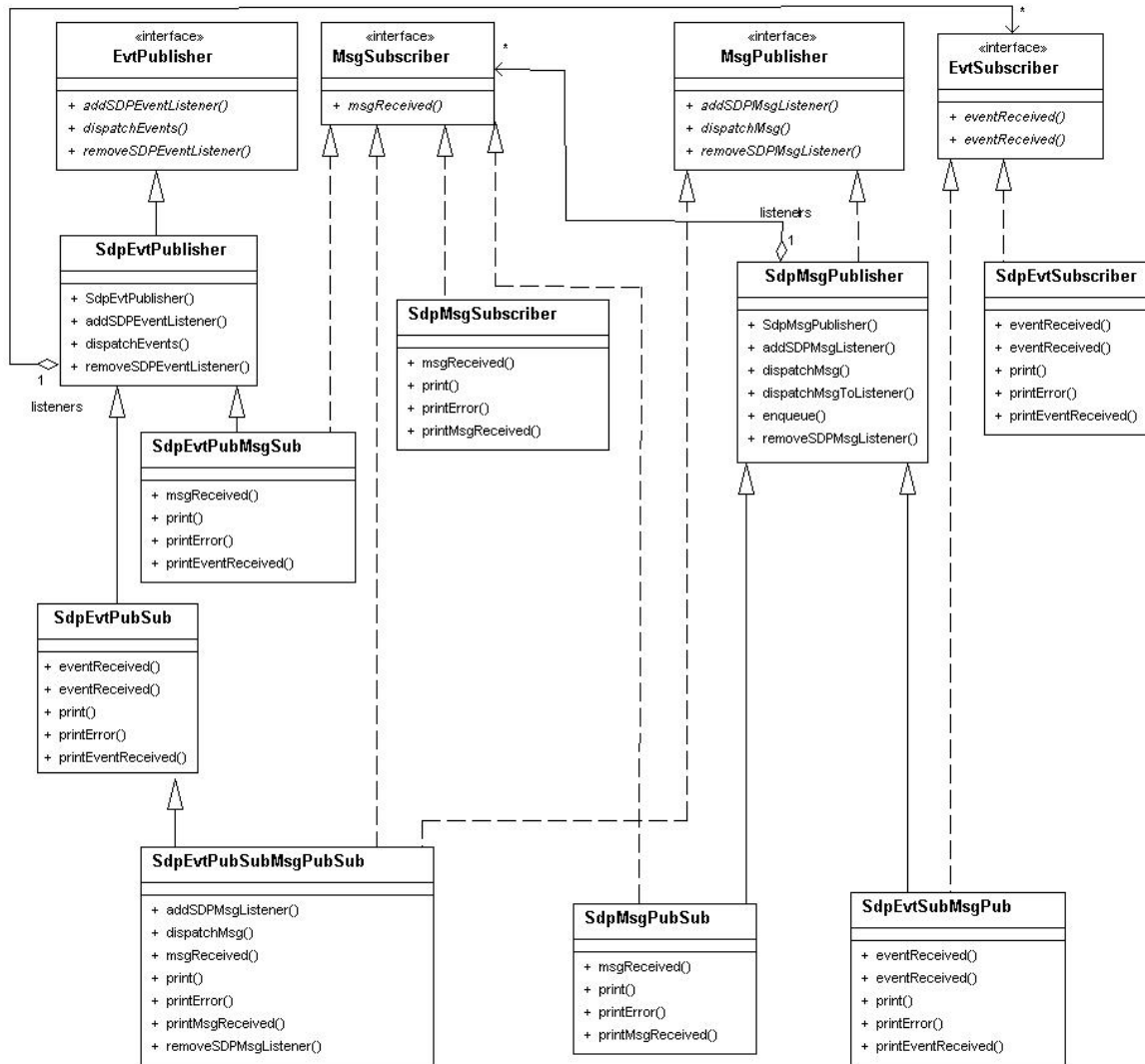


Figure 4-10: Event and message connectors class diagram

The introduced base interfaces are implemented by the classes of Figure 4-10 to provide the base mechanisms for publishing/subscribing for events and for messages. The classes *SdpEvtPublisher*, *SdpEvtSubscriber*, *SdpMsgPublisher* and *SdpMsgSubscriber* provide the basic implementation of the above described interfaces. Some other classes of Figure 4-10 implement more than one interface; in this case, we have a class (for example *SdpEvtSubMsgPub*) that is a message publisher and an event subscriber (for example *SdpEvtSubMsgPub*).

In Figure 4-11, the sequence diagram details the event notification between an *EvtPublisher* and all its *EvtSubscribers* that receive the notification of the event: an object makes a call to the *dispatchEvent* method of the *EvtPublisher* class with the *SDPEvent* event describing the event to be notified. The publisher has a list of its listeners *EvtSubscriber*. For example, in Figure 4-11, a class inheriting from the *SdpEvtPubSub* class is in the list of listeners. For each of its listeners, the publisher calls the method *eventReceived* (containing the *SDPEvent* event definition as a parameter). It is up to the class implementing the interface *EvtSubscriber* to provide an implementation of this method and to handle the received events. For example, in Figure 4-11, the class inheriting from *SdpEvtPubSub* will further dispatch the received

SDPEvent to its own listeners by calling its own method *dispatchEvent* that will execute the same actions as above described for *EvtPublisher*.

A similar mechanism is implemented for messages: the only difference is that the class *SdpMsgPubSub* does not re-dispatch the *SDPMsg* received on its method *msgReceived*.

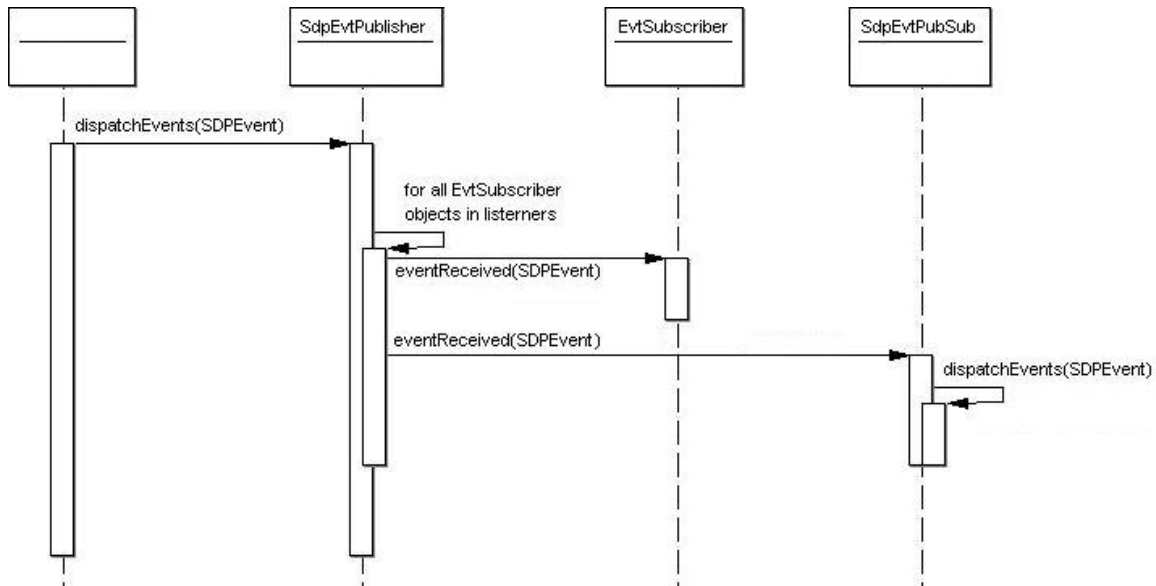


Figure 4-11: Event publish/subscribe sequence diagram

Socket

Sockets are in charge of sending and receiving messages using a specific transport protocol. As we currently assume all-IP networks, we define the corresponding types of socket components: multicast sockets and unicast sockets, where the latter may be either connection-oriented or connection-less. Socket components offer flexibility enabling the implementation of system components in a way that is independent of the underlying transport.

The base abstract class for sockets is *SdpSocket* (Figure 4-12) and it is a subclass of *SdpMsgPubSub*: both its inbound and outbound connections are message connectors. The socket components provided by the middleware are: *UdpSocket* to send and receive on a unicast UDP socket, *TcpSocket* to send and receive on a unicast TCP socket, *UdpMulticastSocket* to send and receive on a multicast UDP socket, and finally *HttpSocket* that is able to handle HTTP messages (including the HTTP header and all the possible options and features offered by the HTTP protocol) on an HTTP connection.

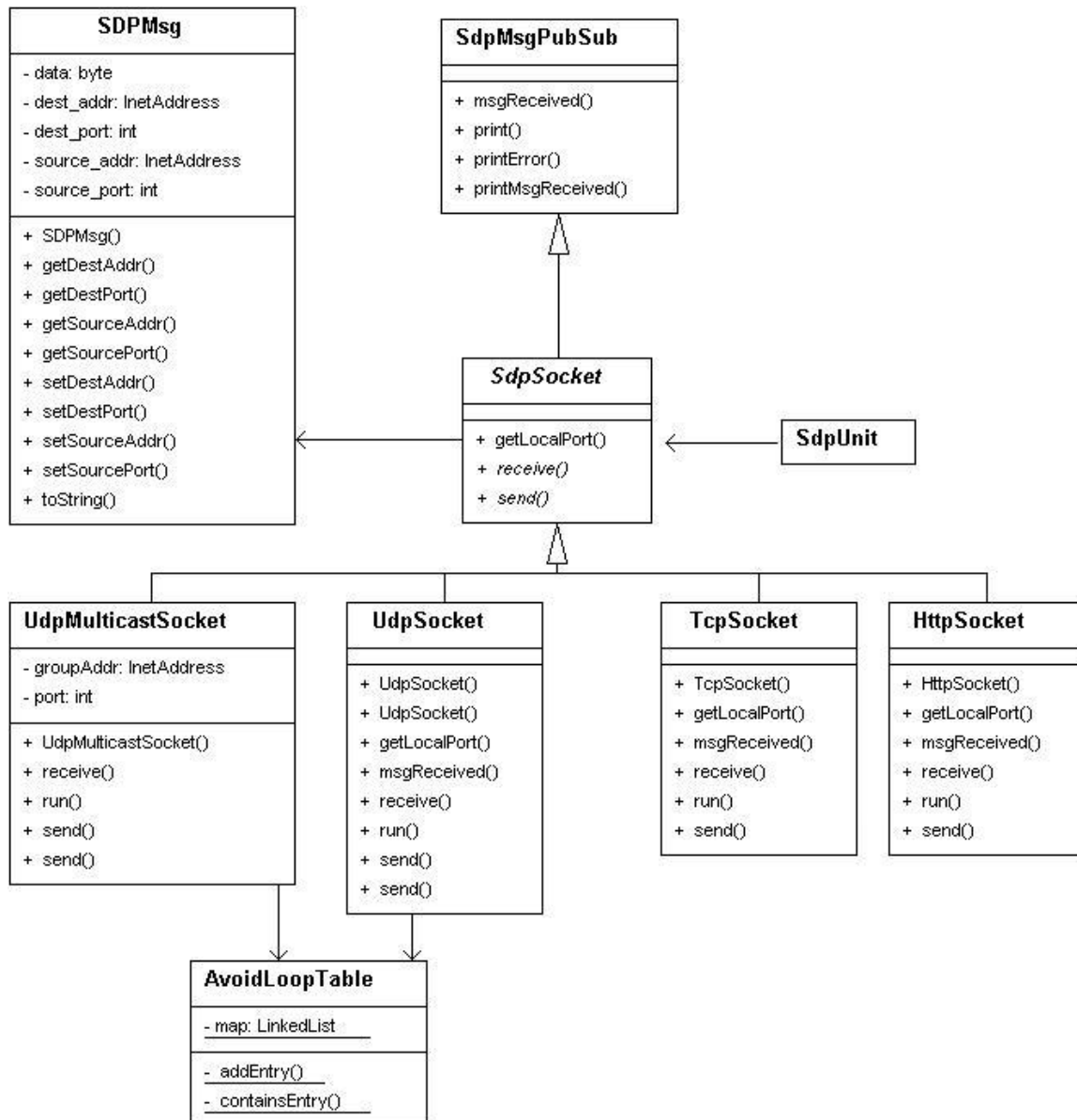


Figure 4-12: Class diagram of the sockets provided by the middleware

When activated (by invoking the *receive* method), the socket starts listening to messages on the assigned address and port. For each new message received (the reception of the message is implemented in accordance with the corresponding transport protocol specification), the socket creates an *SDPMsg* object (adding the content of the message together with the information about source and destination address and port), and delivers it to its listeners using its *dispatchMsg* method. On the other hand, when the unit wants to send a message through the socket, it must invoke its method *msgReceived(SDPMsg)*. The *SDPMsg* object contains all the information required by the *send* method for building the message in the context of the transport protocol supported by the socket: the content of the message, the destination address and port.

We add an *AvoidLoopTable* object shared by all UDP sockets to prevent the middleware from entering in a message handling loop. When a UDP multicast message is sent, its source and destination addresses and ports are registered in *AvoidLoopTable*; when a UDP multicast

message is received by the middleware, it must be discarded if its source address and port appear in *AvoidLoopTable*. If these messages were not filtered, the middleware would receive and process its own messages.

Parser

The role of a parser component is to wait for messages, parse their content and generate a sequence of semantic events in conformance with the implemented protocol specification. Parsers are decoupled from the transport protocol by means of socket components, which may be changed at runtime. As a result, the same HTTP parser instance may parse streams from a UDP datagram, generated by either a unicast or multicast request, as well as from a TCP stream.

The base abstract class for parser components is *SdpParser*, and it is a subclass of *SdpEvtPubMsgSub*: its inbound connector is a message connector, while its outbound connector is an event connector. As *SdpParser* is a subclass of *SdpEvtPubMsgSub*, it provides a method *msgReceived(SDPMsg)* that will be invoked by an *MsgPublisher* object. The method *msgReceived* creates an *InputStream* object containing the *SDPMsg* content and passes it to the abstract *parse* method to be parsed. Each parser that inherits from *SdpParser* must implement the parsing algorithm in its *parse* method. The role of the *parse* method is to extract semantic concepts as *SDPEvent* objects from the *SDPMsg* received, and to deliver each *SDPEvent* object to the *Parser* listeners using the event notification mechanism (*SdpParser* is a subclass of *SdpEvtPublisher*).

Figure 4-13 represents the set of parsers available in the middleware. The *SLPParser* class implements the SLP protocol: each *SDPMsg* object received by the *parse* method must contain an SLP message, and the *SDPEvent* events generated respect the SLP protocol specification⁴. In the same way, the *SSDPParser* class implements the SSDP protocol (UPnP protocol for device and service discovery), and the *HTTPParser* class implements the HTTP protocol. As an HTTP message usually contains a payload that can be from another protocol (for example XML), the parser generates the *Event.SDP_PARSER_SWITCH* event after having finished parsing the HTTP Header. It is up to the unit state machine to set the next parser that will continue parsing the *SDPMsg* and will generate the corresponding events. For the HTTP parser, we make use of the Cybergarage UPnP implementation⁵. Our class *HTTPPacket2* in Figure 4-13 is a subclass of *HTTPPacket*, a class from the Cybergarage implementation. We have fixed some bugs and handled the chunked option of HTTP protocol specifications not implemented in the Cybergarage library. The *DeviceDescrParser* implements a parser that is able to parse the UPnP device descriptions received as response to UPnP messages. The device description syntax is defined by the UPnP specifications⁶. This parser takes into account this syntax and generates two events for each service contained in the device: the first notifies the service description URL (*Event.SDP_SERVICE_DESCRIPTION_URL_PATH*) and the second the control URL of the service (*Event.SDP_CONTROL_URL_PATH*). The *ServiceDescrParser* implements a parser that is able to parse the UPnP service definition taking into account the UPnP specifications⁷ for services; this parser generates events to notify the description of the methods of the service.

⁴ <http://www.faqs.org/rfcs/rfc2608.html>

⁵ <http://www.cybergarage.org/net/upnp/java/index.html>

⁶ http://www.upnp.org/download/UPnPDA10_20000613.htm

⁷ http://www.upnp.org/download/UPnPDA10_20000613.htm

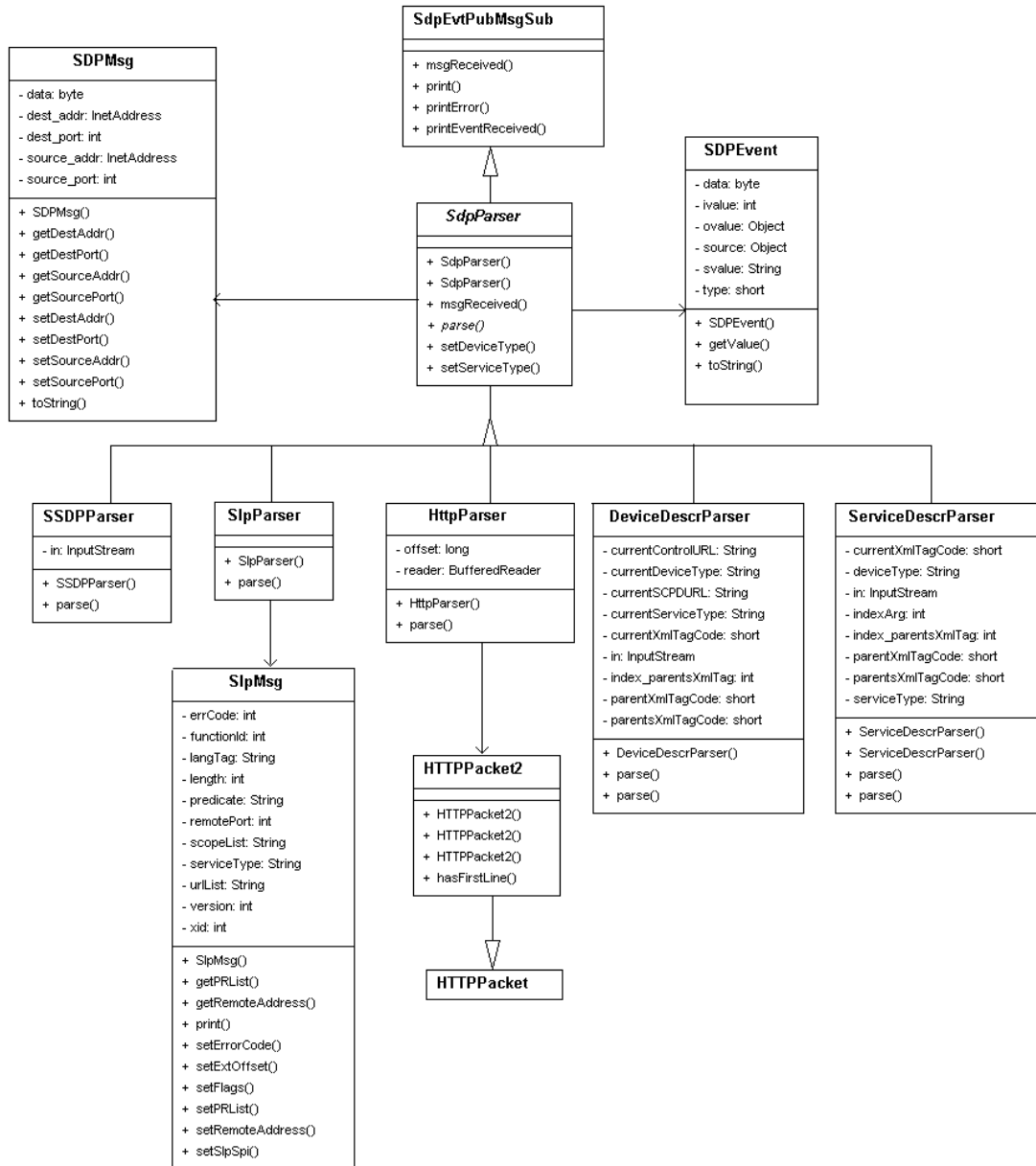


Figure 4-13: Class diagram of the parsers provided by the middleware

All the SDP interoperability components and connectors presented in this section (e.g., parsers, sockets) are not necessarily specific to an SDP; they may be reused in various units, even if not related to the same SDP. For instance the HTTP parser developed for UPnP may be reused for another SDP unit.

Composer

The role of a composer is to generate well-formed messages in conformance with the specific protocol implemented and at the same time coherent with the semantic events received on its event connector from the event notification mechanism.

As a composer is a subclass of *SdpComposer* and *SdpEvtSubMsgPub*, it is a listener of *SDPEvent* objects that are received by the *eventReceived* method and a producer of *SDPMsg* objects. The set of events that a composer is able to handle depends on the protocol implemented; if an event received is not supported, it will be simply discarded. For example, a subset of events generated by a UPnP parser is successfully understood by a SLP composer, whereas specific UPnP events, due to UPnP functionalities that SLP does not provide, are simply discarded by the SLP composer, as they are unknown. When all the events required for the creation of a message in the specific protocol have been received, the corresponding *SDPMsg* object is generated and published with the method *dispatchMsg*. The composer's listener (a socket) will receive and handle the message.

As Figure 4-14 shows, the middleware provides an implementation for SLP and UPnP protocol composers. The *SdpComposer*'s methods *sendReply*, *sendRequest*, *isResponseReady*, *getCurrentDevice*, *getCurrentService*, *isServiceInCache*, *getNextServiceInDevice* and *allServicesInDevice* defined by the abstract class are used by the unit state machine to ask the composer to execute an operation or to obtain information about the current state of the composer. Each concrete composer subclass must provide an implementation of these functionalities.

Unit

A unit implements event-based interoperability for a specific SDP by translating messages of the specific SDP to and from semantic events associated with service discovery; and by implementing coordination processes over the events according to the behavior prescribed by the SDP specification. Units are composed and communicate through their event connectors, whereas they use their socket components to interact with components that are outside the SDP interoperability system. Within a unit, coordination and composition rules among embedded SDP components are specialized with respect to a given SDP according to the unit state.

The class diagram in Figure 4-15 shows the relation between the classes implementing the units and their components (sockets, parsers and composers). The base abstract class *SdpUnit* implements all the basic functionalities required by every unit to manage the components and to allow the unit state machine to coordinate the components. It also identifies the methods that each concrete unit class must implement according to the SDP specification. The unit is the control point of all the components and coordinates the internal interaction between the composers, the parsers and the sockets. *SdpUnit* contains a series of data structures storing its dynamic component configuration that can change over time. These data structures keep a reference to the list of supported sockets, parsers and composers in relation to the protocol specification (for example the UPnP unit needs TCP and UDP sockets, SSDP and HTTP parsers, and a UPnP composer). The methods provided to add one of these components are the following: *addParser*, *addComposer* and *addSocket*. Even if the unit can have multiple references to sockets, parsers and composers, at a specific instance it has only one active socket and composer and one or more active parsers (ordered in a sequential list), depending on the action that the unit is executing. The methods *switchToParser*, *switchToParsers*, *switchToComposer* and *switchToSocket* are used to select the active components. The method *setFollowingParser* is used when a list of active parsers has been set and the parsing of the message requires changing the current parser and switching to the following one in the list.

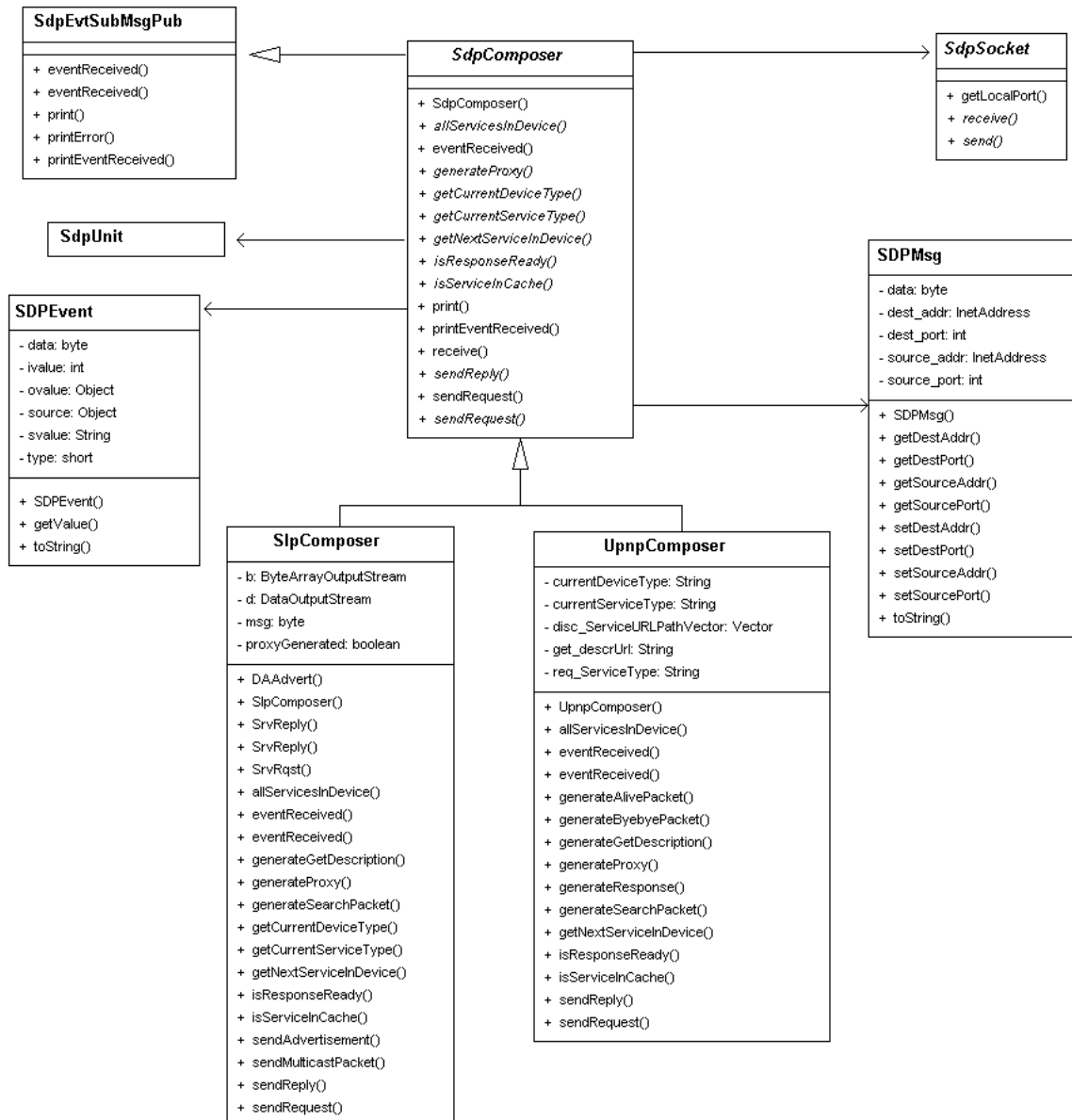


Figure 4-14: Class diagram of the composers provided by the middleware

Since the unit superclass *SdpUnit* is a subclass of *SdpEvtPubSub*, it can produce and receive *SDPEvents*. The *SDPEvents* produced by the unit are dispatched through the method *dispatchEvents*; the unit's listeners that will receive the events have been defined in the initialization process executed by the Monitor to set up the unit composition configuration (see above). On the other hand, the *SDPEvents* are received by the *eventReceived* method. This method must be overloaded by every subclass of *SdpUnit*; its role is to forward the *SDPEvent* to the associated state machine class in order to be handled in the right way. Every unit has a state machine associated with it. For example *UpnpUnitContext* and *SlpUnitContext* are respectively the classes implementing the state machines of *UpnpUnit* and *SlpUnit* units. The state machine engine will take into account the current state and the received event before executing the related instructions and moving to the corresponding state; it may additionally react to the event with actions modifying the unit's current configuration. For example, the state machine class can modify the current socket and parser in order to set up the unit for

receiving a message on the right socket address and port and for parsing it with the right parser.

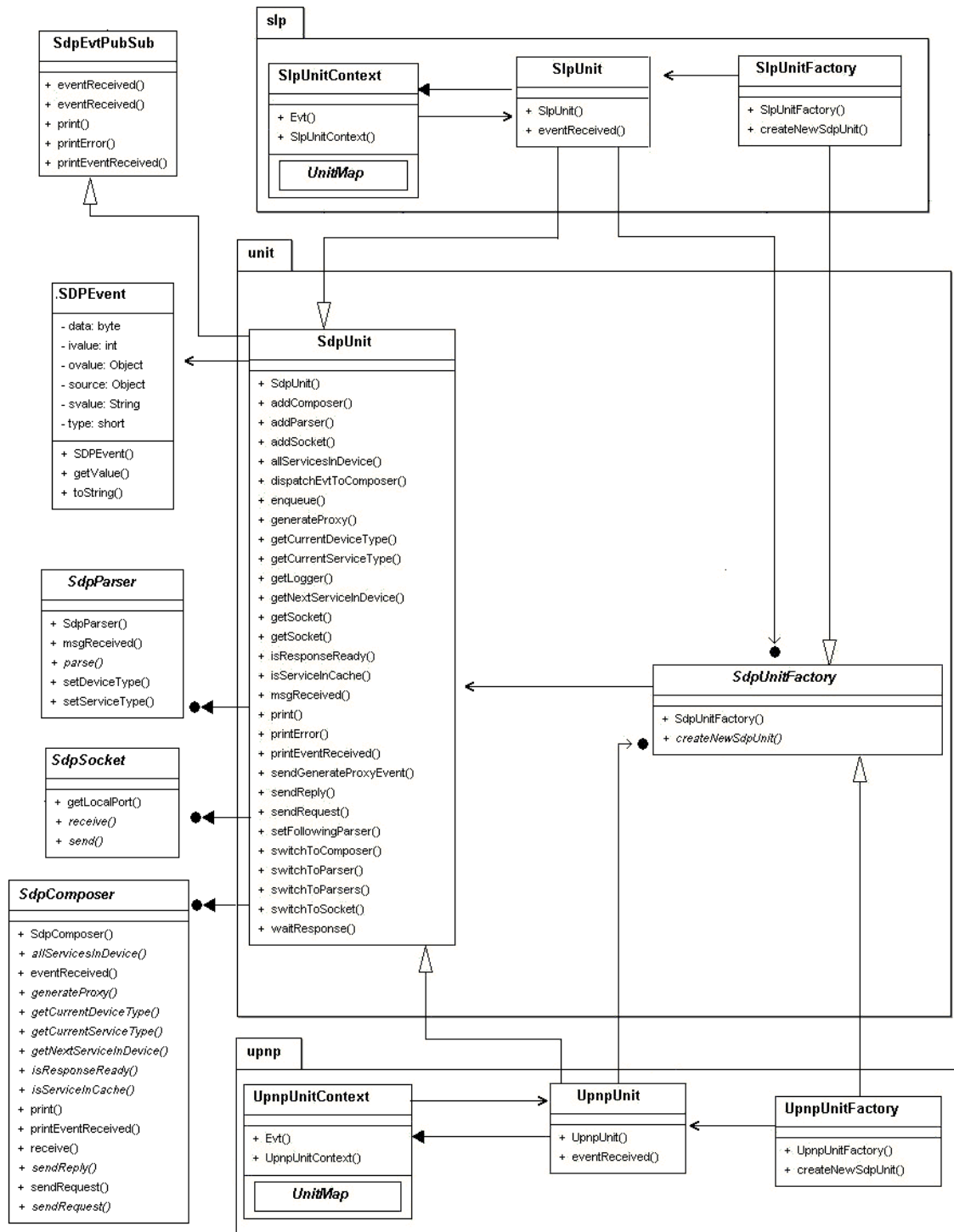


Figure 4-15: Unit class diagram

For the definition of the state machine of each SDP unit, the middleware makes use of the tool *SMC - The State Machine Compiler*⁸ and its associated language and tools. The SM definition in the SMC language must take into account the SDP specification. The activation/deactivation and dynamic composition of unit components is defined by the SM to accomplish the operation that must be provided at a certain moment by the unit. The state machine file defined with the SMC language is compiled with the compiler tool provided by SMC into a java class (*SlpUnitContext* and *UPnPUnitContext*).

Figure 4-16 provides an extract from the SM file of *UpnpUnit*. SMC defines an initial *IDLE* state; for each state defined for the SDP unit, there are two entries that are matched respectively when the SM enters the state (*Entry*) and before leaving the state (*Exit*). The occurrence of an event may cause transitions between states if the event matches both the event type defined in the transition and the optional conditions of the transition.

In the example in Figure 4-16, the state *UPNP_WAIT_CREATE_UPNP_RESPONSE* represents the behavior of the UPnP unit when it is waiting for all the necessary information to build a UPnP reply message. The transition from state *UPNP_WAIT_CREATE_UPNP_RESPONSE* to state *IDLE* (that represents the initial state) will take place when an *SDP_STOP* event is received (generated when the parsing of a message is finished) and the condition *isResponseReady()==true* is true. Further, the state machine will execute the list of actions *switchToParser("SSDP")*, *switchToSocket("UDP")*, *dispatchEvtToComposer(e)* and *sendReply()*. After these actions, the composer will send the UPnP reply; the parser and socket are then set up to receive the next UPnP message.

The unit's methods *sendReply*, *sendRequest*, *waitResponse* are used by the unit state machine to control the unit and its active components: to require sending a reply message, to send a request message, and to wait for a response message, respectively.

```
IDLE
Exit { print("UNIT_UPNP:IDLE:EXIT"); }
Entry { print("UNIT_UPNP:IDLE:ENTRY"); }
{
    Evt(e:SDPEvent)
    [e.getType() == Event.SDP_START]
    START
    {}

    Evt(e:SDPEvent)
    nil
    {
        printEventReceived(e);
        print("UNIT_UPNP:Enqueue");
        enqueue(e);
    }
}

UPNP_WAIT_CREATE_UPNP_RESPONSE
```

⁸ <http://smc.sourceforge.net>

```

Exit {print("UNIT_UPNP:UPNP_WAIT_CREATE_UPNP_RESPONSE:EXIT"); }
Entry { print("UNIT_UPNP:UPNP_WAIT_CREATE_UPNP_RESPONSE:ENTRY"); }
{

    Evt(e: SDPEvent)
    [e.getType() == Event.SDP_STOP && isResponseReady() == true]
    IDLE
    {

        switchToParser("SSDP");
        switchToSocket("UDP");
        dispatchEvtToComposer(e);
        sendReply();

    }

    Evt(e:SDPEvent)
    nil
    {

        dispatchEvtToComposer(e);

    }

}

```

Figure 4-16: Extract from UPnP unit state machine

4.1.3 Evaluation of implementation and performance

We have implemented a first prototype of the service discovery interoperability subsystem of the Amigo interoperable middleware core. Currently, it includes a UPnP unit and a SLP unit. Although our prototype is not yet optimised, it is robust enough for assessing the performance of our approach in different use cases. The following discusses key elements of the prototype. We first discuss its small code footprint requirements compared to existing solutions. We then evaluate its performance by comparing supported response times with native service discovery.

<i>Amigo middleware size requirements</i>				
	Size (KB)	Classes	NCSS	Overhead
Core framework	44	15	789	-
UPnP Unit	125	18	1515	-
SLP Unit	49	6	606	-
Total	218	39	2910	-
<i>SDP library size requirements</i>				
OpenSlp Library	126	21	1361	-
Cyberlink UPnP	372	107	5887	-
Total	498	128	7248	-
<i>Size requirements to provide interoperability with and without Amigo middleware</i>				
SLP & UPnP Library + SLP & UPnP clients	514	-	-	-

UPnP client & Library + Amigo middleware	598	-	-	14%
SLP client & Library + Amigo middleware	352	-	-	-31.5%

Table 4-1: Footprint requirements in KBytes for known libraries and the Amigo middleware core

The prototype is implemented in Java to take advantage of cross platform portability. We are, in particular, able to deploy our solution on any mobile device that embeds J2ME⁹, which provides a Java virtual machine customized for devices with limited resources.

In Table 4-1, we compare the footprint requirements of the Amigo middleware core with the ones of common open-source libraries like *OpenSlp*¹⁰ and *Cyberlink* for Java¹¹. The overall Amigo middleware consists of 39 Java classes and 2910 lines of Non-Commented Source Statement Classes (NCSS). The overall system size is 218 Kbytes. This includes 125Kbytes for the UPnP Unit and 49Kbytes for the SLP Unit. To be interoperable, nodes running UPnP (resp. SLP) applications need to host a native UPnP (resp. SLP) library plus the Amigo middleware. This is to contrast with an interoperable device that is not equipped with our interoperable system, which needs: (i) to host both the full UPnP stack and the SLP library, and (ii) some engineering effort to develop and host an additional SLP (resp. UPnP) client that is equivalent in terms of functionalities to the UPnP (resp. SLP) client.

As further depicted in Table 4-1, the size requirements of a middleware that needs to be interoperable and does include the Amigo interoperable middleware core (includes both full SLP and UPnP) is 514Kbytes when hosting one simple service. In contrast, the size requirement for a middleware dedicated to UPnP (resp. SLP) equipped with the Amigo middleware is 598Kbytes (resp. 352Kbytes). Then, the size requirements increase proportionally with the number of hosted services. The size requirements of an interoperable middleware without the Amigo interoperable middleware core increase faster than the ones of a middleware equipped with the Amigo interoperable middleware core, because, for the former, each time we add a service, we have to add two implementations of the service (e.g., SLP service + UPnP service). Thus, the small size overhead introduced by the Amigo interoperable middleware core with UPnP applications disappears when the number of hosted services increases.

Further, a middleware that needs to host different services, in terms of both functionalities and SDP used, must have all the corresponding native libraries irrespectively of the use of Amigo middleware. However, in this case, the latter still provides efficient interoperability: it reduces drastically both the number of hosted services and, in the long term, the overall middleware size since we do not have to develop and deploy services for each existing SDP.

Experimental results

We evaluate the performance of our interoperability mechanisms by investigating the response time of the Amigo interoperable middleware core when enabling a client dedicated to one SDP to discover a service based on another SDP. Specifically, the experiments consider the case where a SLP (resp. UPnP) client searches a SLP (resp. UPnP) service. We then compare the native client waiting time to get an answer from a native service with its waiting time to get an

⁹ <http://java.sun.com/j2me/index.jsp>

¹⁰ <http://www.openslp.org/>

¹¹ <http://www.cybergarage.org/net/upnp/java/>

answer from an Amigo-interworked service. The impact of Amigo middleware on performance varies according to its location, either on the client or on the service side. Thus in the following, we consider the two cases. In addition, as interoperability is achieved without generating additional traffic, we have not evaluated the network bandwidth consumption. Indeed, the generated traffic is well known since we are neither providing a new service discovery protocol nor altering native protocols.

Although our solution is dedicated to various devices, including resource-constrained ones, all tests are performed on workstations equipped with 256Mbytes RAM on Intel PIV processor rated at 1.8GHz. In fact, currently, to the best of our knowledge, there does not exist any UPnP profile for J2ME devices in the open source community. Thus, the operating system, the Java virtual machine and the performance tools platform used are, respectively, Linux from Redhat Fedora Core 2, JDK1.4.2 from Sun, and the Hyades platform from the Eclipse Foundation. Moreover, the SLP (resp. UPnP) client and SLP (resp. UPnP) service are hosted on different hosts connected to a LAN at 10Mb/s. The SLP client and service are based on OpenSlp, whereas the UPnP client and service use Cyberlink for Java. The given measurements are in msec and are the median of 30 successful tests to avoid a mean skewed by a single high or low value.

	SLP -> SLP	UPnP -> UPnP
Median value (ms)	0.7	40

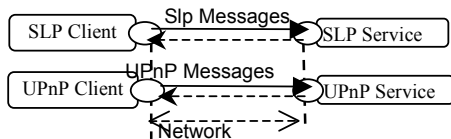
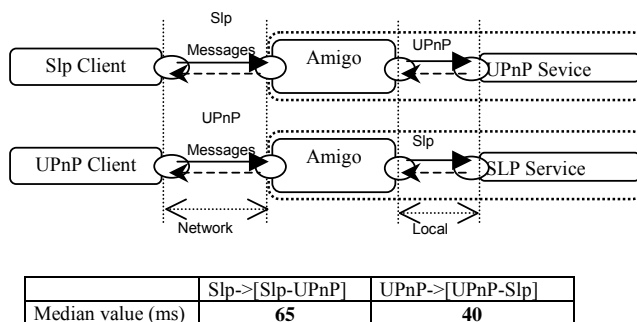


Figure 4-17: Native clients & services

In Figure 4-17, we first give the response time of a search request generated by a native client to get a successful answer from a native service: for SLP, we get 0.7 ms, whereas for UPnP, we get 40ms. It is clear that using SLP is much more efficient than UPnP, which is a higher-level protocol than SLP. These results are considered as references values to enable us to interpret the following results.



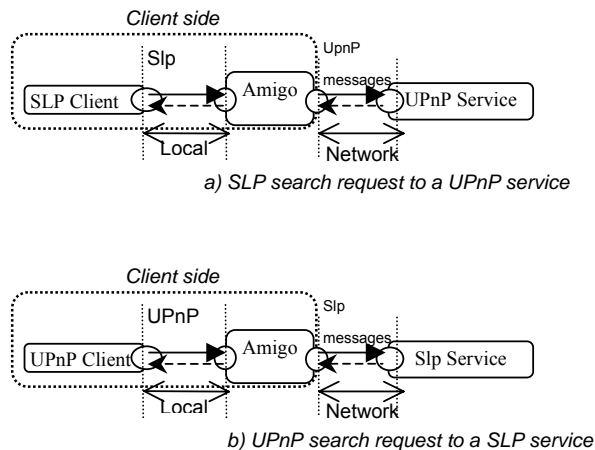
	Slp->[Slp-UPnP]	UPnP->[UPnP-Slp]
Median value (ms)	65	40

Figure 4-18: Performance with Amigo located on the service side

Consider now the case where Amigo middleware is located on the service side to enable the latter to be interoperable with any client independently of its SDP (Figure 4-18). In the context where the client is SLP and the service is UPnP, the client gets an answer in 65ms. The

translation between SLP and UPnP is not direct. For instance, UPnP and SLP search responses are semantically different: a SLP client expects a direct reference to interact with the service discovered, whereas a UPnP client expects a reference to a description file corresponding to the service found. Consequently, the Amigo middleware has translated the SLP request into two local UPnP requests to get the information that is necessary to generate on the network the corresponding SLP response. This means that the Amigo middleware has waited and parsed successively two UPnP responses, thus increasing the SLP responsiveness latency. On the service side, it is clear that the Amigo middleware simulates a UPnP client, and therefore we cannot interfere on the native time taken to get a UPnP response from the service. In this context, the Amigo middleware result is pretty good.

Still in Figure 4-18, when the client is UPnP and the service is SLP, the response time to get an answer is 40ms. In fact, it corresponds exactly to a search request generated on the network from a native UPnP client to a native UPnP service. On the service side, the response time to a SLP request is negligible as the latter is generated locally.



	[SLP-UPnP]->UPnP	[UPnP-SLP]->SLP
Median value (ms)	80	0.12

Figure 4-19: Performance with Amigo located on the client side

When the Amigo interoperable middleware core is located on the client side (Figure 4-19a), the latter becomes interoperable and can discover any service whatever its SDP. If the client is SLP and the service is UPnP, the SLP client gets the answer to its search request in 80ms. It corresponds globally to two native UPnP responses from a native UPnP service. This is obvious, since, as previously, the Amigo interoperable middleware core has translated the SLP request into two network UPnP requests to get the necessary information to generate locally the corresponding SLP response. Once again, the Amigo interoperable middleware core result is encouraging. It is important to note that compared to the case depicted in Figure 4-18, the response time is higher than previously, simply because the UPnP traffic goes across the network between the Amigo interoperable middleware core and the UPnP service, increasing by 15 ms the response time. In the same context, the high response time inherent to the UPnP protocol is confirmed, as a UPnP client gets a response from a SLP service in only 0.12ms (Figure 4-19b). This is due to the fact that, first, the UPnP traffic is local and, then, the only traffic that goes across the network is SLP, which is particularly fast. In addition, the necessary information to generate a search response for UPnP is tiny. We can consider this case as the best case.

The above results show that the Amigo interoperable middleware core is particularly efficient in providing interoperability in all possible contexts.

4.2 Service interaction interoperability (SII)

This section presents our early design and implementation of service interaction interoperability (SII). Initially, we recall from Deliverable D2.1 [Amigo-D2.1] the design principles on which SII is based (Section 4.2.1). Then, we present the early design and first prototype implementation of SII, using the UML language (Section 4.2.2).

4.2.1 Design principles

According to the service-oriented architectural style, interaction protocols identify two application components: a client and a service. The former requires and the latter provides some functionality. For a specific interaction the protocol identifies the client and the service; the client and service roles may be inversed in another interaction.

Practically, the service runs at an address that may be known by the client, either statically at design time or dynamically using some service discovery protocol. However, in both cases, knowledge of the service's address does not mean knowledge of the service's interaction protocol, although it may be assumed when known statically. More specifically, unlike the SDP detection mechanism, the interaction protocol detection cannot be simply based on the address of the interacting parties. Achieving interaction protocol interoperability further raises similar issues as for achieving SDP interoperability, i.e.: (i) dealing with the heterogeneity of service description, which relates to the use of diverse service interface definition languages for interaction (e.g., WSDL for SOAP, IDL for CORBA); and (ii) dealing with different interaction protocols.

Service-oriented computing allows several interaction paradigms between client and service. For example, interaction protocols may be RPC-based, message-oriented or event-based. As discussed in Deliverable D2.1, we focus on RPC-based interactions at a first stage.

To save the client code from dealing with the details of the service's reference, interface and interaction protocol, a component called stub is usually provided by the middleware, assuming knowledge of the interaction protocol on which the service is based. The client then calls methods on the client stub. The stub converts method calls into network protocol messages, and takes care of marshalling method arguments. If the service replies with a message to the client call, the stub unmarshals the results and performs a regular method return to the client application.

Interaction protocol interoperability is achieved using the same method as the one described in the previous section, i.e., it relies on event-based parsing (see Figure 4-20). Two major issues arise from event-based parsing to actually achieve interaction protocol interoperability:

- Mapping of service references between heterogeneous middleware platforms; and
- Identification of the incoming communication protocols, i.e., detection.

We enrich our solution with the facility of dynamic *stub* generation. Stubs are generated according to the client's required interface and to the service description. The stub generation is a two-step process. The step-zero takes place during the development of the client, and corresponds to the classical, static generation of the client-side part of the stub (see Step 0 below), using the client's required interface as input. The first runtime step corresponds to the discovery of a service matching the client's required interface. This further reveals the interaction protocol of the remote service, and may be considered as realization of direct conformance checking between the interaction protocols of the client and the service, from the standpoint of Chapter 2 (see Step 1 below). In the second step, the service's provided

interface will be used for the dynamic generation of the service-side part of the stub (see Step 2 below).

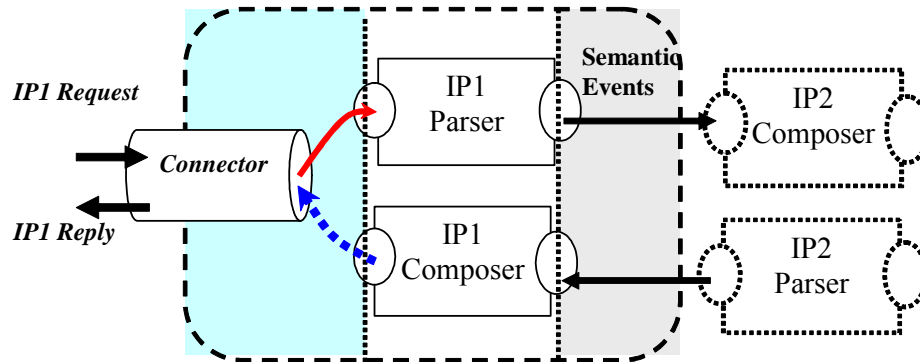


Figure 4-20: Interaction protocol interoperability relying on event-based parsing

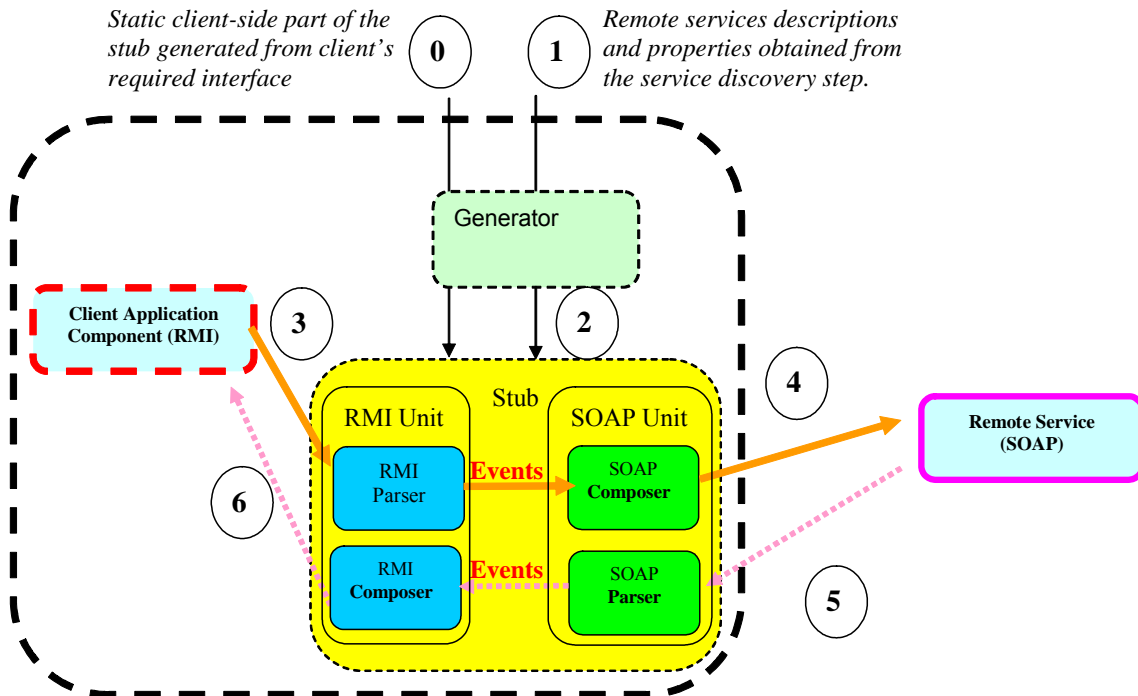


Figure 4-21: Interaction protocol interoperability with dynamic stub generation

More specifically, interaction protocol interoperability is achieved as follows (see Figure 4-21):

- **Step 0:** The *generator* uses the service's required interface of the client application component to generate the client-side part of the *stub*, along with the interface definition data that will be used for the dynamic generation in Step 2. In our example depicted in Figure 4-21, the generator will instantiate the RMI unit (RMI parser and RMI composer), and will create the definition of the RMI interface that will be used for the dynamic generation of the SOAP unit (SOAP parser and SOAP composer). The generator must take into account the interaction protocol paradigm for the instantiation of the components

and the generation of the interface definition data (in the example, RMI uses a synchronous RPC style).

- **Step 1:** The service's description and reference are obtained from the service discovery step. This step is tightly related to the discovery process and the corresponding SDP interoperability system. The service will be described in the service interface definition language (e.g., the SOAP service will be described in WSDL).
- **Step 2:** The generator dynamically instantiates the stub part dedicated to the remote service from the service's description and reference. This part amounts to instantiating the appropriate unit, taking into account the information on the client's required interface (obtained from Step 0) and the remote service's interaction protocol paradigm (available from the service description in Step 1). In our example, we assume that the SOAP remote service follows the synchronous RPC style, same as the RMI client application component.
- **Step 3&4:** The stub acts as the intermediary between the client and the remote service. Specifically, the stub presents to the client application component the same interface as the remote service, but in a compatible format. The client may therefore invoke service operations. Invocations are forwarded to the remote service in the appropriate format required by the service through the stub that holds the reference to the remote service.
- **Steps 5&6:** The remote service, in its turn, may reply to the client with its native protocol, as if the client were running a matching interaction protocol, thanks to event-based parsing interoperability.

Note that the proposed solution resolves the two aforementioned issues, i.e.: (i) the mismatch between service references that are specific to interaction protocols (retrieval of the service reference in Step 1, generation based on the service reference in Step 2, and use of the service reference in Step 4); and (ii) the identification of the incoming communication protocol needed to select the appropriate parser (instantiation of the parser in Step 2 and use in Step 5), together with the enforcement of the appropriate communication paradigm (stub generation based on communication paradigm in Steps 0 and 2). Nevertheless, this assumes a known mapping between the required and provided interface.

4.2.2 Early design and implementation

In this section, we present our early design and implementation of the service interaction interoperability. In Section 4.2.2.1, we provide an overview of the design and the relations among the classes that constitute the prototype, using UML class diagram descriptions. Then, in Section 4.2.2.2, we detail our implementation using UML class diagrams: the internals of each class and its relations with the other classes are surveyed to provide a better understanding of the technical details of the prototype implementation.

4.2.2.1 Overview

In this section, we present our design of the service interaction interoperability (SII) based on the design principles introduced in Deliverable D2.1 and recalled in Section 4.2.1. The design is at an early stage and is essentially a case of study to test our solution for a special case of configuration of client and service interaction protocols. This configuration has been defined to support the integrated prototype described in Chapter **Error! Reference source not found..** In the next phase of the project, this early design will serve as a basis for generalization and will be extended to a more advanced and detailed design that will cover all possible cases of client- and service-side interaction protocols without any restrictions in the configuration of these protocols.

In this first design of the SII, we address 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* (called *stub* in Section 4.2.1) to implement interaction interoperability are not

based on units (and their related components, that is, parsers and composers) and semantic events mechanisms. The alternative solution that we have 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, so the Proxy Provider component of middleware makes it available to be downloaded by the client together with the proxy.

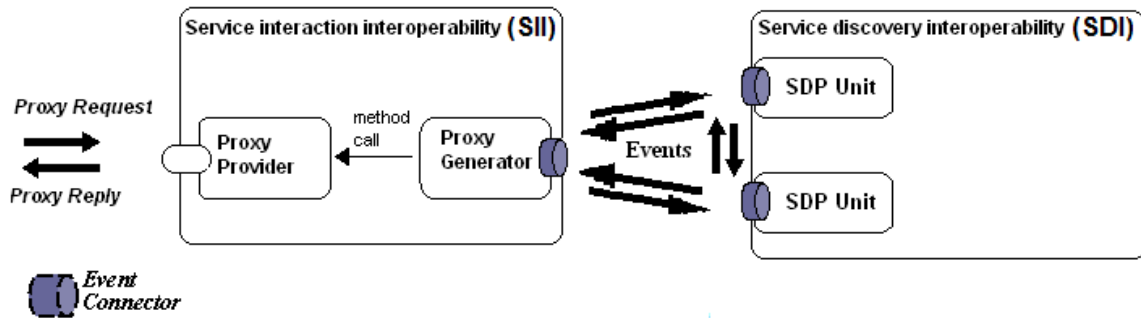


Figure 4-22: Early design of service interaction interoperability components

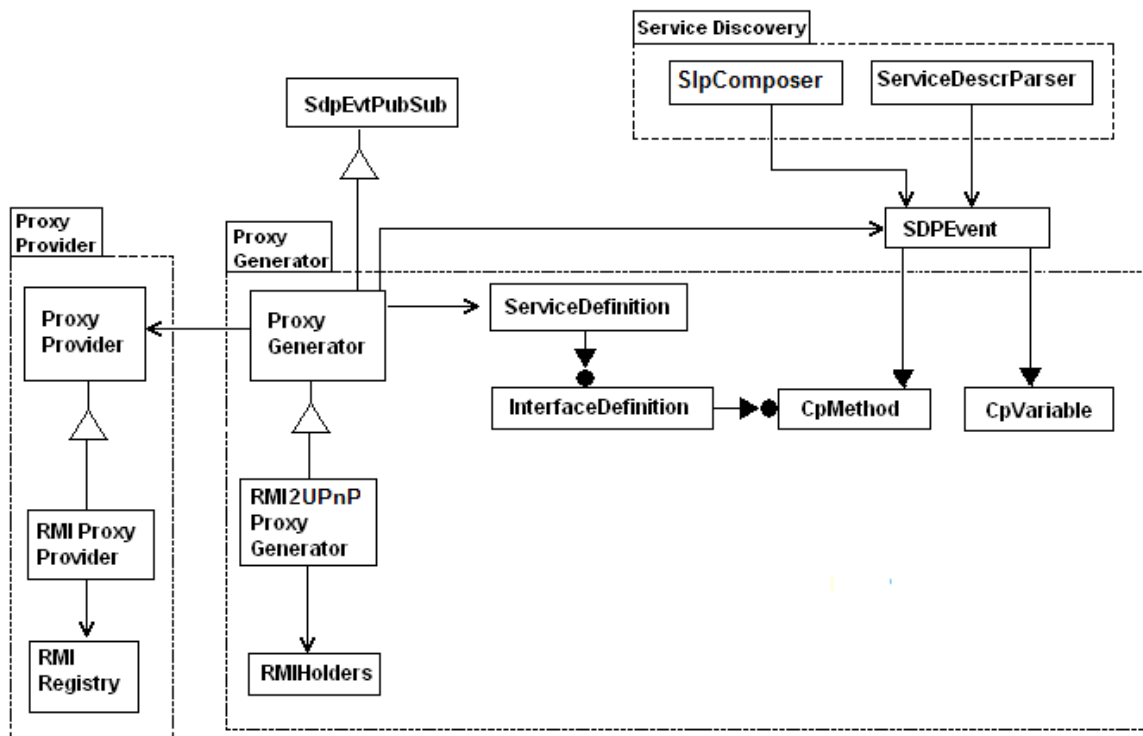


Figure 4-23: Early design of service interaction interoperability

Figure 4-22 depicts our solution for SII. Achieving SII requires service discovery interoperability (SDI) to provide SII with some information about the discovered services. The SDP units' event connectors are configured to mutually exchange events about the ongoing service discovery process. We add an event-based Proxy Generator component with an event connector and configured as a listener and publisher of events on the SDP units' connectors. It will receive all the SDP events and make use of those that are relevant for its function. The role of the Proxy Generator is the one described in Section 4.2.1, that is, to generate a proxy

that is in charge of providing the client with the expected service interface and generating message calls to the remote service in the expected interaction protocol. During the service discovery process execution, the Proxy Generator component receives some information from SDI and, using this information, builds the proxy that will be used by the client to access the remote service. When the proxy is created, the Proxy Generator invokes a method on the Proxy Provider to deploy it. The Proxy Provider will take into account the specification of the supported middleware technology to handle the interactions with the client.

Figure 4-23 depicts in more detail the components that build the SII shown in Figure 4-22 and their interactions with SDI's components.

- *ServiceDescrParser* is a parser activated by the UPnP unit (available from SDI) and designed to parse UPnP services' descriptions. It generates *SDPEvents* containing the semantic description of the service. The *SDPEvents* generated by *ServiceDescrParser* and received by the *ProxyGenerator* contain either *CpMethod* objects or *CpVariable* objects. *CpMethod* represents a service's method, while *CpVariable* represents a variable defined in a UPnP service.
- *SlpComposer* is an SDI component that implements the SLP protocol. In particular, one of its functionalities is to build SLP reply messages containing the address of the discovered remote services. Building such a reply involves an interaction with the SII Proxy Generator and Proxy Provider, because the message will contain the address of the proxy (for interoperability) as deployed by the SII Proxy Provider instead of the real address of the remote service.
- *ServiceDefinition* is a memory representation of a remote service. It includes information about the location of the service and a set of *InterfaceDefinition* objects, one for each interface exposed by the service. An *InterfaceDefinition* object includes the list of the provided methods, each one represented by a *CPMethod* object.
- *ProxyGenerator* defines the generic interface and implements the basic functionalities for proxy generation.
- *RMI2UPnPProxyGenerator* implements the logic of proxy generation for RMI clients and UPnP services.
- *RMIHolders* is a generic name for a set of classes introduced to resolve a difference between RMI and UPnP concerning the method arguments: UPnP services' methods support multiple output arguments for primitive types (e.g., numbers, strings) in opposition to RMI that supports only one return value.

The service interaction interoperability only supports RMI clients at this stage, so the Proxy Provider subsystem includes only an *RMIProxyProvider* that makes use of an RMI Registry from RMI technology.

The following is a short description of the steps implemented by the SII for the proxy generation:

1. The middleware obtains the service description. Each service can support one or more interfaces; the role of the unit in this step is to get all the information related to each of these interfaces, that is, the list of all methods provided to client for invocation. The service's description and reference are obtained from the service discovery step. This step is tightly related to the discovery process and the corresponding SDP interoperability system. In order to obtain the service description, each SDP unit (described in Section 4.1) is defined in conformance with the SDP specification and taking into account the supported interaction protocols.
2. The service description is parsed and transformed into a series of semantic events that are published in the same way and with the same mechanisms as SDP events described in

Section 4.1. The SDP unit coordinates the parsing using the SDP parser associated and able to understand the description of the service.

3. The semantic events generated in step 2 are used to reconstruct a memory representation semantically equivalent to the service description obtained in step 1.
4. The service memory representation obtained from step 3 is used to generate the proxy that will be used by the client to invoke methods of the service.
5. The client can finally interact with the remote service using the proxy as an intermediary. Specifically, the proxy presents to the client application component the same interface as the remote service, but in a compatible format. The client may thereby invoke service operations. Invocations are forwarded to the remote service in the appropriate format required by the service through the proxy that holds the reference to the remote service. The remote service, in its turn, may reply to the client with its native protocol, as if the client were running a matching interaction protocol.

4.2.2.2 Detailed description

This section details the different components and mechanisms that implement the service interaction interoperability: the SDI and SII interaction through the use of events, the proxy generator and the proxy provider. The details of the implementation of each component and mechanism are described using UML class diagrams.

SDI and SII interaction through events

Figure 4-24 shows the class diagram of *ServiceDescrParser*, a parser designed for UPnP services (it has already been described in Section 4.1) that is activated by the UPnP unit to parse the description of a service. Its method *parse* reads the service description content and generates *SDPEvent* events.

ProxyGenerator defines the generic interface and implements the basic functionalities for proxy generation. As its role is to generate the proxy for the remote service, it requires all the necessary information about the service description. This information can be provided by SDI and, in particular, by the *ServiceDescrParser* by means of appropriate events. *ProxyGenerator* is interested in events of type *SDP_SERVICE_METHOD_DESCRIPTION* and *SDP_SERVICE_VARIABLE_DESCRIPTION*. The former contains an object of type *CpMethod* with the definition of a method (name, arguments, service and interface reference). For each argument of the method, it contains the name, the data type and the direction (i.e., IN for an input parameter and OUT for a return parameter). The latter event type contains an object of type *CpVariable* used to define a UPnP variable, as described in the UPnP service description (name, data type, service and interface reference). This second type of event has been introduced because the definition of method arguments is divided in two sections in the UPnP service definition. In the first section, the methods with their name, argument names and respective directions are defined. Then, in the second section each argument data type is defined. The complete method definition (with association of arguments with their respective data types) will be reconstructed when all the information from the two sections will be collected by the *ProxyGenerator*.

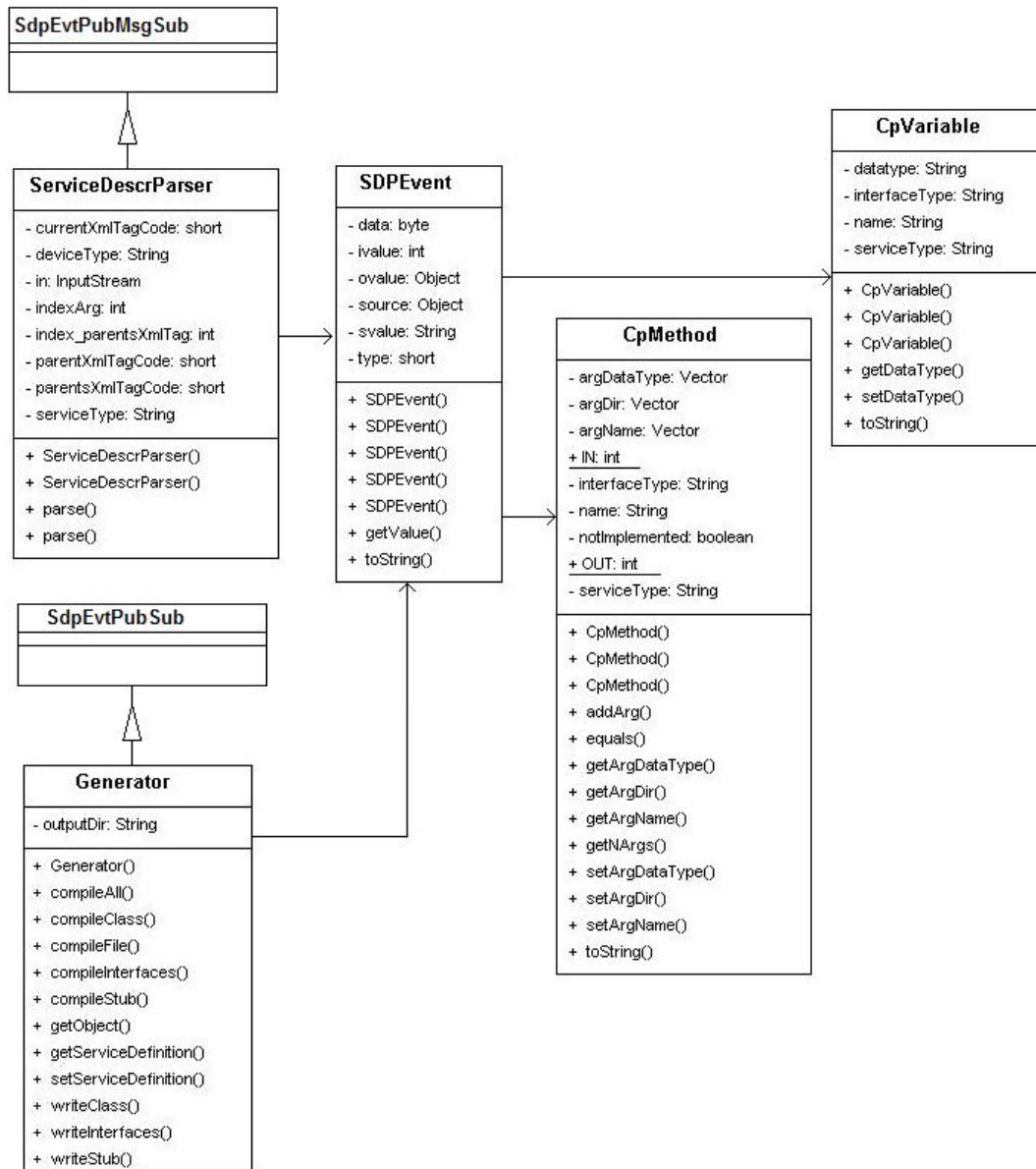


Figure 4-24: Diagram of classes used for interaction between service discovery and service interaction interoperability systems

Proxy Generator

Figure 4-25 shows the class diagram of the classes involved in the process of remote service proxy generation.

When all the events concerning service's methods description (*SDP_SERVICE_METHOD_DESCRIPTION* and *SDP_SERVICE_VARIABLE_DESCRIPTION*, discussed in the above subsection) have been received and processed by *ProxyGenerator*, the latter finally produces a memory representation of the service in a *ServiceDefinition* object. The *ServiceDefinition* memory representation and the actual remote service description are semantically equivalent, but the *ServiceDefinition* object is independent and not aware of the technologies and

protocols on which the service is based. *ServiceDefinition* includes the name of the service along with some information about the location of the service: the IP address of the host and the port where the service is running. Further, it contains a series of *InterfaceDefinition* objects, one for each interface exposed by the service.

The *CPMethod* and *CPVariable* objects contained in the events handled by the proxy generator have a reference name to the service and interface they are associated with. This information is used by the generator to associate the method definition to the right *InterfaceDefinition* object of *ServiceDefinition*. The *InterfaceDefinition* object contains the name of the service's interface together with some information (*URIPath*) used to construct the complete URI address that has to be used to invoke a method call on the specific service's interface. The complete URI of an interface's method call can be reconstructed by using the address obtained by concatenation of the following data: *ServiceDefinition.host* + *ServiceDefinition.port* + *InterfaceDefinition.URIPath*. An *InterfaceDefinition* object further includes the definition of the provided methods in a list of *CPMethod* objects. Each *CPMethod* object contains the name of the method it represents, the list of arguments with respective name, data type and direction (either input or output argument), and the name of the service and interface it is associated with.

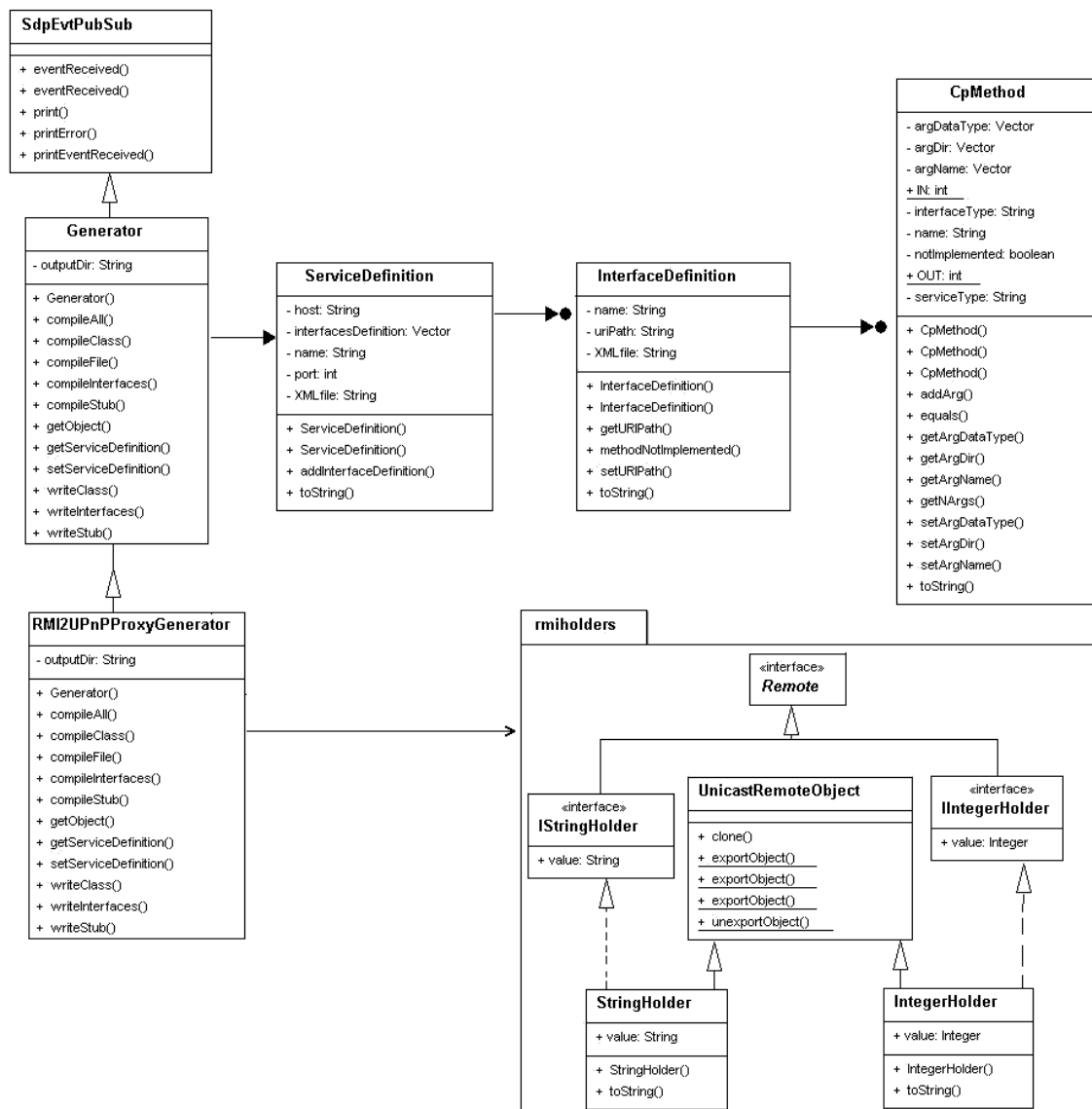


Figure 4-25: Diagram of classes used for proxy generation

Starting from the *ServiceDefinition*, *RMI2UPnPProxyGenerator* realizes the logic of proxy generation. As we indicated above, *RMI2UPnPProxyGenerator* is, at the moment, the only implementation of *ProxyGenerator* provided by SII, and it addresses specifically the case of interoperability between a RMI client and a UPnP service. To achieve this task, it has to produce a RMI-compatible proxy that will be provided to the client to access the remote service. The proxy must present a RMI-compatible interface and, at the same time, instantiate the part of the proxy dedicated to the remote service, that is, it must implement UPnP service call marshalling and unmarshalling (producing UPnP compatible messages and being able to understand the UPnP replies from the service).

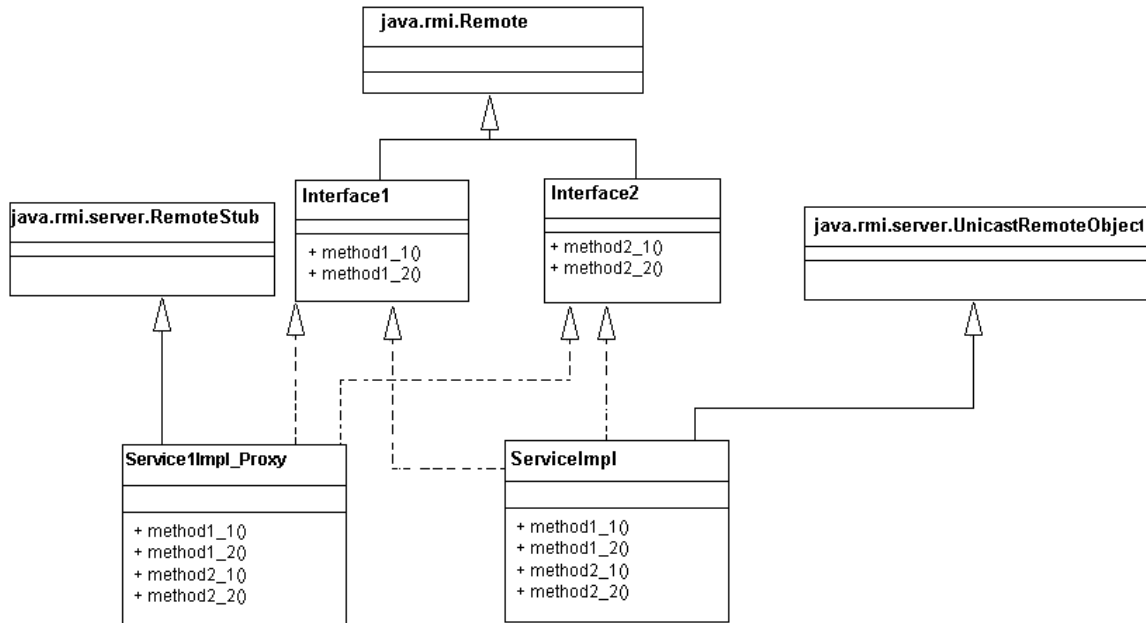


Figure 4-26: Example of generated classes for interfaces, service and proxy

For the deployment of the proxy on the Proxy Provider (see next subsection for technical details), the *RMI2UPnPProxyGenerator* assumes that the service implements all the interfaces in the *InterfaceDefinition* list. Since the proxy must be a front-end to the service, it must implement the same interfaces.

Each *InterfaceDefinition* object corresponds to a `.java` interface file generated with the method *writeInterface* and named *InterfaceDefinition.name*. The generation of this file involves the creation of the signatures of all the methods owned by the interface including: method's name, return value, arguments with their name, type (mapped to the corresponding java/RMI data type) and direction (input or output).

Then, *writeClass* generates a `.java` file named *ServiceDefinition.name* and representing *ServiceDefinition*: it must implement all the interfaces. However, since its role is simply to permit the deployment of the proxy, it will contain only the list of the methods from all the interfaces; their respective implementation will be void.

Finally, the proxy is generated with the *writeProxy* method. The result of the proxy generation is a `.java` file that implements all the methods from all the service's interfaces. The proxy file is generated taking into account the communication protocol of the remote service and the client required interface. For each method call, the method name and the input arguments

(with their names and values) are used to generate a UPnP message that is sent on an HTTP connection, to the address *ServiceDefinition.host* + *ServiceDefinition.port* + *InterfaceDefinition*.

Figure 4-26 shows an example of the generated classes and interfaces for a service *Service1* defined with two interfaces with two methods respectively. The interfaces generated (*Interface1* and *Interface2*) must inherit from *java.rmi.Remote* and must define the methods following the RMI specifications. Then, the service implementation *Service1Impl* inherits from the RMI class *java.rmi.server.UnicastRemoteObject* and implements the two interfaces with a void implementation of all the four methods. Finally, the actual implementation of the proxy, *Service1Impl_Proxy*, inherits from the RMI class *java.rmi.server.RemoteStub* and provides an implementation for all the four methods from the two interfaces consisting in the remote UPnP service call.

When all the *.java* files for interfaces, service and proxy classes have been created, then *compile* methods (i.e., *compileStub*, *compileInterface* ...) are used to transform *.java* files into java binary *.class* files.

In the creation of method calls in proxy and in interfaces, the different specifications between UPnP and RMI raise a problem concerning methods' arguments that must be resolved by SII. UPnP services' methods support multiple output arguments for primitive types (e.g., numbers, strings) in opposition to RMI that supports only one return value. Because of this difference in the two technologies, the middleware introduces a collection of classes that appears in Figure 4-25 with the generic name of *rmiholders*. *rmiholders* are used in the generated RMI proxy in replacement of method's output arguments. For each primitive data type (e.g., *String*, *Integer*) we define an *rmiholder* class (e.g., *StringHolder*, *IntegerHolder*); each method output argument (one or more) in the *CpMethod* definition is replaced with the corresponding *rmiholder* class to hold the argument value returned by the method invocation.

Proxy Provider

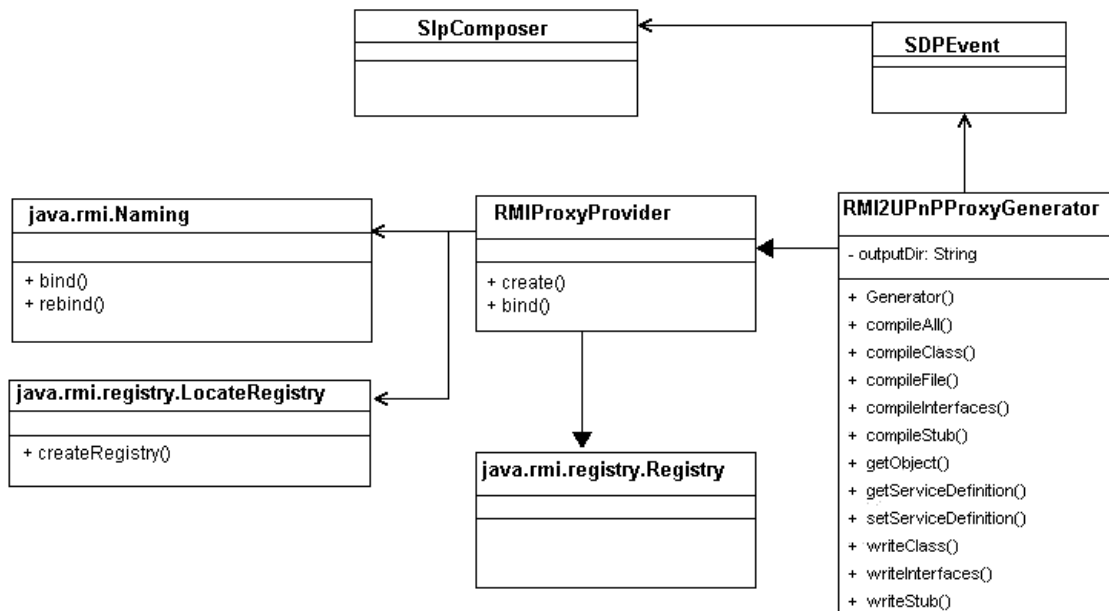


Figure 4-27: Diagram of classes used by proxy provider

The role of the proxy provider component is essentially to set up the system for enabling the client to obtain the proxy generated by the *ProxyGenerator*. As the proxy provider is tightly

related to the Proxy Generator, the current version of SII provides only an RMI protocol implementation for this component (*RMIProxyProvider*).

As Figure 4-27 shows, *RMIProxyProvider* is implemented using the classes provided by RMI technology: *java.rmi.registry.LocateRegistry* is used to create an instance of *java.rmi.registry.Registry*, while *java.rmi.Naming* is used afterwards, when the registry has been created and *ProxyGenerator* has completed the proxy generation process and has to add a new instance of the proxy in the RMI registry.

Adding a new proxy instance for a discovered service involves binding a proxy object instance (obtained from *ProxyGenerator.getObject*) to a corresponding address in the RMI Registry. In the example of Figure 4-26, the object that will be bound is *Service1Impl*; the RMI Registry will associate *Service1Impl_Proxy* to it and check the existence of all the implemented interfaces. When a service proxy is bound to one address through the registry, it is immediately available at the given address and the registry is in charge of providing it to clients asking for a proxy instance.

The proxy generated by *RMI2UPnPProxyGenerator* to invoke remote service methods makes use of the UPnP protocol. The client that will make use of it is not aware of the fact that the remote service and the downloaded proxy are UPnP-based. Thus, not all the classes referenced by the proxy are available on the client side. The RMI registry is configured to make accessible all the classes referenced by proxies on an HTTP server. It is up to the *RMI2UPnPProxyGenerator* and *RMIProxyProvider* to export the classes generated along with the proxy and the classes they reference by following the RMI registry configuration.

The address to which the proxy is bound is finally sent, as an *SDPEvent*, to the *SLPComposer*, which will use it in replacement of the actual UPnP service address to create the SLP reply for the client.

Then, the RMI-based client will make use of the standard RMI protocol to access the RMI Registry *java.rmi.registry.Registry* created and managed by the *RMIProxyProvider* to obtain the service proxy.

4.3 Programming and deployment framework for Amigo services

4.3.1 Overview

This section aims to define programming interfaces and a programming framework to be used by developers of either application services or middleware components of the Amigo system. The goal is to enable middleware components and application services to be developed independently on the basis of these well-defined interfaces and then deployed together on the same execution platform to form an "instance of the Amigo architecture" as defined in D2.1 [Amigo-D2.1].

To that purpose, components need:

- to agree on programming interfaces associated to Amigo abstractions like a service, a service description, a service request, a context information etc.;
- to agree on some bootstrapping mechanism that allows components running on the same execution platform to execute together.

To fulfill the first need, programming interfaces can be specified independently of the programming language and the execution environment. The second need, however, refers to an execution framework. Rather than defining a new "Amigo execution platform", we propose to rely on existing standards, such as the .NET application platform or the OSGi application platform.

OSGi specifications define a standardized, component-oriented computing environment for networked services. It is supported by a great number of companies, the OSGi Alliance, and benefits from an active industrial and free community (the OSGi implementation Oscar is now, for instance, an Apache project). The OSGi framework defines a Java platform where components called "bundles" can be deployed, started, stopped, or updated at run-time. Bundles can interact by publishing services and by using services published by other bundles (the word "service" stands here for a Java object exposing a well-known Java interface).

.NET is a software development platform focused on rapid application development, platform independence and network transparency. .NET technology provides the ability to quickly build, deploy, manage, and use connected, security-enhanced solutions with Web services and other network technologies. The .NET Framework is language neutral: currently, it supports C++, C#, Visual Basic, JScript and COBOL. The .NET Compact Framework is a streamlined version of the .NET framework designed to run on mobile devices with limited resources, like memory and battery power, including smart devices like PDAs, smartphones and set-top boxes.

Section 4.3.2 provides the first design of Amigo basic interfaces, while section 4.3.3 is an early design of the Amigo OSGi-based deployment framework and deals with defining behaviour for "Amigo-conformant OSGi bundles". Although implementation issues are beyond the scope of this section, an example of implementation is given in the case of UPnP.

4.3.2 Early programming interfaces design

This section defines basic interfaces that correspond to abstractions of the Amigo architecture, such as:

- Interfaces corresponding to application services and networked devices ("Amigo Service", "Amigo Action", *AmigoServiceDescription*, etc.). These interfaces are described in Section 4.3.2.1;
- Interfaces corresponding to Amigo middleware services. In this early design, only the "enhanced discovery lookup" is described in Section 4.3.2.2.

These interfaces do not rely on any technology and could be used in any context of development language or environment.

4.3.2.1 Basic concepts: *AmigoService*, *AmigoServiceDescription*, *AmigoAction*

As shown in Figure 4-28, an *AmigoService* is an object that provides a set of actions. Implementations of *AmigoService* could be:

- a proxy to a networked service in a particular technology;
- a local service; or
- the result of the composition of services linked or not to a device .

An *AmigoAction* represents an action that can be invoked on an *AmigoService*. If the *AmigoService* is a proxy, invoking an action will send a message through the corresponding protocol stack, wait for the response, and return the result.

AmigoServiceDescription represents the semantic description of the service using the Amigo semantic service specification language introduced in Chapter 2. This interface is not detailed in this early design.

A description of available actions must be provided by each service through the *getActions* method. Each service can also export the description of the device (location, hardware references ...) to which it is attached through the *AmigoDeviceDescription*. Note that an *AmigoService* is not necessarily linked to a hardware device and therefore can export an empty *AmigoDeviceDescription*. When relevant, this device description is however useful, in

particular for context-aware discovery: when looking for the nearest service of such kind, it is helpful to know which physical device hosts such or such services.

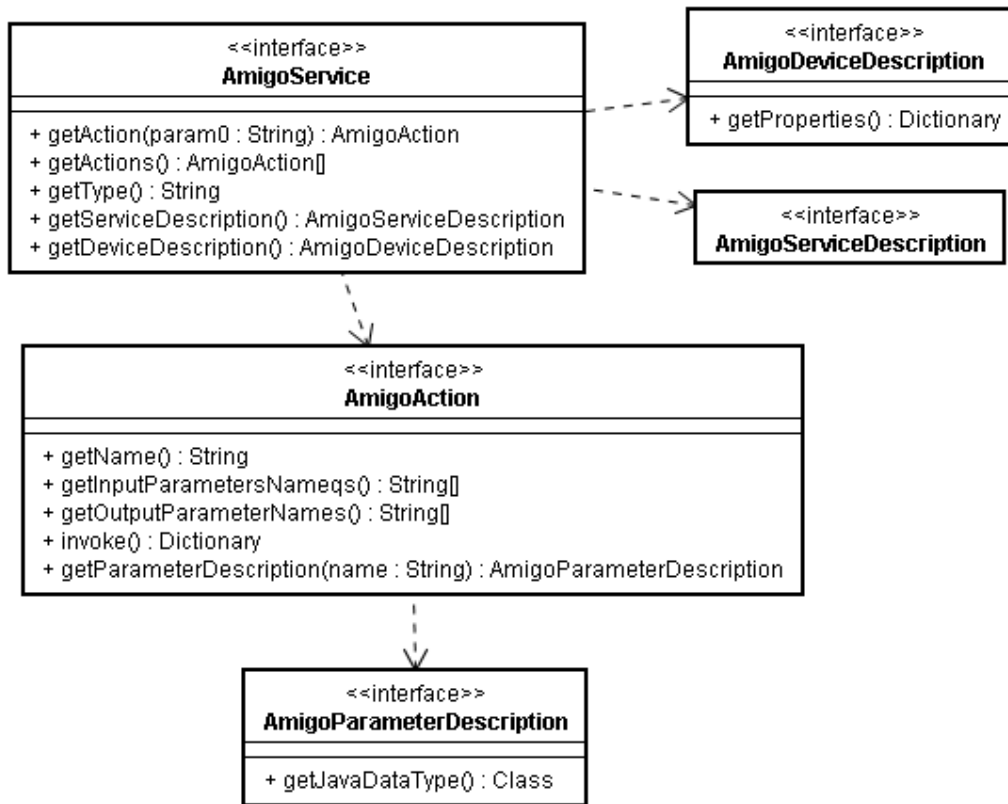


Figure 4-28: Class diagram of Amigo basic concepts: AmigoService, etc.

4.3.2.2 The Amigo enhanced lookup

This section specifies the programming interface that an implementation of the Amigo enhanced lookup should provide to components running on the same execution platform (Amigo-aware clients). As shown in Figure 4-29, the *AmigoServiceLookup* interface allows:

- Active discovery: retrieve services matching semantic criteria using the *lookupService* method.
- Passive discovery: subscribe for being notified of services matching semantic criteria using the *register* method. Clients must provide an instance implementing the *AmigoLookupClient* interface.

4.3.3 Early OSGi-based deployment framework design

In order to develop an application that, for example, uses the *AmigoService* interface, knowing the interface is not enough: the life cycle of objects must be defined, and bootstrapping methods to retrieve instances implementing interfaces are required. The following sections deal with these aspects in the context of an OSGi platform.

The OSGi framework offers several facilities, among which the *local service lookup* and *service tracker* that allow deployed components to:

- Register "OSGi services" (that is, Java objects) together with a set of properties;
- Retrieve registered "OSGi services" that exhibit some interface, possibly providing a filter (in LDAP form) on declared properties;
- Ask to be notified anytime an OSGi service corresponding to a given Java interface and a filter on declared properties is registered or unregistered.

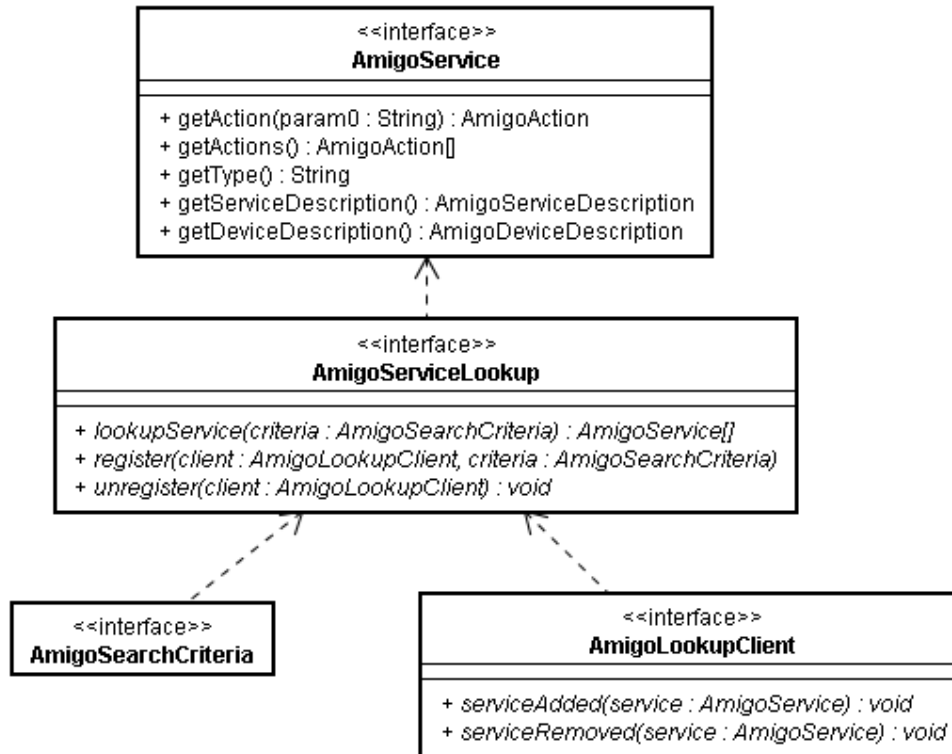


Figure 4-29: Class diagram of AmigoServiceLookup

The Amigo OSGi framework relies on the service lookup and service tracker (together simply called "OSGi lookup" in the following) to bootstrap interactions between "Amigo-aware bundles". By 'Amigo-aware bundles', we mean either a middleware component or an application component packed in an OSGi bundle.

Section 4.3.3.1 defines the contracts that Amigo middleware bundles will have to respect in order to be integrated into the Amigo platform. It defines, for example, "Amigo driver" (an Amigo driver is an OSGi bundle that is specialized on a given technology, e.g., UPnP, and instantiates java objects implementing the AmigoService interface according to devices present on the network), and "Amigo publishers", which make available AmigoService instances according to a given technology. Then, Section 4.3.3.2 defines how Amigo-aware components running on an OSGi platform will use an OSGi-based Amigo middleware. Finally, Section 4.3.3.3 discusses the implementation of drivers and publishers.

An OSGi platform running on a given device with a given set of Amigo components will define an OSGi-based "instance of the Amigo architecture". It will be able to interact with legacy devices and services, as well as with other instances of the Amigo architecture which may be (or not) OSGi-based.

Our introduced specifications are inspired from the "UPnP base driver" specifications in OSGi specification release R3: OSGi release R3 defines first a set of Java interfaces (UPnPDevice,

UPnPService, UPnPAction) that correspond to the concept of UPnP protocol. It defines also the contract that a bundle should respect in order to be considered a conformant "UPnP Base Driver". A "UPnP Base Driver" does not provide any interface per se. When started, it performs in background the following tasks:

- Manage the SSDP protocol, and provide instances of UPnPDevice corresponding to each UPnP device discovered. These instances are published by the OSGi lookup and can be discovered by any other bundle that knows only the Java interfaces. Invoking methods on these instances will transparently result into UPnP requests and responses.
- Listen to the OSGi registration of local objects implementing UPnPDevice, and announce these devices onto SSDP.
- Handle requests coming from the network concerning local UPnP devices and transform these requests into Java method invocations.

4.3.3.1 Amigo conformant bundles

Amigo drivers

An *Amigo Driver* is (generally) specific to a protocol, for example UPnP. It is in charge of building and managing proxies corresponding to remote services that use this technology. It does not provide an interface per se (or may provide an administration interface, to allow, e.g. for filtering or scoping). It provides implementations of AmigoService, AmigoAction, etc. corresponding to the specific technology. Its contract is:

- Whenever it discovers a new service on the network, it creates a proxy for this service and publish it locally as an AmigoService (using the OSGi and Amigo lookups).
- By invoking on published proxies the methods specified by the basic interfaces (AmigoService, AmigoAction...), it performs the necessary network operations so that the corresponding action is invoked on the remote service.
- Whenever it detects that a service has disappeared from the network, it deregisters the objects that have been built (according to the protocol, this may be a lease system or a direct "deregister" announcement or a combination of both).

Clients do not directly interact with the drivers: they take advantage of the available Amigo drivers by discovering the Amigo services using the OSGi lookup or the enhanced Amigo lookup.

Amigo publishers

An *Amigo Publisher* bundle is (generally) specific to a communication and a discovery protocol. It allows local Amigo services (instances of AmigoService) to be published using a given network discovery protocol. It takes care of:

- Handling active requests coming from external clients: asking the local Amigo lookup, and producing the corresponding answer;
- Listening to the publication of local Amigo services and publishing these services onto the network: this generally involves exporting the service (building a reference or URL for this object) and publishing this reference on a discovery protocol like SLP or SSDP;
- Handling network calls onto the published URLs: parsing the requests to find which local AmigoService is targeted, the action name and the parameter values; and then invoking the corresponding AmigoAction, building the response and sending it on the network.

Note that the same bundle may act as both a driver and a publisher, as in the case of the UPnP base driver.

The presence on the same platform of a driver for a given technology T1 and a publisher for another technology T2 enables interoperability between a client using T2 and a server using T1. Thus, a developer has only to concentrate on using the technology he/she is familiar with, and the AmigoServiceLookup will automatically provide discovery and interaction of this service in the different available technologies.

Amigo enhanced lookup bundle

A conformant Amigo enhanced lookup bundle tracks locally registered instances of AmigoService. It publishes an instance of AmigoServiceLookup. Clients have therefore two ways of finding Amigo Services:

- Non-Amigo-aware clients can use the OSGi lookup, which allows research criteria to be specified in LDAP syntax;
- Amigo-aware clients can use the AmigoServiceLookup, which is able to handle complex research criteria and complex service description, possibly including dynamic context information.

An advanced implementation of AmigoServiceLookup may use semantic matching to create on demand a new AmigoService as a composition of several already available services.

4.3.3.2 Using OSGi-based Amigo middleware

An Amigo client or middleware service hosted on an Amigo OSGi platform can use the OSGi lookup (or the Amigo enhanced lookup if available) to discover instances of AmigoService that are available on this platform. These instances are either local to the platform or proxies to services hosted by remote devices. Figures 4-30 to 4-34 show examples of use.

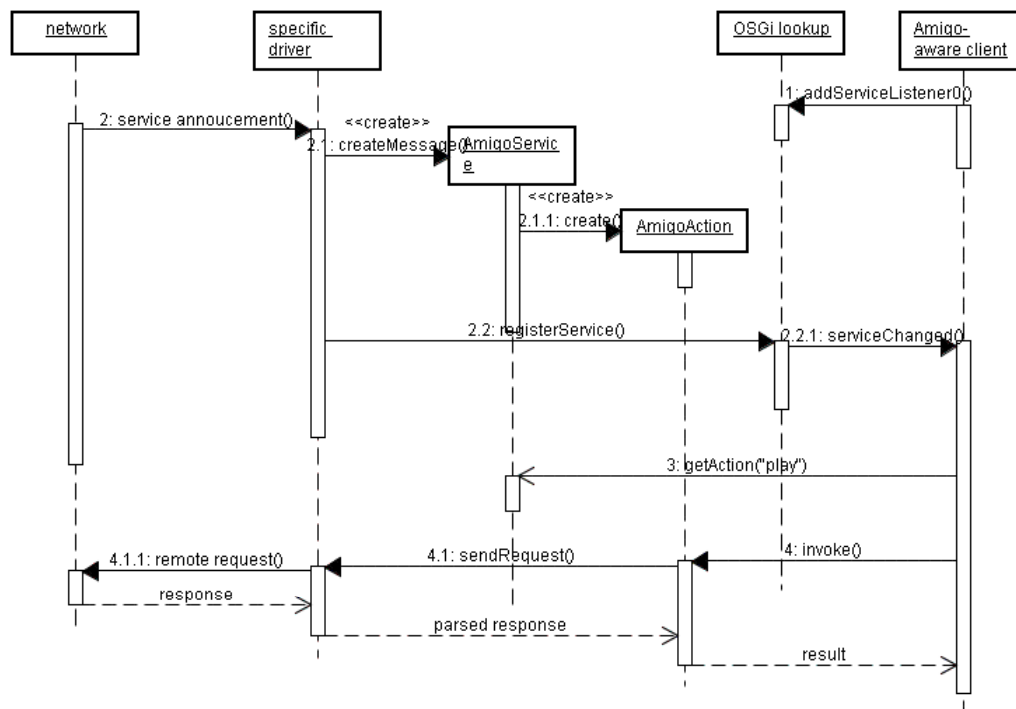


Figure 4-30: Sequence diagram showing an OSGi discovery example

In Figure 4-30, an Amigo-aware client uses passive discovery to obtain a media player (1). A technology-specific driver detects a service announcement on the network (2). It creates an instance of AmigoService and registers it locally (2.2). The client is notified (2.2.1). It asks for an action called "play" (3). The client invokes the action (4), which results in sending a request on the network, waiting for the response, parsing the response and returning the result to the client.

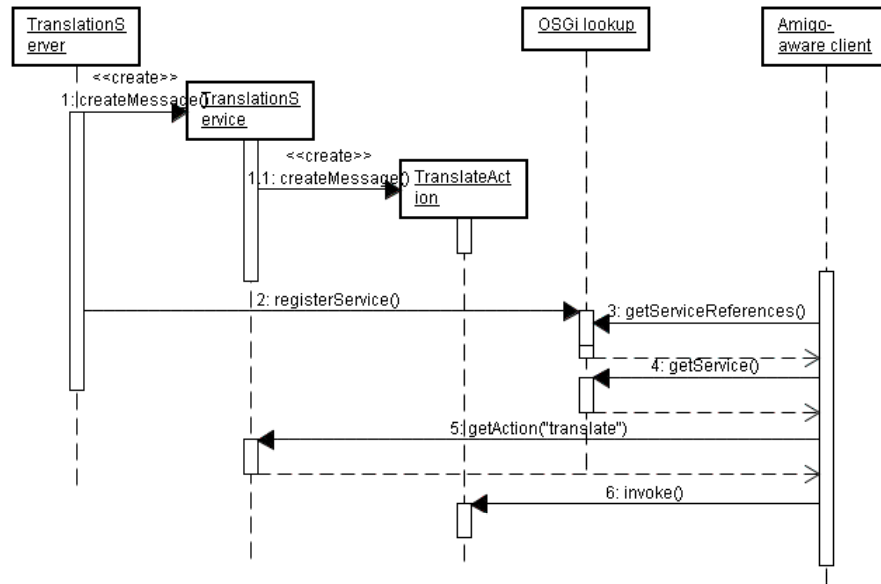


Figure 4-31: An Amigo-aware client uses the OSGi lookup and active discovery to retrieve a Translation Service

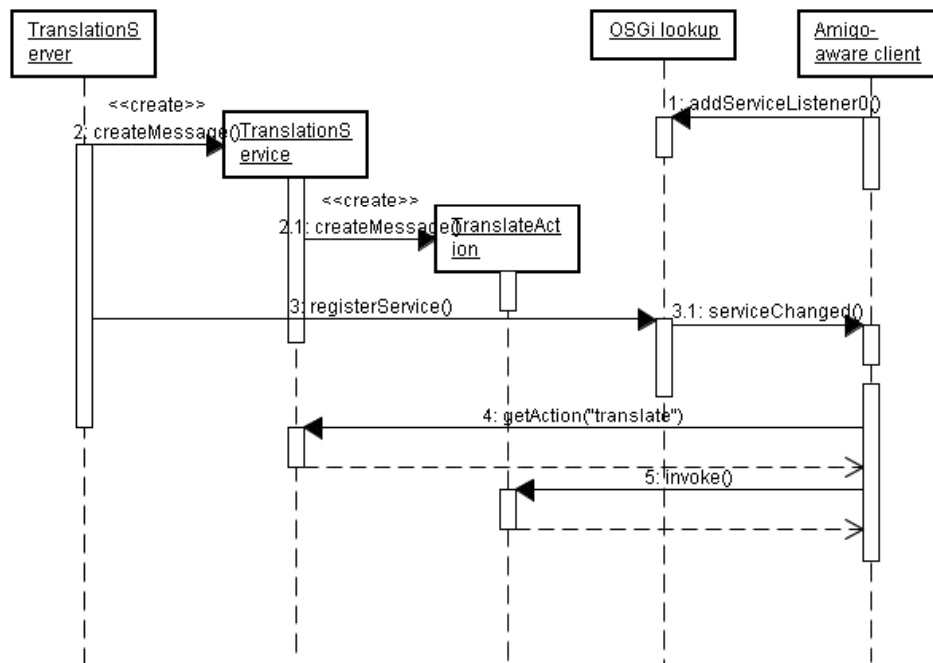


Figure 4-32: An Amigo-aware client uses the OSGi lookup and passive discovery to retrieve a Translation Service.

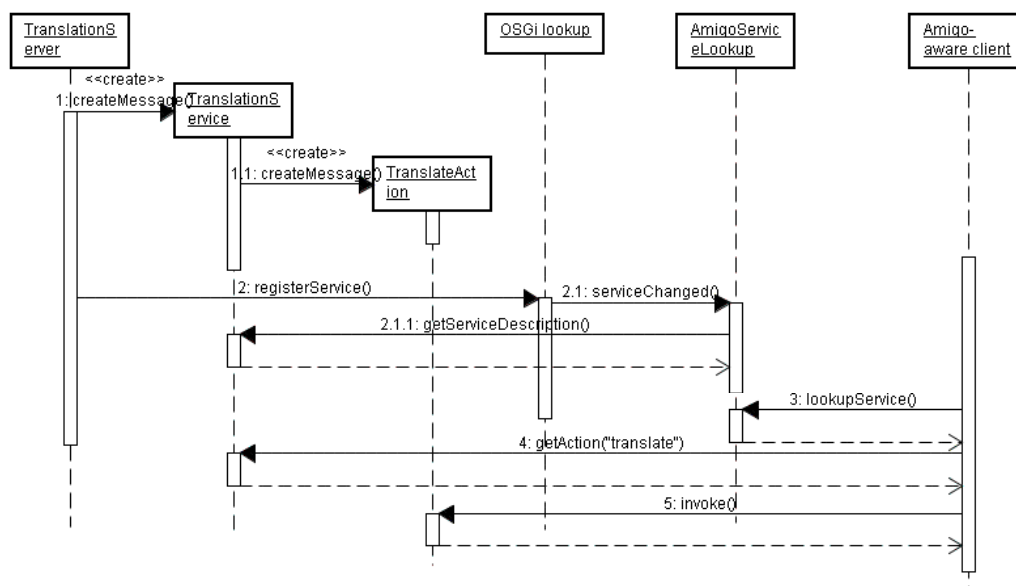


Figure 4-33: An Amigo-aware client using the Amigo enhanced lookup

In Figure 4-33, an Amigo-aware client is using the Amigo enhanced lookup. The Amigo Service lookup listens to OSGi registration of AmigoServices. When an AmigoService is announced (2.1), it asks for the service description in order to be able to provide enhanced service discovery.

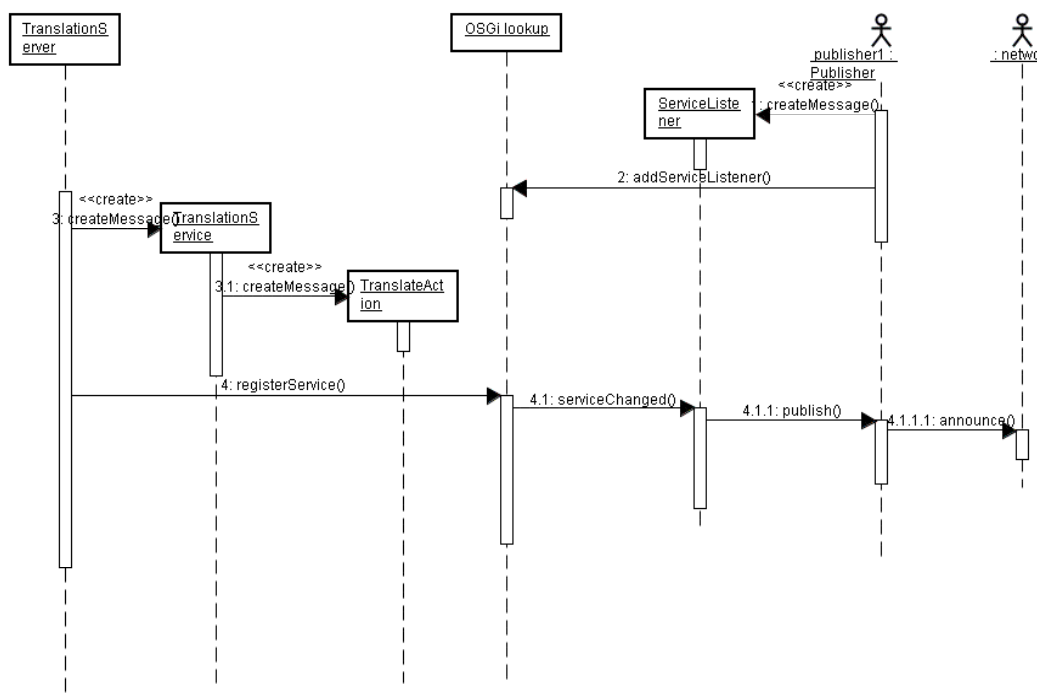


Figure 4-34: Publication of a translation service according to a specific protocol

In Figure 4-34, a translation service is published according to a specific protocol: the translation server creates an Amigo service and registers it locally. If a network publisher is

present, it publishes it on the network according to the communication protocol(s) and discovery protocol it handles.

4.3.3.3 Implementing drivers and publishers

The internal design of drivers and publishers may follow different patterns. Hereafter, we provide a non-exhaustive discussion of implementation methods.

First, implementation may rely on monitors, composers and parsers as defined in Section 4.1. This is not specified in this early design.

Second, in some cases (e.g. Jini, UPnP) where work on OSGi mapping has already been done by a standardization group, a convenient way is to build on this work. For example, an Amigo UPnP driver listens to the creation of UPnPDevice instances: whenever a UPnPDevice corresponding to a remote device is announced on the OSGi lookup, the driver requests the services of this device, creates the corresponding UPnPAmigoService instances and publishes them. The generic UPnPAmigoService class implements the AmigoService interface that contains references to a UPnPService and a UPnPDevice, and delegates all handling to these encapsulated objects (see Figure 4-35).

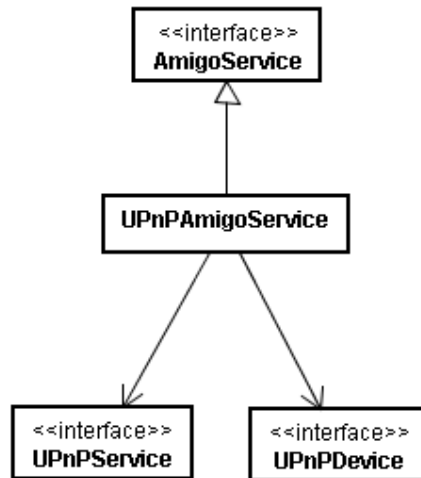


Figure 4-35: Example of implementing an AmigoService as an UPnPAmigoService

This is illustrated in Figure 4-36. The AmigoUPnPDriver declares a service listener (1) to OSGi service events. When a new UPnP device is announced through SSDP, the UPnP driver creates (3) and publishes (4) an instance of UPnPDevice. The AmigoUPnP driver service listener interface is notified (4.1). It calls the `getServices` methods of the UPnPDevice (4.1.1) and creates for each service an instance of AmigoUPnPService (4.1.2). It announces these services on OSGi (4.1.3) so that other Amigo-aware bundles can discover them.

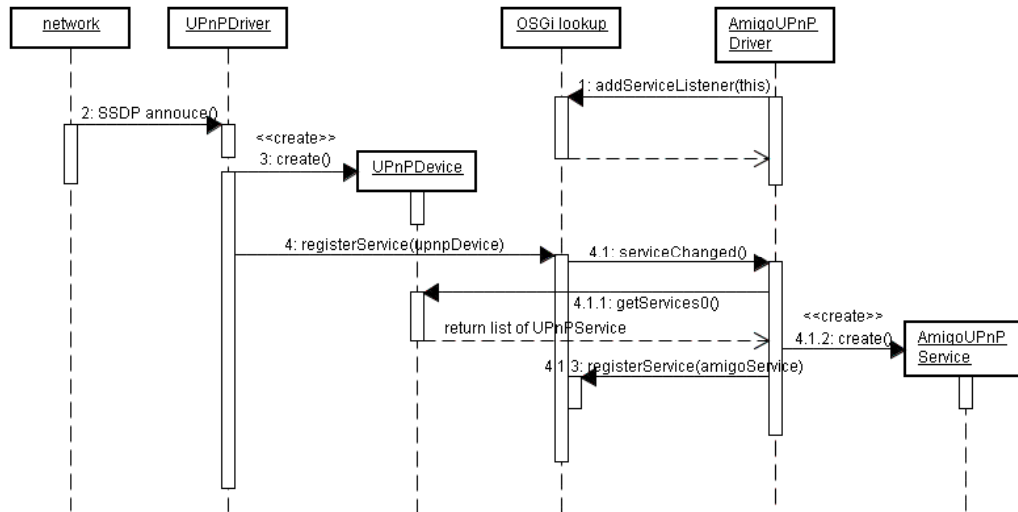


Figure 4-36: Registering UPnP Services of an UPnPDevice as AmigoUPnP Services

Once the AmigoUPnPService is declared, clients can retrieve it by using either the OSGi lookup or the Amigo enhanced lookup. Figure 4-37 illustrates a case of use where a client asks for AmigoService with certain properties (1). Service references are returned by the OSGi lookup, among which that of a UPnPAmigoService instance. The client decides to use this service (2) – however, the client knows only the AmigoService interface exposed by the UPnPAmigoService. The client requires a specific action by its name (3). It invokes the action (4). Invoking the action results into invoking the corresponding UPnPAction (4.1), which results in a SOAP request. Responses are handled symmetrically.

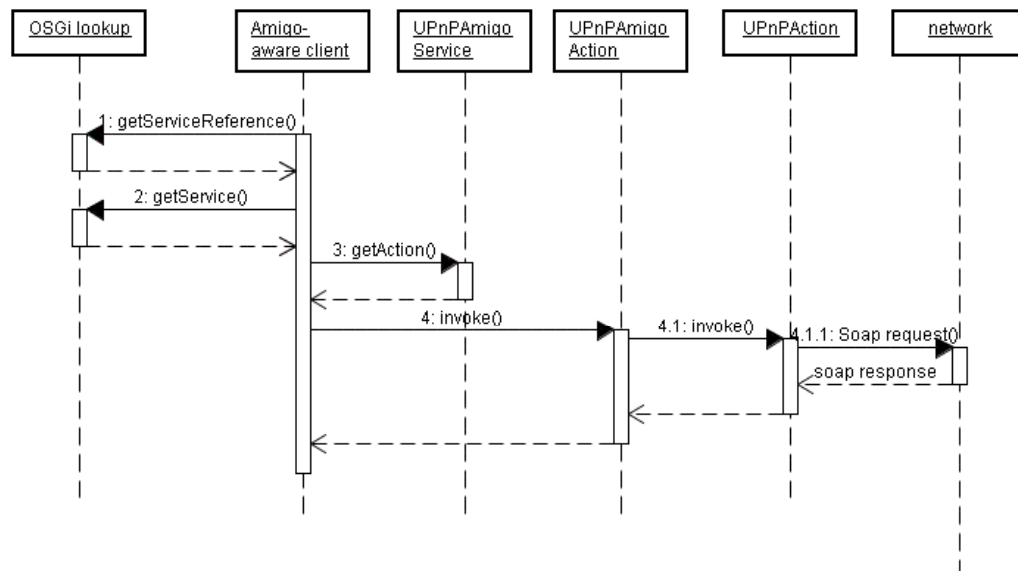


Figure 4-37: Lookup and access to the UPnPDevice through the AmigoUPnPService

Symmetrically, a UPnP publisher provides encapsulations of AmigoService that implement the UPnPDevice and UPnPService interfaces (see Figure 4-38). Whenever a local AmigoService is announced, it creates an instance of AmigoUPnPService – and an instance of AmigoUPnPDevice – and publishes the AmigoUPnPDevice on the local OSGi registry. If a

UPnP base driver is started (at this time or later on), it will be notified and publish the AmigoUPnPDevice using SSDP.

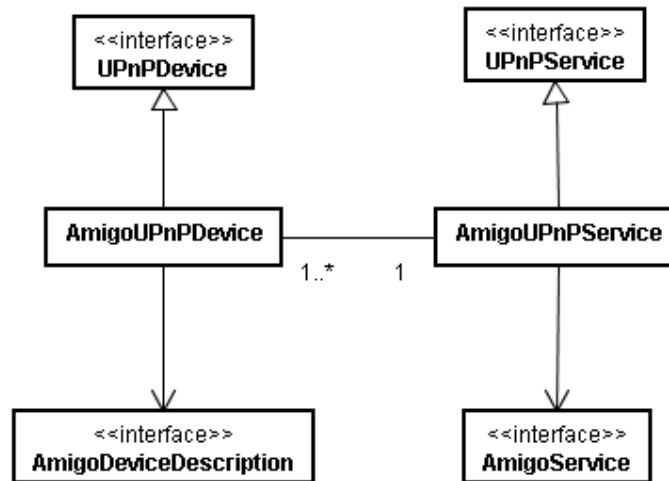


Figure 4-38: Mapping of AmigoUPnPDevice and AmigoUPnPService with UPnPDevice and UPnPService

Third, a publisher may also be composed of an export factory linked to a specific communication protocol and a lookup linked to a specific discovery protocol: the export factory provides an "export" function: `String export(AmigoService service)` that possibly builds useful structures (by calling `UnicastRemoteObject.export` and `Naming.rebind` in the case of RMI, `POA.export` in the case of CORBA) and returns a URL. The lookup provides a method that allows to register a URL according to a given discovery protocol.

As stated before, this discussion of implementation methods is not exhaustive, and several publishers/drivers following different designs may be deployed on the same platform and interact with each other or with other Amigo-aware bundles.

4.4 Domotic interoperability

This section describes the interoperability mechanisms applied to realize domotic domain interoperability. Initially, we recall from Deliverable D2.1 [Amigo-D2.1] the design principles on which domotic domain interoperability is based (Section 4.4.1). Then, we provide an overview of the early design of domotic interoperability and the details of the prototype implementation using the UML language (Section 4.4.2).

4.4.1 Design principles

The Amigo domotic service architecture aims at integrating the diverse existing domotic systems towards flexible networked domotic service provision in the Amigo home environment.

The Amigo interoperable middleware core supports different service discovery protocols (e.g., UPnP and SLP), but, unfortunately, currently existing domotic systems are not ready to interact with it yet. As described in Deliverable D2.1, there is a great diversity and heterogeneity of domotic devices. Amigo device classes were introduced there in order to classify the existing domotic devices. Consequently, the middleware shall support the different domotic systems and provide interoperability mechanisms amongst them. In Deliverable D2.1, the following architectural components were introduced: *bus controllers*, *proprietary device*

factories and *discoverable device factories*, which gradually enable passing from proprietary access mechanisms to common, technology-independent interfaces for domotic devices (see Figure 4-39). By employing these components, the Amigo domotic device classes can be integrated into the Amigo domotic architecture.

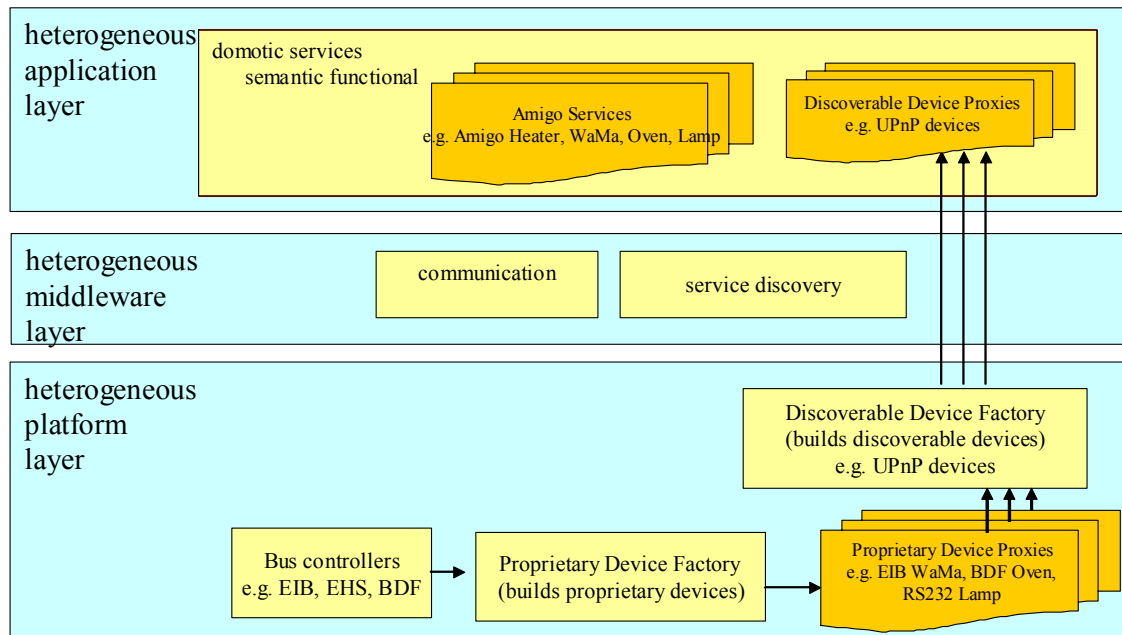


Figure 4-39 : Amigo domotic architecture

By means of the proposed architecture, any domotic device, classified in any Amigo domotic device class, can be discovered via Amigo service discovery interoperability; thus, the services offered by the domotic device can be used in the environment. We provide a common interface for domotic services (e.g., UPnP device interface) accessible within Amigo, independently of the physical devices' low-level features and communication protocols. From the Amigo application point of view, it is not necessary to know if an application is actually accessing an EIB, EHS or BDF lamp, because it just sees an Amigo service enabling control of a lamp, not an EIB, EHS or BDF lamp. The bus controllers, proprietary and discoverable device factories, as described in Deliverable D2.1, support this common interface of domotic services. We shall also stress that discoverable device factories can be implemented to enable not only a standard SDP (e.g., UPnP), but several ones. We may choose to develop a UPnP-related factory or a SLP-related one, or both of them.

A domotic bus is a subsystem that allows interaction amongst several domotic devices that support the corresponding bus protocols for communication and discovery and are connected to the bus physical layer (e.g. twisted pair, power line, radio...). Thus, a BDF device is a domotic device that can be connected to the BDF physical layer (power line) and that supports the BDF protocols for discovery and communication. A bus controller is responsible for enabling the connection with a particular domotic bus (BDF, EIB, EHS...), sending and listening to messages on the bus. This architecture is easily extensible to support new devices and buses by means of adding new bus controllers (if a new bus system must be supported) or updating the factories (if a new device must be supported).

4.4.2 Early design and implementation

In this section, we provide a description of the early design and implementation of domotic domain interoperability. In Section 4.4.2.1, we provide an overview of the design. Then, in

Section 4.4.2.2, the design and prototype implementation are detailed using UML class and sequence diagrams: the details of each class and its relations with the other classes are surveyed to provide a better understanding of the technical details of the prototype implementation.

4.4.2.1 Overview

In this section, we provide an overview of domotic interoperability within the Amigo interoperable middleware core, which makes any domotic device discoverable by components that use service discovery/interaction protocols. The following subsections present the domotic architecture components (Bus Controllers, Proprietary Device Factories and Discoverable Device Factories) needed to obtain the required final result: produce Discoverable Device Proxies for domotic devices. For our early design, two device classes are addressed representing the most common domotic devices: Amigo Legacy Devices and Amigo Base Devices; further, the UPnP protocol has been chosen as the employed Service Discovery Protocol.

Amigo Legacy Devices

An Amigo Legacy Device is a very simple domotic device that is not integrated in a domotic bus; thus, it does not need any domotic bus support. As it is a rather isolated element, communication with this device will be based on proprietary protocols with a strong dependency on manufacturer technologies.

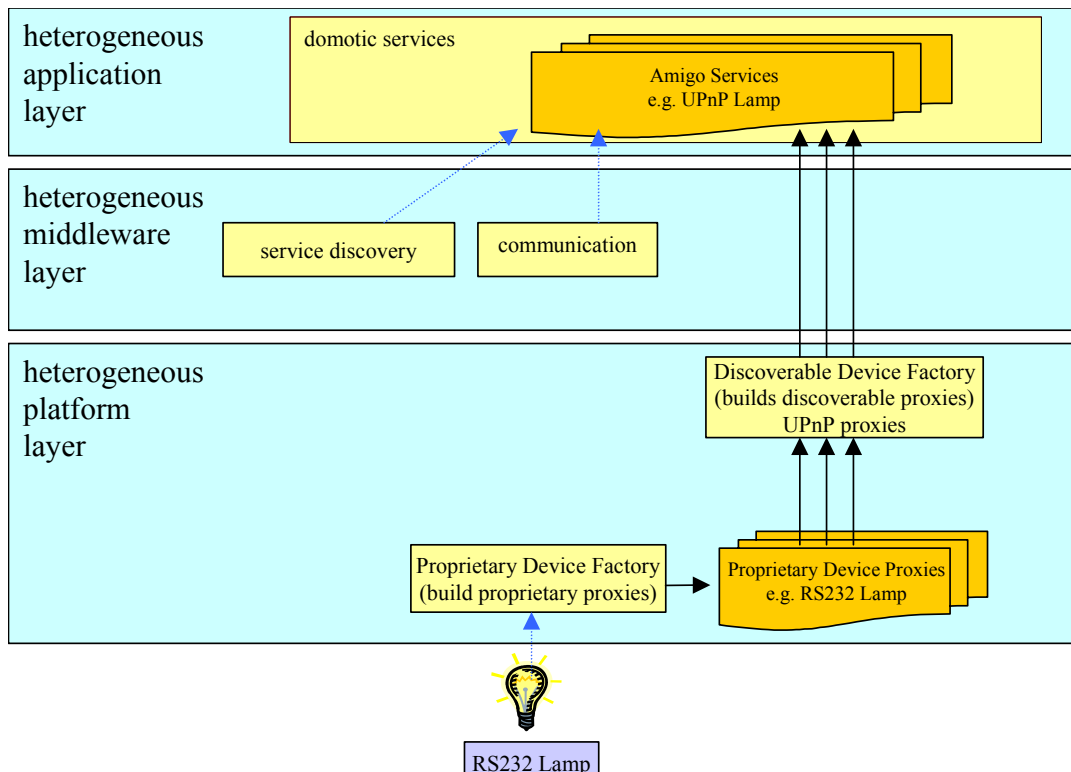


Figure 4-40: Amigo Legacy Device Architecture

Since a Legacy Amigo Device is not connected to a domotic bus, it does not need a bus controller. The Proprietary Device Factory builds the Proprietary Proxy of the physical device, and the Discoverable Device Factory builds a Discoverable Proxy (e.g., UPnP proxy) from the

Proprietary Proxy (See Figure 4-40). Amigo service discovery can discover this UPnP proxy; thus, we have achieved our goal to make this class of devices available in the Amigo environment.

Amigo Base Devices

An Amigo Base device is any domotic element that is integrated in a domotic bus. In order to be able to integrate bus-dependent devices into the Amigo system, it is necessary to provide domotic bus support in the Amigo architecture. Some of the existing buses can discover installed devices, but not by using standard SDPs; thus, the Amigo interoperable middleware core cannot directly discover the services offered by an Amigo Base Device. Figure 4-41 depicts a typical BDF infrastructure, where BDF domotic devices are connected to the home Power Line.

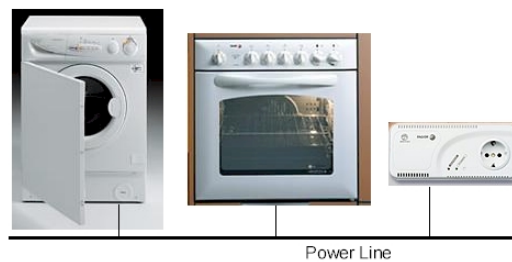


Figure 4-41: BDF infrastructure

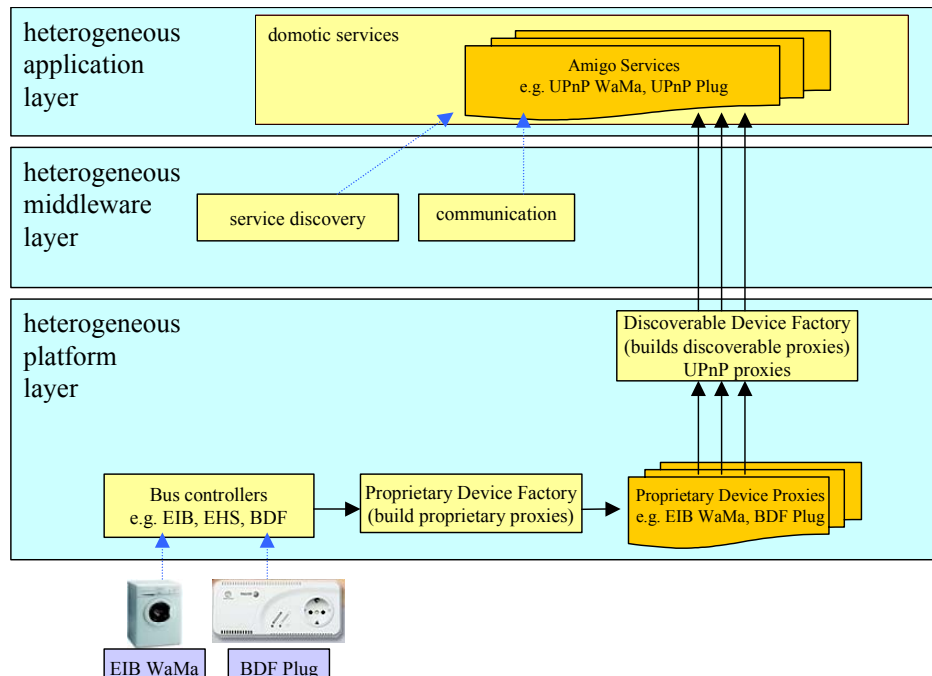


Figure 4-42: Amigo Base Device Architecture

In the Amigo domotic architecture, domotic bus support is provided by a Bus Controller that listens to the bus. When a physical device on the bus is detected by listening to the corresponding bus messages, the Proprietary Device Factory builds the Proprietary Proxy of the device, and the Discoverable Device Factory builds a Discoverable Proxy from the Proprietary Proxy. Again, in this way, we have made this class of devices available in the Amigo environment.

4.4.2.2 Detailed description

Amigo Legacy Devices

A lamp controlled via RS232 interface will be provided as an example of a legacy device (see Figure 4-43). It is connected to a serial port of a PC. Software running on this PC will detect the connected device and instantiate the corresponding UPnP proxy to access the lamp. In the same way, if the lamp is disconnected, the UPnP device will be removed. By means of this UPnP proxy the lamp can be switched on and off remotely from any UPnP control point in the network.

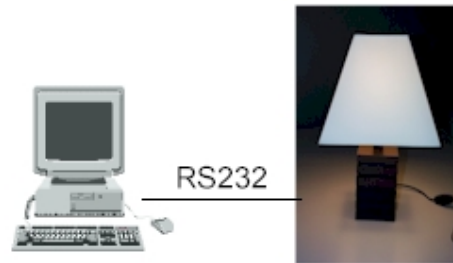


Figure 4-43: RS232 Lamp

Figure 4-44 shows the class diagram used to integrate this Amigo Legacy Device into the Amigo interoperable middleware core, and the corresponding UPnP proxy generation. The *RS232Driver* class is responsible for the communication via RS232, checking the connection and sending messages to the lamp. The *LampModel* class is a model of the physical lamp and represents its current status. The *Monitor* class has the responsibility of periodically checking that the lamp is connected and updating its status when requested. The *UPnPLamp* class is a *UPnPDevice* that acts as a UPnP proxy of the physical lamp.

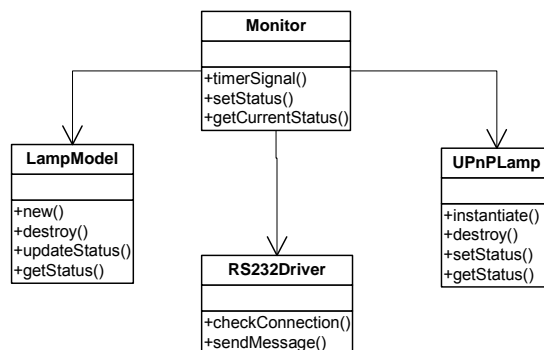


Figure 4-44: RS232 Lamp Class Diagram

Figure 4-45 shows the sequence diagram followed in order to discover a connected RS232 lamp. The *lampMonitor* is periodically (1) checking the RS232 port (2), waiting for a response from the lamp. When it receives a positive response (*ok*), it instantiates a new *LampModel* object (4) and the corresponding *UPnP*Lamp proxy (5).

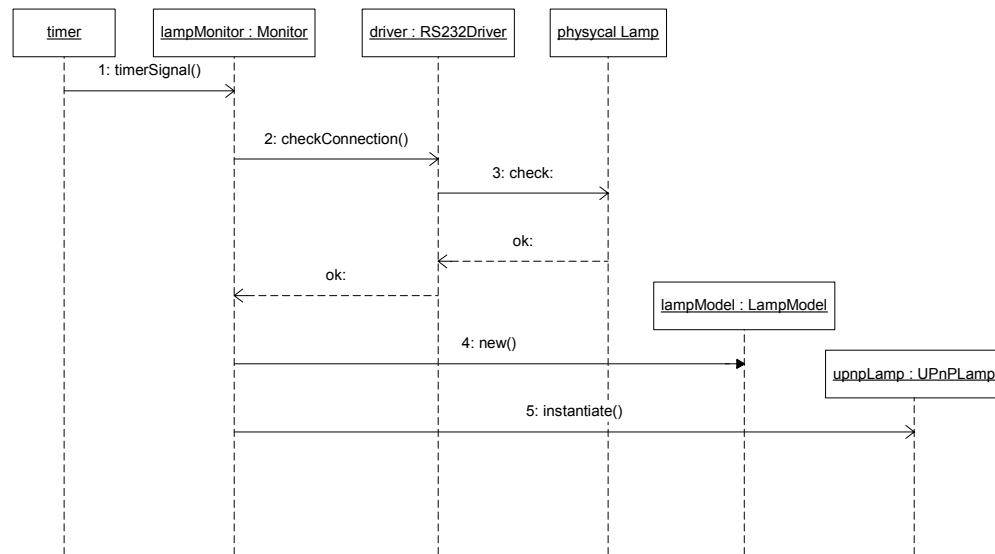


Figure 4-45: Lamp discovery sequence diagram

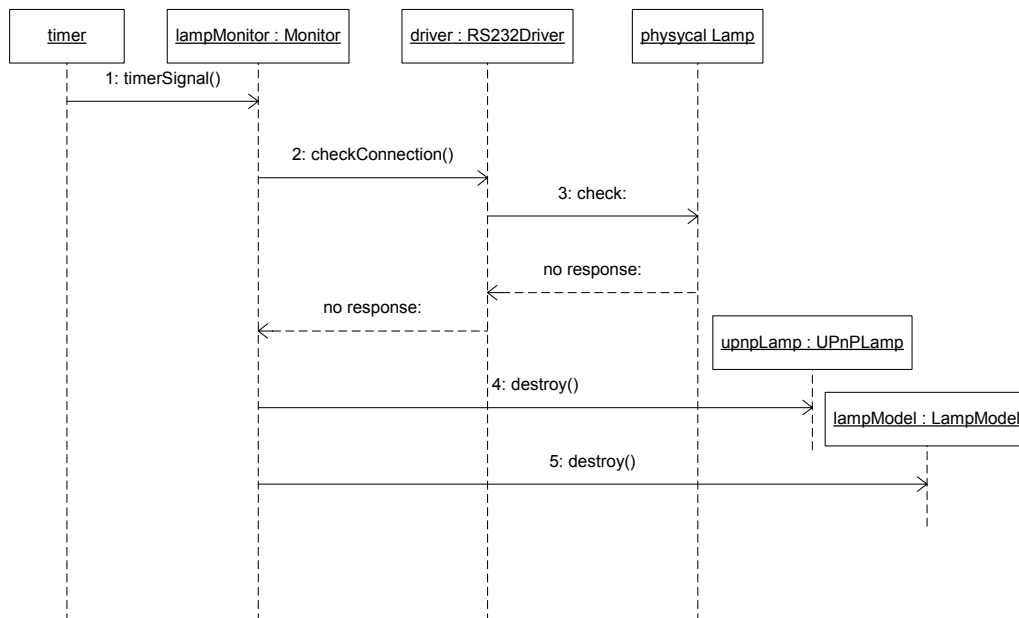


Figure 4-46: Lamp removal sequence diagram

The sequence depicted in Figure 4-46 describes how the UPnP proxy is destroyed when the physical lamp is disconnected.

Figure 4-47 shows how the lamp can be switched on and off from an external UPnP control point. When the *upnpLamp* object receives the *setStatus* message from a control point (1), the *lampMonitor*'s *setStatus* method is invoked (2). This one tries to switch on/off the lamp according to the received message by means of the *RS232Driver* (3) which sends the adequate message to the physical lamp (4). Only if the response from the lamp is *ok*, the *lampModel* is updated with the new status. Thus, the lamp model always represents the current status of the physical lamp.

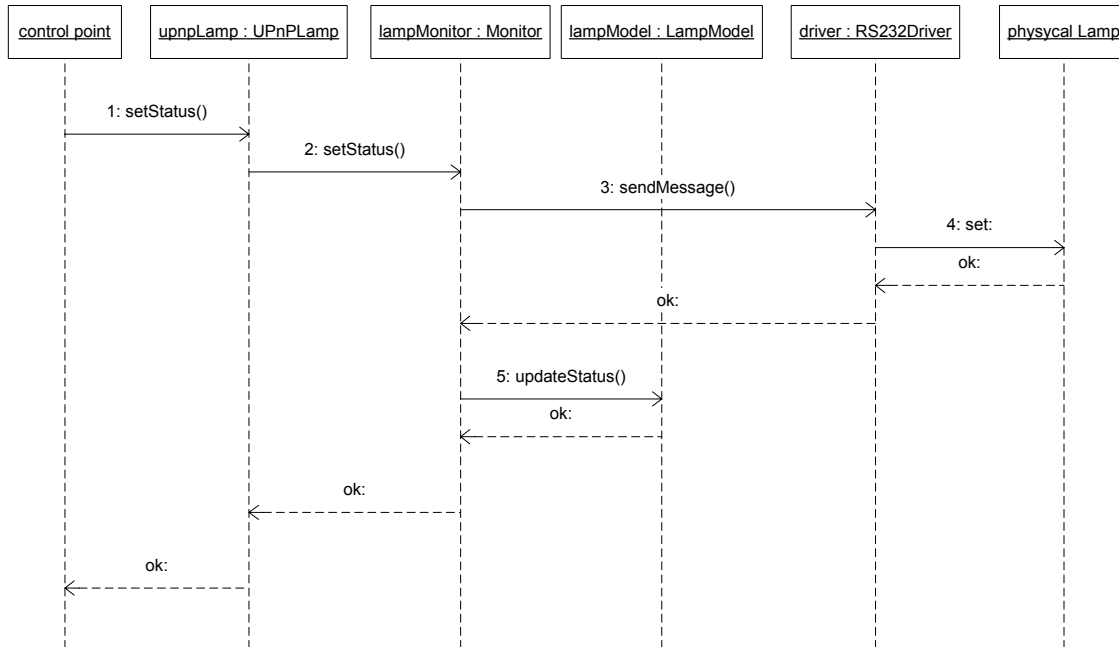


Figure 4-47: Action invocation sequence diagram

Amigo Base Devices

As an example of Base Device, a BDF device has been chosen; concretely a BDF Plug (See Figure 4-48).



Figure 4-48: BDF Plug

A BDF Plug consists of a domestic *Schuko* receptacle (see Figure 4-49) that, when plugging any electrical device (i.e. coffee machine, lights...) in it, allows to connect/disconnect the plugged device remotely. The connection start and end time can also be scheduled.

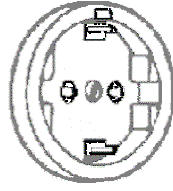


Figure 4-49: Schuko receptacle

A set of software components running on a PC will be in charge of listening to the BDF bus (power line), detecting the connected device (the BDF plug), and instantiating the corresponding UPnP proxy to access the BDF device. Using this UPnP proxy, the plug can be discovered and accessed by other services.

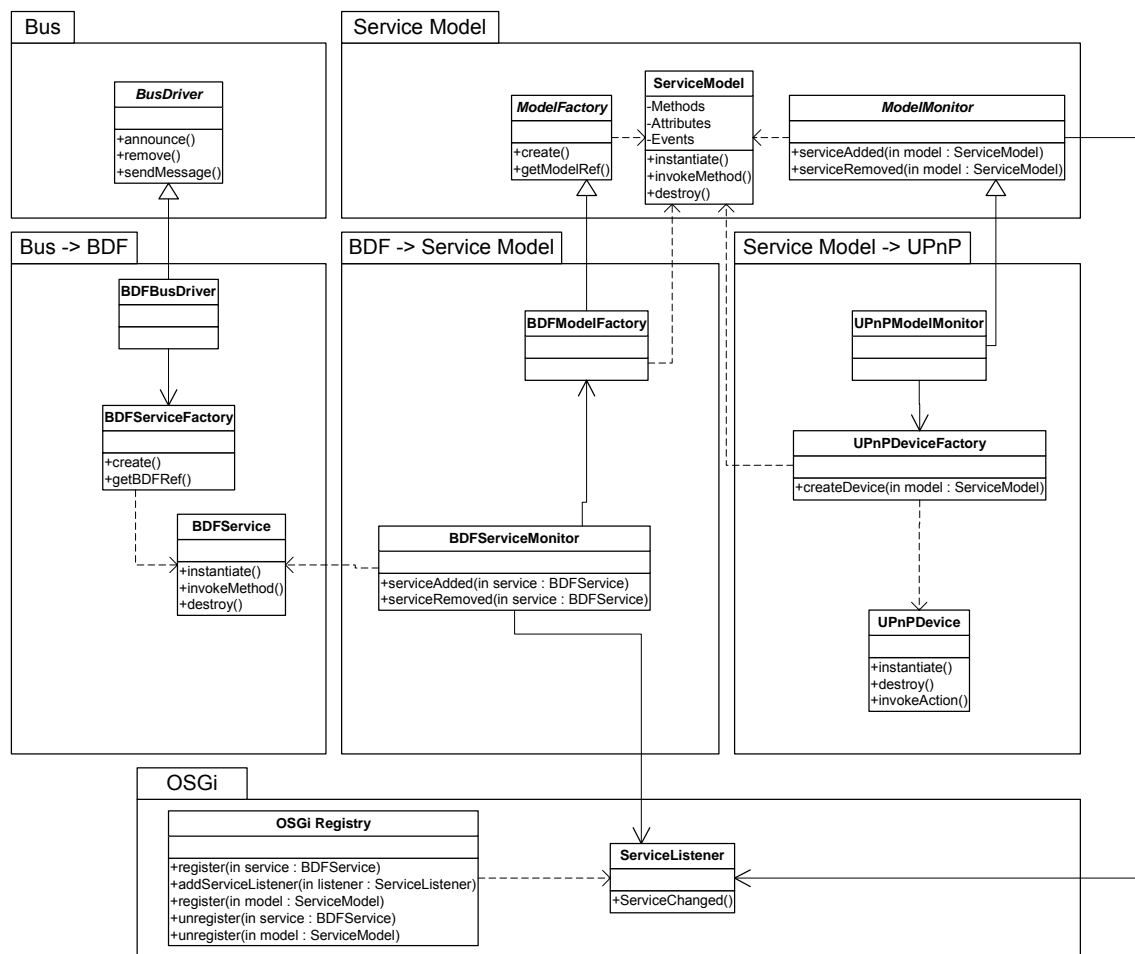


Figure 4-50: Class diagram for BDF support

Figure 4-50 shows the class diagram used to implement the BDF support and the UPnP device proxy generation.

The base class *BusDriver* defines the basic methods and provides the basic functionalities that are necessary to listen and speak to a generic domotic bus. The *BDFBusDriver* class specializes the base class by implementing the methods for receiving announcements and

goodbyes from devices in the BDF bus, and for sending BDF messages to any element in the bus. The *BDFService* class represents the functionality of any BDF device. The *BDFServiceFactory* class will create the necessary instances of *BDFService* in accordance with the devices present in the bus. Thus, using these classes, *BDFService* instances are obtained from bus messages.

Another class, *BDFServiceMonitor*, attends to newly registered or unregistered *BDFService* instances in a service registry. For the current prototype, the OSGi framework will be used, so the mentioned components will be implemented as OSGi bundles, and BDF services will be registered in the OSGi registry (as an implementation of the service registry). The OSGi service registry provides a comprehensive model to share objects between OSGi bundles. The OSGi specification defines an event (*ServiceChanged*) to handle the coming and going of services from the OSGi registry, enabling dynamic notification of newly registered and unregistered services. Services are just Java objects that can represent anything (in this case, *BDFService* instances).

The *ServiceModel* class represents a generic service and enables describing the methods, attributes and events, if any, of a specific service. The *ModelFactory* class is a base class and is responsible for instantiating generic *ServiceModel* objects from specific services. The *BDFModelFactory* class, a specialization of the *ModelFactory* class, implements the methods to obtain *ServiceModel* instances from *BDFService* instances. The *ModelMonitor* is a base class to receive notifications of added or removed *ServiceModel* instances in a service registry (OSGi registry).

The *UPnPDeviceFactory* class uses *ServiceModel* objects to instantiate *UPnPDevice* instances. *UPnPModelMonitor* specializes the *ModelMonitor* class and, when it receives a new *ServiceModel* registration notification, asks the *UPnPDeviceFactory* to instantiate the corresponding *UPnPDevice* object. In a similar way, when the *ServiceModel* is unregistered, the *UPnPModelMonitor* receives the appropriate notification and the previously created UPnP device is destroyed.

Figure 4-51 shows the sequence diagram that represents the process from the announcement of the domotic Plug in the BDF bus to the UPnP device generation. First of all, both monitors, *BDFServiceMonitor* and *UPnPModelMonitor*, subscribe themselves to the service registry to receive notifications of registrations and unregistrations (1 and 2). When the newly installed Plug sends a message to the bus advertising itself according to the BDF protocol, the *bdfBusDriver* object receives a message (3) with the information about the new device. By means of a *BDFServiceFactory*, it gets an instance of a *BDFService* to access the physical device (4 and 5), and registers it in the service registry (6). Then, the service registry notifies to its subscribers (*bdfMonitor*) that a new *BDFService* has been registered (7), and the *BDFMonitor*, using a *BDFModelFactory*, gets a *PlugModel* instance of a generic service representing the installed Plug (8 and 9). This generic service is also registered in the service registry (10). As *upnpModelMonitor* is notified that a new *ServiceModel* (*plugModel*) has been registered, by means of a *UpnPDeviceFactory*, it instantiates a new *UPnPDevice* (*upnpPlug*) from the generic *ServiceModel*.

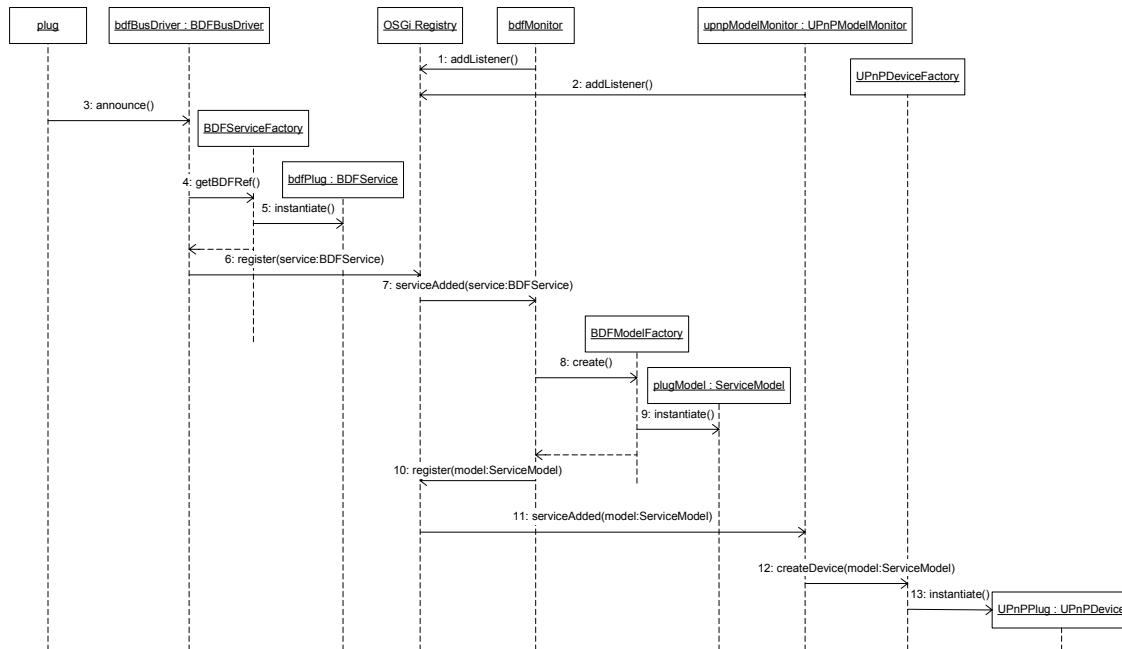


Figure 4-51: Device presentation sequence diagram

Figure 4-52 depicts the process followed when a device is removed from the bus. When the plug is removed, the *bdfBusDriver* object receives a message (1) about the device removal. So, it unregisters the previously instantiated *BDFService* from the service registry (2) and destroys it (5). *BDFMonitor* is then notified that a *BDFService* (*bdfPlug*) has been unregistered (3) so, following the same process, it unregisters the *plugModel* (4) and destroys it (5). In a similar way, *upnpModelMonitor* is then notified about the *ServiceModel* unregistration and destroys the *UpnpDevice*.

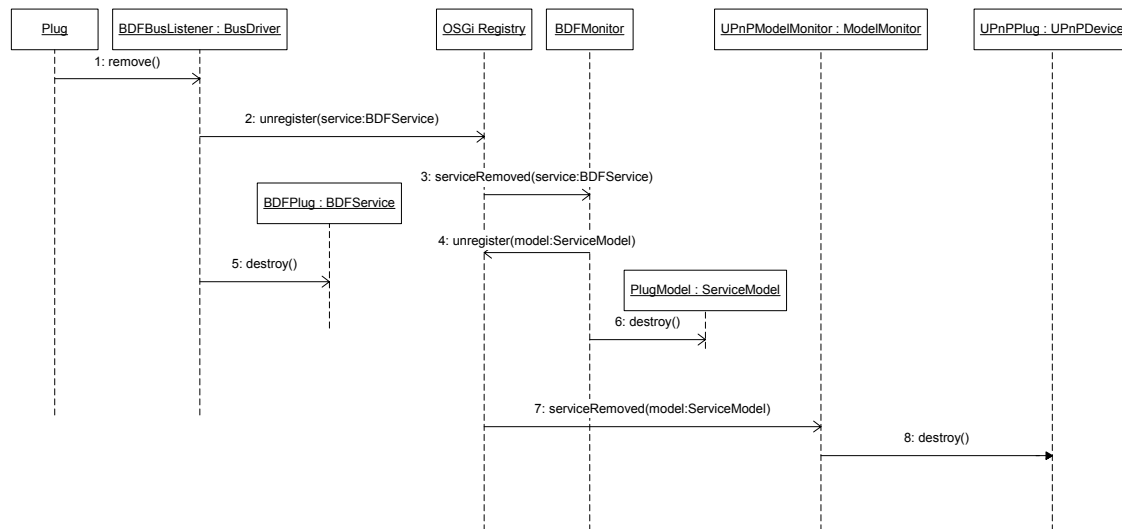


Figure 4-52: Device removal sequence diagram

Figure 4-53 shows the sequence followed by an action invocation in which a UPnP Control Point sends a SOAP message specifying the action to be invoked to the UPnP Device. When an action is invoked on the *upnpPlug* (1), the corresponding method is invoked on the *plugModel* (2). Then, the latter invokes the *BDFService* method (3), which asks the *BDFBusDriver* to send the appropriate message to the BDF bus; this message is finally received by the physical Plug. The return values, if any, follow the opposite way.

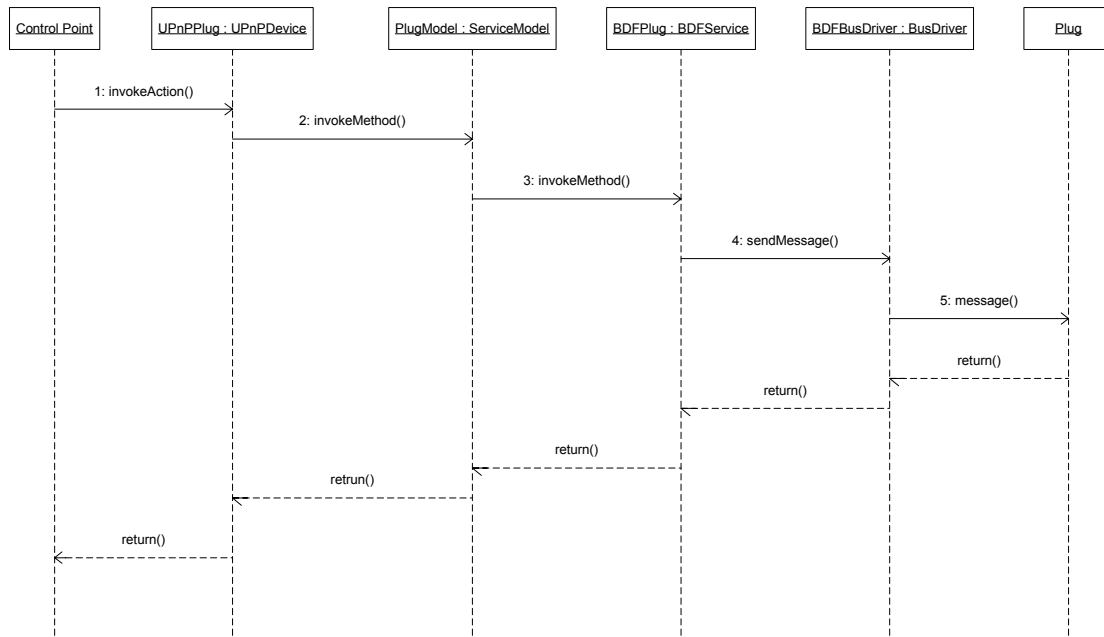


Figure 4-53: Action invocation sequence diagram

4.5 Consumer Electronics interoperability

This section describes the interoperability mechanisms applied to realize Consumer Electronics (CE) domain interoperability. Initially, we recall from Deliverable D2.1 [Amigo-D2.1] the main areas in the CE and multimedia domain in which interoperability problems lie (Section 4.5.1). Then, we provide our architecture for QoS interoperability that refines and finalizes the abstract CE architecture defined in Deliverable D2.1 (Section 4.5.2).

4.5.1 Overview

Nowadays, most CE and multimedia interoperability efforts are directed towards the adoption of a set of standards ensuring compatibility between devices by enforcing compliance with the appropriate standard with respect to functionality (e.g., UPnP AV for content distribution and rendering control, MPEG4 as codec standard, etc.). Different efforts towards CE and multimedia streaming interoperability include among others the Digital Living Network Alliance (DLNA), the Consumer Electronics Association (CEA), the Internet Streaming Media Alliance (ISMA) and the Home Audio/Video Interoperability (HAVi). The Amigo interoperable middleware shall support the different convergence standards and provide interoperability between them. There are five main areas in the CE domain and multimedia in which interoperability problems lie:

- Service discovery and interaction: the mechanisms via which CE devices make their services known to and accessed by the networked environment;
- Streaming protocols: the protocols that actually transport content and related signaling;
- Content Formats: the way in which content is coded and presented;
- Content Description: the mechanisms that enable identifying and finding content;
- Quality of Service: the technologies used by CE devices that enable a guaranteed quality of service in the communication between two CE devices or a CE device and another device.

In the following, we discuss in more detail these five areas.

Service discovery and interaction

Different standardisation institutions and previous projects have addressed this problem providing state of the art analysis of the different service discovery protocols available. After an analysis of the different standardisation institutions' and alliances' proposals, it is obvious that there is a small set of discovery (and in some cases interaction) protocols widely accepted, some with more, some with less popularity in the CE domain: mainly SLP, UPnP, Jini, Bluetooth SDP, WS-D and Salutation.

Service discovery and interaction interoperability in the CE domain must be resolved (see DMS and DMR mapping to the Amigo abstract architecture in Deliverable D2.1 [Amigo-D2.1]). UPnP is the most popular standard (recommended by DLNA, CEA) within this domain. Interoperability between this standard and the other standards mentioned above is addressed in the service discovery and interaction interoperability sections of this document (respectively Sections 4.1 and 4.2). In the case of CE legacy devices, a proxy will provide an intermediate device interface to the network and probably a standard audio or video connection (e.g., RCA, Euroconnector) to the legacy device (e.g., Set Top Box) enabling discovery and interaction with the legacy device.

Streaming protocols

Analysis of the interoperability in the CE domain within the context of streaming protocols concerns studying which protocols are the main ones and how they are used in multimedia streaming. Deliverable D2.1 presented the abstract Amigo architecture for DMR and DMS; availability of streaming protocols is required on both entities.

Some streaming technologies such as RealAudio and Windows Media utilize dedicated servers that support superior UDP and RTSP transmission. Other formats such as Shockwave, Flash, MIDI, QuickTime and Beatnik are primarily designed to stream from a standard HTTP Web server. It is clear that HTTP and RTSP are the main streaming protocols used by CE devices and will be the ones to take into account in Amigo. RTSP uses a combination of reliable transmission over TCP (used for control) and best-effort delivery over UDP (used for content) to stream content to users. HTTP streaming uses only TCP and is thus referred to as pseudo-streaming, since technically it is possible to stream via HTTP. But HTTP is much likely to cause major packet drop-outs, and it cannot deliver the same amount of streams as UDP and RTSP transmission. Herein lies the difference between most low-end solutions and more professional broadcasting solutions that require dedicated servers and extra bandwidth and server capacity.

Interoperability at this level may be attacked with two possible approaches:

- Provide interoperability methods for transforming HTTP requests to RTSP ones and vice versa. In fact, UPnP AV supports this kind of proxies [UPnP CD].
- Resolve interoperability at platform level, requiring either or both a streaming client and server for each different technology to be implemented on a device.

Considering real-time requirements of streaming, the second approach is a much more reasonable one and can provide a solution for streaming interoperability between CE devices in Amigo. We will not address this issue in WP3, as it is rather integration work than research work.

Content Formats

Interoperability in the multimedia domain further concerns codecs and file-formats. The DLNA guidelines specify that all devices claiming support of some media type must support a specified codec/format of this media type (LPCM for audio, JPEG for images, and MPEG2 for video), in order to ensure that all devices supporting a given media type are compatible (see Deliverable D2.1). However, this is not possible for certain devices with, for example, limited memory (e.g., mobile domain), so some type of format adaptation may be necessary. Since this mainly concerns content distribution, it is tackled as part of the content adaptation issue discussed in Deliverable D3.1c [Amigo-D3.1c].

Content Description

An analysis of interoperability in the CE domain within the context of content distribution is required. DLNA guidelines define UPnP AV services as the standard for content browsing and transfer management between devices, as well as for content rendering control. Other interoperability institutional efforts either do not address directly this point or adopt UPnP as standard at the time of writing. Thus, CE devices implementing the UPnP AV services will interoperate within these functionalities seamlessly. However, heterogeneity of content descriptions poses an added problem. DLNA guidelines establish, through UPnP AV, DIDL Lite, based on an extended set of Dublin Core, as the content description standard. However, an application performing content navigation may use any of the other popular content description schemes (such as MPEG-7 Description Definition Language, included in the TV Anytime specifications). Interoperability between applications and devices using different content descriptions must be achieved in order, not only to assure access to the content available at home and outside the home from different devices offering different content navigation technologies, but also to avoid to a certain extent content duplication. Since this mainly concerns content distribution, it is tackled as part of the content adaptation issue discussed in Deliverable D3.1c [Amigo-D3.1c].

Quality of Service

Access networks, which have been improved by the innovation in broadband access technology and the investment in access infrastructure such as copper enhancements (ADSL, SDSL, and VDSL), Fiber-To-The-Home (FTTH), and Wireless Local Loop (WLL), still cannot match the bandwidth inside the home. While the latter can be over 400 Mbps (e.g., IEEE 1394), the one of the access network is generally below 2 Mbps. Although currently the Internet offers a straightforward best effort delivery service, where there is no commitment to bandwidth or latency for senders, more and more delay-sensitive services are being developed and deployed on the Internet, requiring QoS. Therefore, both domains, the Internet and the home network, tend to adopt QoS techniques as a reaction to the emergence of more demanding applications and services (e.g., Video on Demand and videoconference).

Current Layer 2 home network technologies only support packet priority QoS. This is not a disadvantage since prioritized traffic: is compatible with QoS-unaware networks; works with dynamic data rates; there are many standards available; is easy to manage; has low overhead; and supports simultaneous services. Moreover, when interoperability is the objective, a priority-based QoS architecture can accommodate QoS-unaware devices and applications. DLNA will probably adopt UPnP QoS as the QoS interoperability standard. For

the time being, UPnP QoS regards mainly packet-priority-based QoS, but using generalised levels of priority that decouple it from the actual Layer 2 priority scheme.

However, there are other more mature QoS technologies, present on the Internet and other public and private networks (e.g., RSVP). Interoperability between these technologies and the CE interoperability standards shall be provided to assure QoS of multimedia streaming from sources outside the Amigo home. Deliverable D2.1 introduced QoS support as a requirement for the middleware, which was directly derived from the existence of multiple heterogeneous streams with different QoS demands in the home network. There, the common solution of providing interoperability by enforcing compliance with the appropriate standard (DLNA directives in this case) was presented. However, we consider that providing a general interoperability mechanism that would adapt easily to appearing QoS technologies, and that provides immediate interoperability between two existing ones of a completely different nature, is a step further towards the project objectives. Our choice is further reinforced by the recommendation of Chapter 10: Assessment of Amigo interoperability of D2.1 identifying the need for a QoS interoperability mechanism between the QoS standards of the in-home network and the ones of networks outside the home. In the present chapter, we elaborate our approach addressing this issue.

4.5.2 Refined Architecture

Having as a starting point the Amigo abstract multimedia streaming architecture with all its components as introduced in Deliverable D2.1 (see Figure 4-54), along with the analysis of the interoperability issues presented in the previous section, the next step is to refine these components that will be implemented in the Amigo interoperable middleware:

- **DRM:** The DRM management will be addressed within middleware modules related to Security and Privacy (Task 3.5);
- **Accounting and Billing:** The management of accounting and billing will be addressed in the related Task 3.8;
- **Content Management and Content Storage:** This module is currently being addressed in the related Task 3.7 (see Deliverable D3.1c [Amigo-D3.1c]).
- **Service Discovery and Message Communication Protocols:** These services will use the common service discovery and interaction mechanisms addressed in Sections 4.1 and 4.2 of this document.
- **Streaming Protocols and Streaming Session Control Protocols:** We discussed these in the previous section: they will not be addressed in WP3.
- **QoS Support:** This module will be addressed in this section, according to the discussion of the previous section. We will provide interoperability between Internet QoS technologies and CE QoS standards. As indicated, the two state of the art technologies outstanding in their respective domains are UPnP QoS and RSVP, so our interoperability solution between access network and home network QoS technologies will use these as a starting point. These two technologies are essentially different: the first provides QoS inside a local area network on a packet class differentiation basis; the second provides QoS on Internet connections on a 'per flow' reservation basis. Section 4.5.2.1 provides the background for our QoS interoperability solution, which is presented right then (Section 4.5.2.2).

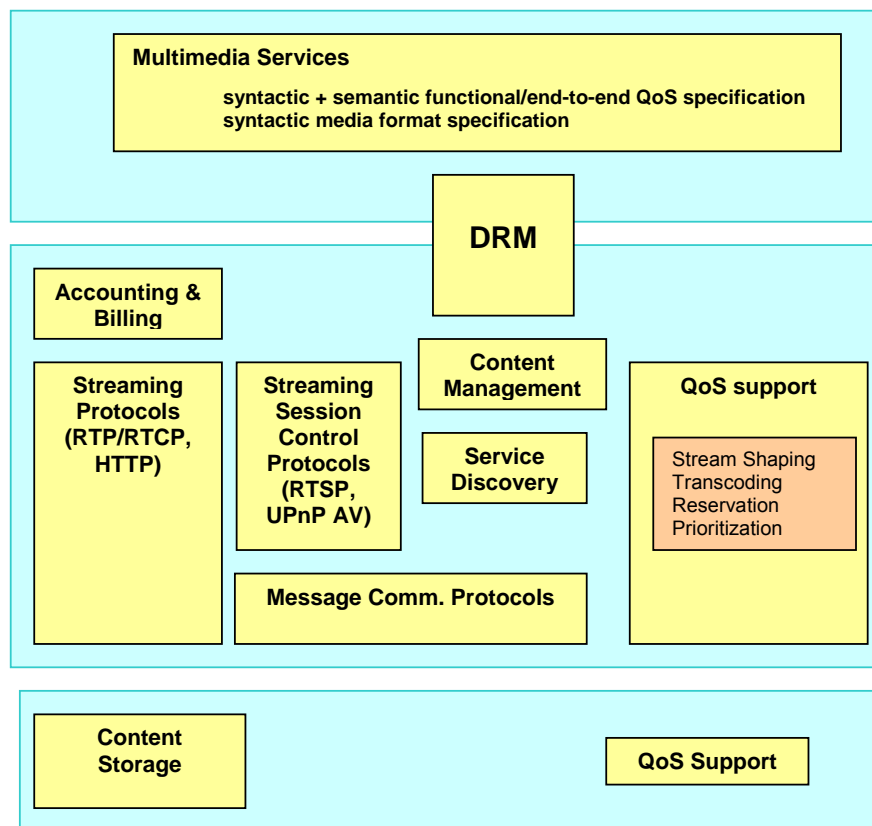


Figure 4-54: Amigo abstract Multimedia Streaming Architecture

4.5.2.1 Background

UPnP QoS

A base study of UPnP QoS was presented in Deliverable D2.1. We summarize in this subsection, the main UPnP QoS operational issues of interest from an interoperability point of view.

A UPnP QoS scenario comprises the following elements (see Figure 4-55):

- **QoS Policy Holder:** Contains the criteria that will be applied upon the assignation of QoS to the streams. Traffic streams are classified using priorities established according to the traffic and user importance.
- **QoS Manager:** Composed of a QoS Manager Service and a QoS Management Entity, represents the QoS front-end to the Control Point.
- **QoS Devices:** The network devices that will carry the data streams.

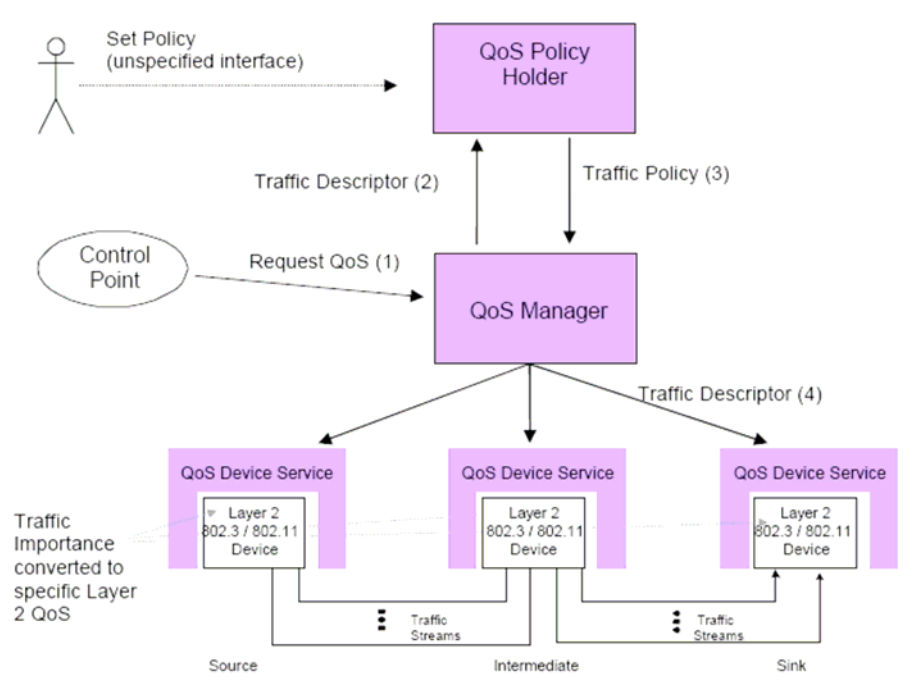


Figure 4-55: UPnP QoS scenario (see [UPnPQoS])

For uniformity with the UPnP QoS specifications issued by the UPnP Forum ([UPnP]) this document uses the same terminology as that in the original documentation for referencing actions. This is “Entity_name : action_name”, with the following acronyms:

QoS : Quality of Service

QM : QosManager

QD : QosDevice

The UPnP QosManager will be in charge for setting up and controlling the QoS assigned to a certain flow of data, acting as a Control Point to UPnP QosDevice services.

The general operation procedure is started by the QosManager discovering the QosDevice services available on the network. Once first step is accomplished, it can get information about the device by using the actions “QD:GetQosDeviceCapabilities” and “QD:GetQosState” offered by the QosDevice. The first action returns a new parameter called “QosDeviceCapabilities” detailing the capabilities and the network interfaces present on the devices, while the second returns the current state of the device as well as the parameters “QosDeviceState”, “NumberOfTrafficDescriptors” and “ListOfTrafficDescriptors”.

Qos Policy Holder services must be discovered by the QosManager, expecting only one of them to be present. In case none or more than one are discovered, a default policy (FCFS) is assumed by the QosManager.

In a UPnP AV scenario any application that acts as a control point can require the QosManager services to set up the desired QoS for a given data stream.

First of all, the control point requires the “QM:RequestTrafficQoS” to call for the QosManager service set up, using the “InitialTrafficDescriptor” as a parameter, receiving “TrafficHandle”, “NumPolicyHolders” and “UpdatedTrafficDescriptor” as response. Every “TrafficDescriptor” contains the following parameters:

- Tspec: contains a description of Content QoS Requirements. The XML structure consists of "TspecIndex" an identifier that also indicates preference (smaller equals higher), "AVTransportUri", "AVTransportInstanceID" identifier associated with the AV Transport service associated with the content item for which QoS is requested (see [UPnPAV]), and "TrafficClass" associated with the traffic stream.
- TrafficHandle: unique identifier associated with a particular traffic stream (or instance of TrafficDescriptor).
- AvailableOrderedTspecList:
- ActiveTspecIndex: indicates the index of the current active Tspec from the TspecList.
- QosBoundarySourceAddress: address from which the QoS traffic stream enters the UPnP network when it originates outside the home network. Used by the QosManager for path determination.
- QosBoundaryDestinationAddress: termination point address for UPnP QoS when the traffic stream terminates outside the home network. Used for path determination.
- TrafficImportanceNumber: integer indicating the priority of a traffic stream according to the traffic type. Highest equals more priority.
- TrafficID: contains information for classification and identification of packets, containing the following: SourceIP, DestinationIP, SourcePort, DestinationPort, Protocol.
- UserName: user who requested a traffic stream.
- CpName: control point that requested the traffic stream.
- VendorApplicationName: single URI string associated with the application initiating the UPnP QoS Action.
- PortName: URI string associated with the port used by an application for its connection set-up.
- ServiceProviderServiceName: URI string identifying a service offered by a service provider.
- TrafficLeaseTime: lease-time associated with a particular traffic stream expressed in seconds.
- MediaServerConnectionId: optional field used to identify a traffic stream that is being setup by the UPnP AV CP requesting QoS when it is streaming multiple Tspecs. Obtained via the "CM::PrepareForConnection" action.
- MediaRendererConnectionId: see MediaServerConnectionId

Any unknown arguments may be left blank and therefore filled after the traffic QoS is implemented in the QosDevice. Parameters that may also be included by the control point are "UserName", "CpName", "VendorApplicationName", "PortName" and "ServiceProviderServiceName".

Information available in the QosPolicyHolder may be used for retrieving the "TrafficImportanceNumber" and "UserImportanceNumber" for the current traffic stream.

The control point invokes the "QM:RequestTrafficQos" action providing the "InitialTrafficDescriptor" and obtaining the parameters "TrafficHandle", "NumPolicyHolders" and "UpdatedTrafficDescriptor". This action is offered by the QosManager QoS Management Entity which queries the QosPolicyHolder to get the "TrafficImportanceNumber" (ranged from 0 to 7) and "UserImportanceNumber" (from 0 to 255).

In order to determine the devices in the path of the data stream, the QosManager invokes the "QD:GetPathInformation" action offered by the devices, obtaining the "PathInformation" parameter. It may also invoke the "QD:GetQosDeviceInfo" in order to retrieve the

“PortNumber” and “ProtocolInformation” belonging to the given “TrafficDescriptor”. Once the devices have been identified, the QoSManager QoS Management Entity calls the “QD:GetQoSState” action to each one of them retrieving the “QoSDeviceState”, “NumberOfTrafficDescriptors” and “ListOfTrafficDescriptors”. Next step consists of the “QoSManager” issuing the “QD:SetupTrafficQoS” action with the input “SetupTrafficDescriptor” and “QoSStateId” arguments to the devices in the path of the traffic. If successful, it is now when the control point receives the updated “TrafficDescriptor” argument in response to its former “QM:RequestTrafficQoS” petition. If failed, the proper error message is sent.

Updating the current QoS associated with a particular traffic implies the use by the CP of “QM:UpdateTrafficQoS” providing the “TrafficHandle” and “RequestedTrafficDescriptor” arguments and receiving the “ImplementedTrafficDescriptor” and “NumPolicyHolders”. The update process means repeating the previous admission control process using the new “TrafficDescriptor”.

Releasing such QoS is achieved through the issue of the “QM:ReleaseTrafficQoS” action offered by the QoSManager using the “RevokeTrafficHandle” as parameter. When receiving this action, the QoSManager will issue the “ReleaseTrafficQoS” action to all the devices in the path of the data flow, providing them with the “ReleaseTrafficHandle” parameter.

Whenever the path of the traffic changes, the QoSDevice updates the “PathInformation” variable and issues an event to the subscribed QoSManager. The “PathInformation” variable consists of the “LinkReachableMacs”, “LinkId”, “MacAddress”, “ReachableMac” and “Bridgeld” arguments.

RSVP

Resource ReSerVation Protocol, also known as RSVP, is an Internet protocol that enables Internet applications to request enhanced IntServ quality-of-service (QoS). It supports two classes of service:

- *Controlled load Service* - this is an attempt to provide a guarantee that a network appears to the user as if there is little other traffic - it makes no other guarantees - it is really a way of limiting the traffic admitted to the network so that the performance perceived is as if the network were over-engineered for those that are admitted.
- *Guaranteed Service* - this is where the delay perceived by a particular source or to a group is bounded within some absolute limit. This may entail both an admission test and a more expensive forwarding queuing system.

Basically, RSVP is a receiver-initiated protocol. The sending node is just to pass the requirements of the traffic to the receiver via sending a PATH message. The receiving node is responsible for initiating the resource reservation by sending back a RESV message.

In RSVP-enabled network architectures, each RSVP host will contain RSVP-aware applications, an RSVP API, a RSVP Daemon, and a RSVP protocol stack that consists of admission control, Policy control, packet scheduler and packet classifier. RSVP-aware applications send or receive data flows using RSVP in parallel: unlike legacy applications, they interact with RSVP daemon to require the QoS support from network. Each of RSVP APIs is a set of procedures used by applications to interact with RSVP daemon. In general, they are created as libraries and linked by programs at run time. Typical RSVP API is RAPI library on UNIX platforms [BH98].

The RAPI procedures consist of *rapi_session()*, that is used to initialize a session, *rapi_sender()*, to notify RSVP daemon of the sender traffic characteristics, *rapi_reserve()*, to notify RSVP daemon of the reservation parameters, and *rapi_getfd()* as well as *rapi_dispatch()*, those are used together to receive notification of events (see Figure 4-56). The RSVP daemon is responsible for handling the RSVP signaling. It must be able to deliver

RSVP messages to the network and all RSVP messages received by the host must be passed to it.

In the receiver, an RSVP agent application waits for PATH messages from sender. It extracts QoS parameters from PATH message, and send RESV message back with the readjusted QoS parameters. The sender firstly delivers the QoS request, then the receiver determines to follow the sender's proposal or readjust the parameters, depending on the receiver network environment. The RSVP Agent of the receiver first uses RAPI to create a RSVP session and register a callback function. When RSVP daemon in receiver receives a PATH message, it triggers a RAPI_PATH_EVENT event. Then the Upcall (callback) function registered is called to parse the PATH messages in order to get QoS parameters such as Tspec, token rate, bucket depth, etc. Finally, the RSVP Agent uses the RAPI to send RESV messages back with the receiver application's QoS parameters need.

Therefore all of the hosts in the client side must active the RSVP daemon to handle the RSVP message events. The application and the RSVP daemon establish a Unix socket connection to exchange the information; this allows to trigger and handle the PATH and RESV events. Finally, RSVP daemon determines when to deliver PATH and RESV messages, and communicate with the underlying layer for bandwidth reservation information to establish the QoS connection. By calling RSVP API, the Internet application can create, maintain and release a QoS connection, and forward any error information to the application.

The parameters interchanged in the use of rapi functions are:

RAPI Session

An application calls this routine to define an API session for sending or receiving, or both, a single simplex data flow. The RAPI session routine define de IP - Port address, protocol IP and the Event Return parameter. The latter points to an upcall routine that is invoked to notify the application of RSVP errors and state change events. Pending events also cause the invocation of the upcall function when dispatched.

RAPI Sender

An application calls this routine to register as a data sender. The RAPI sender specifies the session handle that was returned by a successful call to the **rapi_session()** routine. Also it specifies several RSVP objects like:

- **Sender Template:** Points to a RSVP API (RAPI) filter specification structure that specifies the format of data packets to be sent, or is NULL. This template is in the form of a *filter spec* used to select sender's packets from others in the same session on the same link. The *sender template* has the same power and format as the *Resv message's filter spec*. It specifies the *sender IP address* and optionally the *UDP/TCP sender port*. This is an optional parameter.
- **Sender Tspec:** Points to a Tspec that defines the traffic characteristics of the data flow the sender will generate. This information is used to prevent *over-reservation* and *Admission Control* failures
- **Sender Adspec:** Points to a RAPI Adspec structure, or is NULL. This information is passed to the local control traffic, which returns it updated; the updated version is then forwarded downstream in *Path* messages.

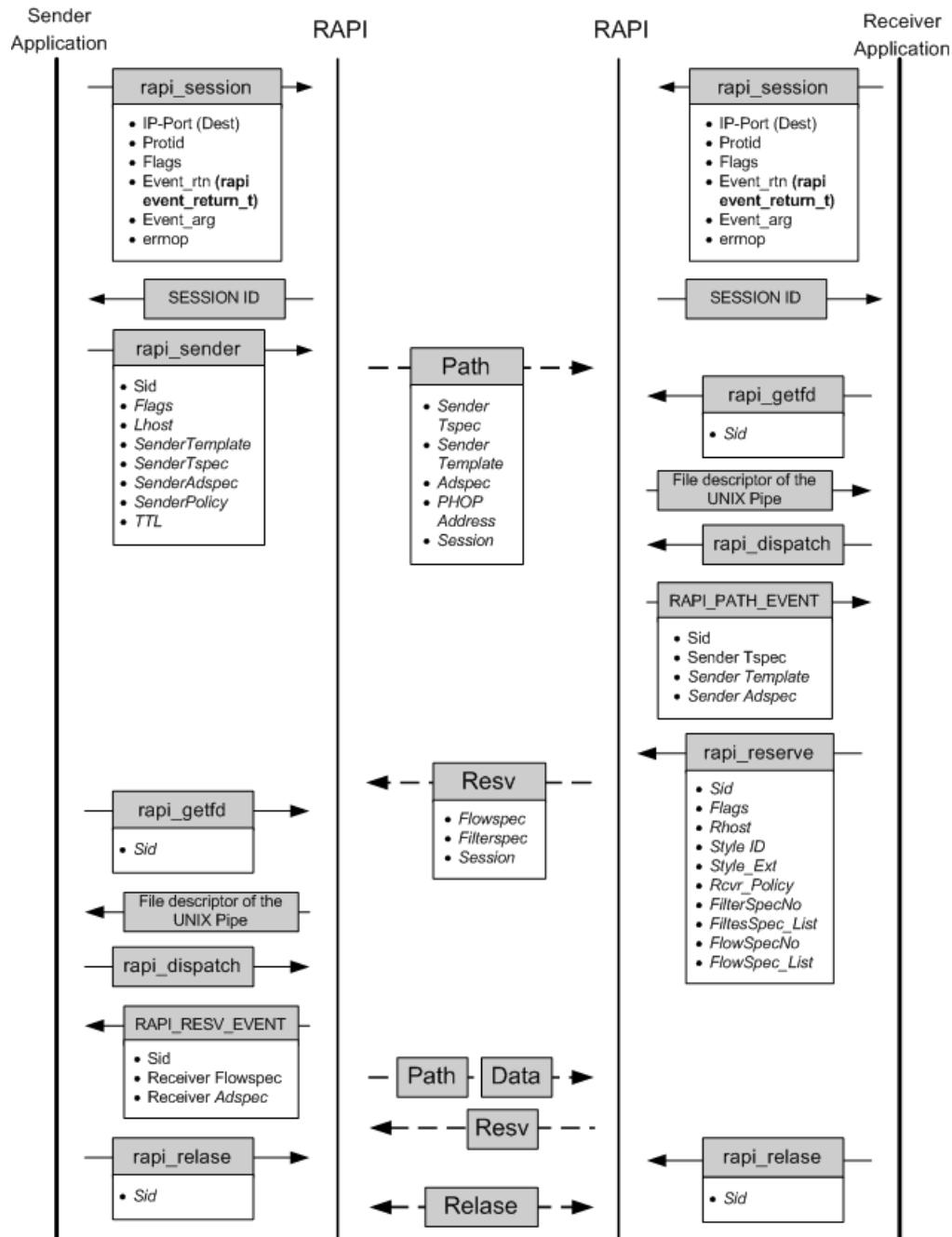


Figure 4-56: RAPI Interaction.

RAPI Reserve

An application calls this routine to make a QoS reservation as a data receiver. The RAPI reserve specifies the session handle that was returned by a successful call to the **rapi_session()** routine. Routine *Style ID* parameter specifies one of the following reservation style identifiers: Wildcard Filter (RAPI_RSTYLE_WILDCARD), Fixed Filter (RAPI_RSTYLE_FIXED) and Shared Explicit (RAPI_RSTYLE_SE). Also it specifies several RSVP objects like:

- **FilterSpec:** The filter spec, together with a session specification, defines the set of data packets (the flow) to receive the QoS defined by the flowspec. If the *FilterSpecNo* parameter is zero (0), this parameter is ignored.
- **FlowSpec :** The flowspec specifies a desired QoS. The flowspec is used to set parameters in the node's packet scheduler or other link layer mechanism, while the filter spec is used to set parameters in the packet classifier. Data packets that do not match any of the filter specs for the session are handled as best-effort traffic. If this parameter is zero (0), the **rapi_reserve()** routine removes the current reservation or reservations for the specified session and ignores the *FilterSpec_List* and *Flowspec_List* parameters.

The parameters sent in RSVP messages are:

PATH

- *SenderTspec:* contains the QoS parameters for sent traffic.
- *SenderTemplate:* Parameter by which the sender identifies itself and describes the format of the packages that the emitter generates. It also contains the IP address of the sender, source port and protocol.
- *Sender adspec:* it informs of the state of the network to enable the receiver application to initiate the calculation of the properties of QoS that will settle down in the way.
- *Session:* Destination IP address/port and protocol identifier to which the socket is sending.

RESV

- *FlowsSpec:* contains desired QoS parameters for traffic to be received.
- *FilterSpec:* contains the source or sources from which QoS-enabled traffic will be received.
- *Session:* contains the destination of the sent traffic.

4.5.2.2 QoS interoperable middleware architecture

In the previous section, two major QoS technologies, one applicable to the home environment and one to the wide area networks, have been presented together with their specific parameterizations of quality of service. In order to provide interoperability between QoS technologies, an abstraction of QoS is required as a framework. Initial related work has been elaborated as a semantic ontology in Deliverable D3.1a [Amigo-D3.1a], and will be refined in future work. Based on these semantics, QoS requirements can be understood by the interoperability modules and translated to the appropriate technology syntax and procedure using technology-specific interoperability methods that will make the semantics effective. Our general architecture for QoS interoperability is depicted below:

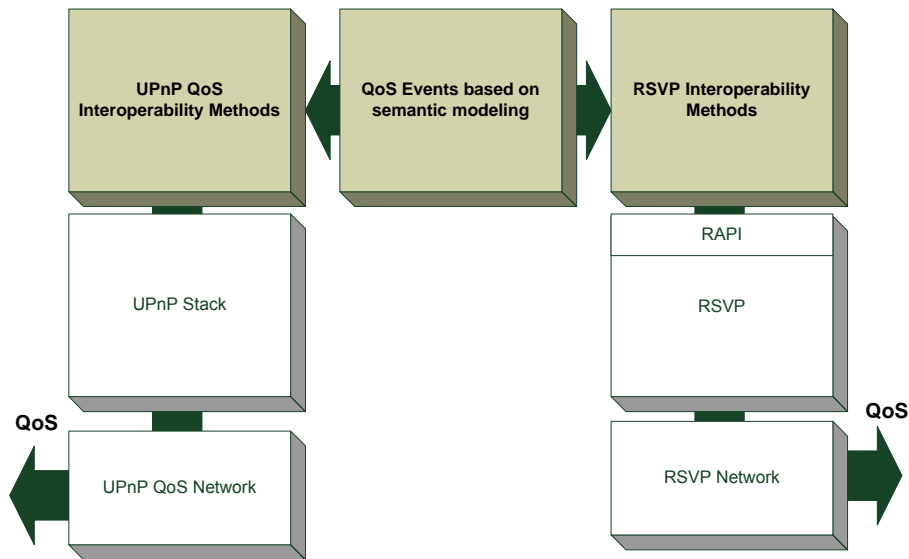


Figure 4-57: QoS interoperability: UPnP QoS and RSVP interoperability.

The Interoperability Methods presented in Figure 4-57 will provide intrinsic QoS technology difference adaptation (i.e. 'per flow' bandwidth reservation to packet priority mapping, specific calls, etc.) at middleware level and will relay on the interaction protocol specific stacks. The translation of the abstract QoS events to a real QoS technology will be performed by the interoperability methods whose architecture is proposed below and depicted in Figures 4-58 and 4-59.

There are two different cases for which interoperability must be provided (not necessarily at the same time) depending on the direction of the QoS requirement initiation. Therefore, for each QoS technology the middleware must provide inbound mechanisms to serve network initiated demands, and outbound mechanisms to serve middleware initiated demands (that is, initiated from another QoS technology network or a local application): in Figure 4-58 these are referred as *QoSP Inbound* and *QoSP Outbound units* respectively. A *QoSP unit* performs the protocol specific actions corresponding to a QoS event. *Outbound units* will be communicated via QoS events with *Inbound units* and therefore both will be event publishers and mutual event subscribers. Interoperability between QoS protocols is provided in one direction by an inbound unit and an outbound unit connected with each other. The unit pair subscription to events will be generated by the middleware based on the locally available QoS technologies and on incoming QoS demands. In Figure 4-58, the *UPnP outbound unit* makes use of the interoperability mechanism presented in Section 4.1 to use any communication protocol (such as RMI invocations) for forwarding demands to the local QoS technology methods (e.g. UPnP QoS Manager service).

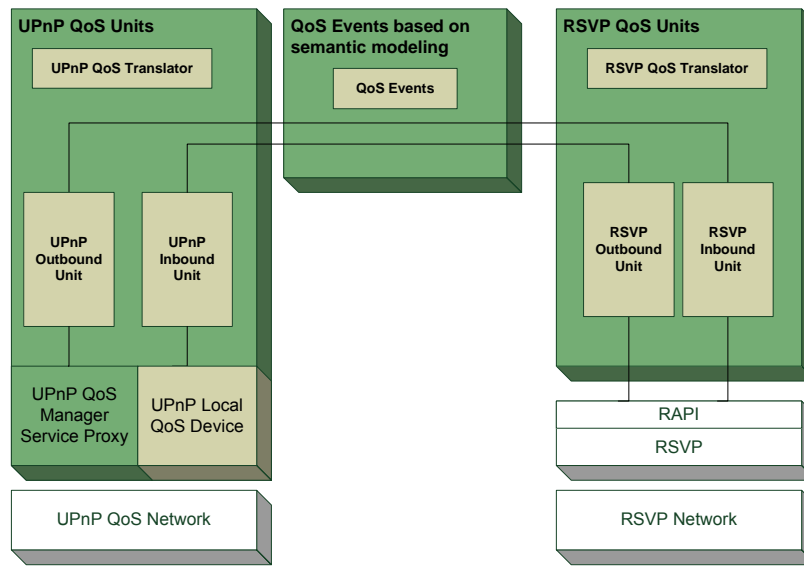


Figure 4-58: QoS interoperable middleware architecture refined for UPnP QoS and RSVP

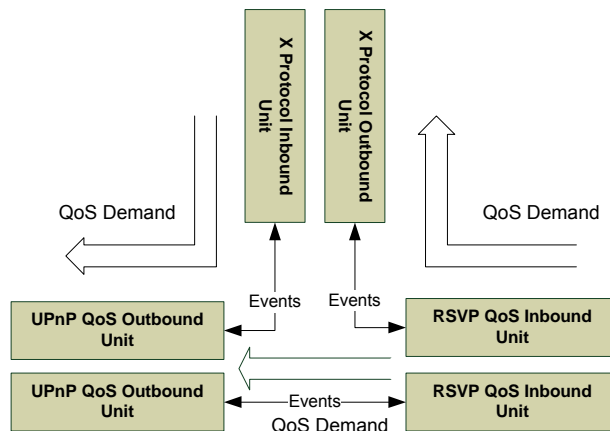


Figure 4-59: Multiple QoS interoperability based on unit pairs.

On the other hand, the *UPnP inbound unit* must be aware of QoS demands invoked on the QoS Device service requiring interoperability. In the case of invocation based protocols this awareness must be blocking (i.e. by using an upcall mechanism) since no response can be given to the client until an acknowledgement is received from the other technology network.

The *units* related to a given technology make use of a *Translator* which is responsible of mapping events in the common QoS semantics into the UPnP QoS specific vocabulary and vice versa (see Figure 4-60). The translator module is bidirectional and is common for all units based on the same technology.

Outbound and inbound units will differ mainly in the protocol specific methods they use e.g. as it was mentioned above a *UPnP QoS Outbound Unit* will interact with a possibly remote QoS Manager service, while the *UPnP QoS Inbound Unit* will interact with the local QoS Device implementation. An abstract QoS Outbound Unit will be composed of a state machine and an Outbound Connector. The connector will receive QoS events from the state machine which will be translated into the appropriate vocabulary using the *Translator* and will perform the appropriate actions and invocations onto drivers or services. This procedure is similar to the

one carried out in the opposite direction: a response from a service or driver may lead to an event which will be expressed, after being mapped by the translator, in the common event vocabulary. This event will be managed by the state machine and possibly passed on to the associated listener.

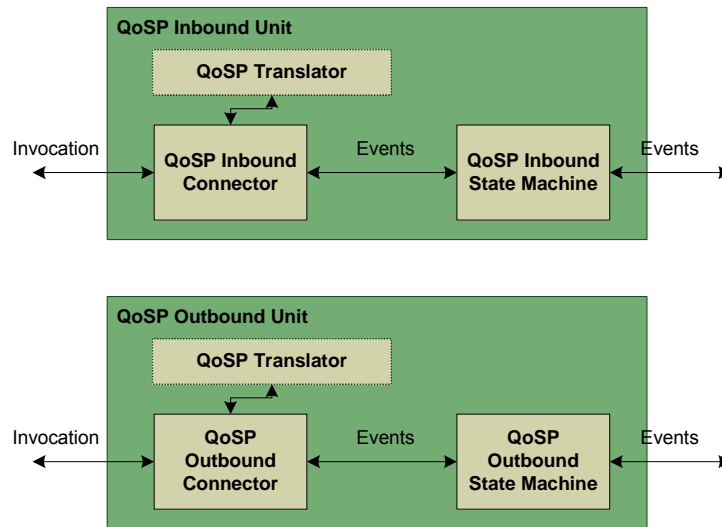


Figure 4-60: Abstract QoS Unit Architecture

Table 4-2 lists the identified QoS events. These events will be produced by QoSP inbound and outbound connectors triggered by the interaction with the protocol specific methods. The unit's state machine will damp protocol specific transitions between common QoS events and retain state conscience between these.

Event	Description
QoS_REQUEST	Request a quality of service
QoS_ACCEPT	Accept the request or the proposal
QoS_NEGOTIATE	Negotiate the initial request with a proposal
QoS_RELEASE	Terminate the QoS session

Table 4-2: Identified common quality of service events

The following example will help to illustrate the theory of operation of the proposed architecture. Suppose there is a gateway between a RSVP network and a UPnP QoS network. The middleware will connect the four QoSP Units by making outbound units subscribe to inbound units events and *vice versa*. When a RSVP "Path" message is received the RAPI receiver upcall function in the *Inbound Connector* will construct the QoS_REQUEST event with the parameters obtained from the *RSVP QoS Translator* and will pass it to the state machine. The *Translator* may simplify or not, depending on QoS semantics, RSVP specific data like the filterspec or the adspec parameters. If waiting for this event, the inbound state machine will change state and throw the event with its parameters, which will be captured by the UPnP outbound unit. The UPnP QoS Outbound unit will process the event if appropriate according to the current state, by mapping the event parameters to the UPnP own vocabulary through the *Translator* (i.e. completing the QoS semantic based event by possibly assigning a "trafficImportanceNumber" to the flow description) and passing it to the UPnP QoS Outbound

connector that will in turn access the UPnP QoS Manager Service (using Control Point methods or a service proxy). If the result of this invocation comes to be a success then a confirmation event will be back-propagated and transitions will take place in both state machines, resulting in a "Resv" message towards the RSVP network.

5 Integrated Prototype

The integrated prototype presented in this section is aimed at demonstrating the interoperability mechanisms provided by the Amigo interoperable middleware core and elaborated in Chapter 4 of this document. This integrated prototype provides a first, proof-of-concept integration of several interoperability mechanisms across the Amigo domains, i.e., the PC, mobile, domotic and CE domains.

Section 5.1 describes the integrated prototype infrastructure and the details of the configuration of the different devices and software components composing the prototype. Section 5.2 presents how the interoperability mechanisms elaborated in Chapter 4 are integrated into this prototype, each one taking care of a specific aspect of interoperability. Finally, Section 5.3 illustrates the visualization tool used to show the interactions and messages exchanged among the components of the prototype.

5.1 Scenario and integrated prototype infrastructure

Scenario

It's a Tuesday evening and John is returning home after a day at work. He opens the door, takes his Personal Remote Control (PRC) out of his pocket and, as there is not enough light in the living room, turns on the lamp using the PRC's domotic GUI. Then, sitting on the sofa, he switches the TV set on, and using the PRC's GUI for Consumer Electronics, he browses the multimedia content directory available on the DMS of the Amigo home environment via the wireless network. After having listened to some music, he finally decides to watch a movie. Before playing the movie, he turns on the coffee machine (using the PRC's domotic GUI) to warm up some coffee. And when coffee is ready, he finally turns off the lamp using the PRC's domotic GUI and sits on the sofa to watch the movie.

Integrated prototype infrastructure

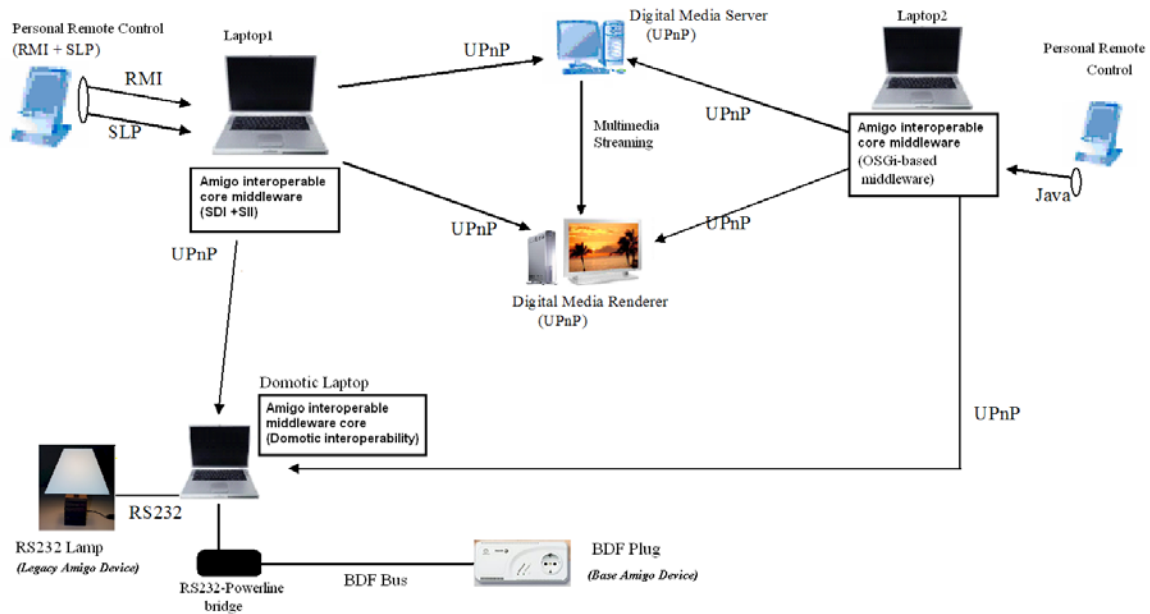


Figure 5-1: Integrated prototype infrastructure

Figure 5-1 shows the infrastructure of the integrated prototype that implements the scenario described above. The figure also shows the interactions among the different devices and components in terms of service discovery and interaction, with a reference to the protocol of each exchanged message and, for each component, the protocols on which it is based.

The Personal Remote Control is realized by two PDAs (Personal Digital Assistants) that run some client applications providing the functionalities to control the CE and domotic devices available in the Amigo home environment.

The Digital Media Server (DMS) provides the multimedia contents. The Digital Media Renderer (DMR), which visualizes multimedia contents, consists of a Philips Streamium connected to a TV set through a SCART cable.

The Laptop1 hosts the Amigo interoperable middleware core mechanisms described in Sections 4.1 and 4.2, while the Laptop2 incorporates the programming and deployment framework described in Section 4.3.

The Domotic Laptop controls the domotic devices installed in the home environment, that is, the lamp and the coffee machine, connected to the laptop through a plug.

The PDAs, the laptops, the DMS and DMR are all connected through a Wireless WiFi Network (infrastructure mode).

As the home is equipped with a machine running the Amigo interoperable middleware core, the services and clients running on the different devices can be discovered and can interact, even if based on different service discovery protocols (SDPs) and service interaction protocols (SIPs).

We detail below each device shown in the integrated prototype infrastructure figure with a description of the device and of the software provided.

Personal Remote Control

The essential role of the Personal Remote Control is to provide remote control software for controlling the Consumer Electronics subsystem (DMR and DMS) and the Domotic subsystem (plug and lamp).

The PDAs are Sharp Zaurus SL-6000 running Linux as operating system and supporting the Java platform Java2 Micro Edition Connected Device Configuration¹² (J2ME CDC) with Personal Profile (PP). The remote control client software deployed on the PDA is a Java application with a graphical user interface (GUI) based on Java `awt` library (J2ME CDC/PP supports only this graphic library, but does not support the Java graphic `swing` library).

The application is based on SLP for service discovery and RMI for service communication. The application sends SLP messages to discover instances of DMS and DMR in the environment. When the address of the service is returned by SLP, as the application is based on RMI technology, it looks up in the RMI Registry at the address notified by SLP to obtain the service proxy to access the remote service via RMI remote method calls.

The operations supported by the client GUI are: browsing the multimedia content directory provided by the DMS, selecting a multimedia content from those offered by the DMS and ask the DMR to play the content and further, to pause or stop the content that the DMR is currently playing.

Laptop1

¹² <http://java.sun.com/j2me/index.jsp>

The operating system running on the laptop is Linux and the software it provides is the Service Discovery Interoperability (SDI) and Service Interaction Interoperability (SII) of the Amigo interoperable middleware core described in Sections 4.1 and 4.2.

Laptop2

The operating system running on the laptop is Linux and the software it provides is the OSGi-based framework implementation described in Section 4.3.

Digital Media Server (DMS)

The Digital Media Server (DMS) can be either a laptop or a PC running Windows as operating system. The DMS software installed on the machine is Philips Media Manager¹³, a UPnP based software available only for Windows that provides access from a networked machine to multimedia content (music, movies and pictures) stored on the PC and explicitly made available through the DMS software. The official UPnP specifications for this device are part of the UPnP Device Control Protocol (DCP) standards¹⁴.

Digital Media Renderer (DMR)

The Digital Media Renderer (DMR) is a Philips Streamium SL300i¹⁵, a UPnP-based device that enables reproduction of multimedia content (music, movies and pictures) stored in an accessible DMS on a TV connected via a SCART cable. The official UPnP specifications for this device are part of the UPnP Device Control Protocol (DCP) standards¹⁶.

Domotic Laptop

The operating system running on the laptop is Windows XP. It provides the RS232 lamp proxy and the BDF Plug proxy that are respectively based on .NET framework and on Oscar OSGi Framework implementation¹⁷. All the components described in Section 4.4 realizing home automation interoperability run on this laptop.

The laptop accesses the power line using a RS232-Powerline bridge. This bridge is connected to a serial port of the laptop. As only one serial port is available in this laptop, a USB-Serial adapter is also required because both the bridge and lamp are connected to the laptop via serial ports.

Lamp

The lamp used is a prototype table lamp that has an RS232 serial connector. Its proxy running on the domotic laptop advertises the lamp as a UPnP device when connected. The lamp can be remotely switched on/off by any UPnP Control Point in the network.

Plug

¹³ <http://www.streamium.com/support/MediaManagerPC.cfm>

¹⁴ <http://www.upnp.org/standardizeddcps/mediaserver.asp>

¹⁵ <http://www.streamium.com/products/sl300i/>

¹⁶ <http://www.upnp.org/standardizeddcps/mediaserver.asp>

¹⁷ <http://oscar.objectweb.org/>

The domotic plug is a Fagor ED200-S DomoSwitch¹⁸ that supports the BDF protocol. It only requires to be connected to the home power line as any other electrical device. No further connections are required. Any other non domotic electrical device plugged in it can be remotely switched on/off or scheduled to be switched on/off at the desired time.

5.2 Integrated prototype realization using the Amigo interoperable middleware core

5.2.1 Integration of SDI and SII

In this section, we detail how the Amigo interoperable middleware core (SDI and SII) running on the laptop is integrated into the prototype to allow all the devices and software components to work together even if they are based on different technologies.

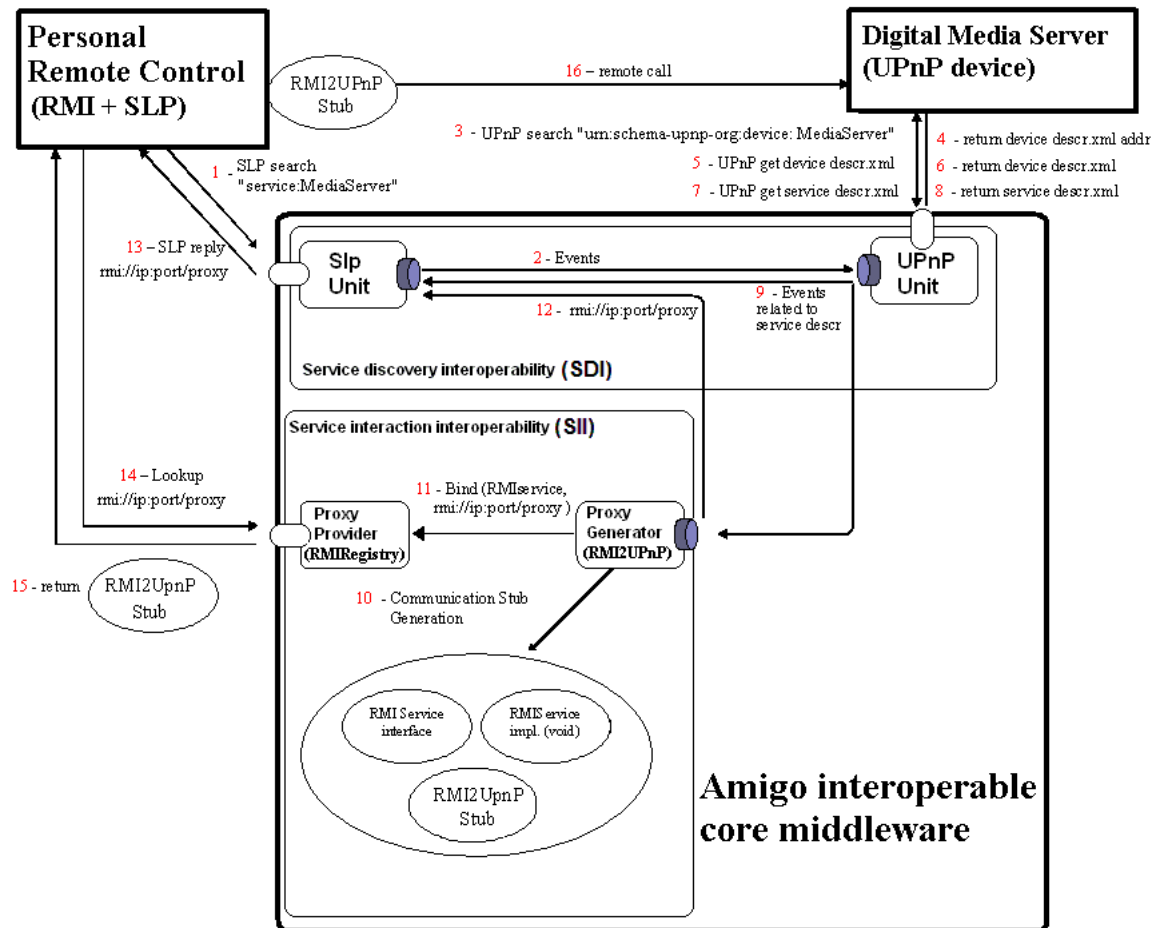


Figure 5-2: SDI and SII integration into the prototype

Figure 5-2 shows the sequence of messages and events exchanged among the different components involved in the integrated prototype to enable service discovery and interaction

¹⁸ http://www.fagor.com/es/domotic_n/index.html

through interoperability methods. In particular we focus on the interaction between the Personal Remote Control, the Digital Media Server (DMS) and the Amigo interoperable middleware core (SDI and SII). Below, we detail each step of the interaction represented in Figure 5-2:

1. The client application is based on SLP for service discovery. The name of the service to be discovered is fixed to the identifier corresponding to UPnP standards for DMS¹⁹: `service:MediaServer`. The SLP request message is received by the SLP unit of the Amigo interoperable middleware core.
2. The SLP request message is parsed by the SLP unit's parser and the semantic events are generated and dispatched to the UPnP unit and its composer using the mechanisms described in Section 4.1.
3. The UPnP unit's composer translates the semantic events into a semantically equivalent UPnP device discovery message for device `urn:schema-upnp-org:device:MediaServer`.
4. The UPnP device matching the discovery message replies with a message containing the URI of the document describing the device and all the services supported by the device. The message is received by the UPnP parser and the related events are generated.
5. The events describing the device description URI are used by the UPnP unit's composer to send a UPnP message to request the device description XML document.
6. The UPnP parser (*DeviceDescr* class described in Section 4.1) receives the UPnP device description and generates the events related to the device description and to the services supported by the device (for each service supported, the device description contains the URI of the XML document that describes the service).
7. For each service, the events related to the description's URI are used by the UPnP unit's composer to send a UPnP message to get the service XML description.
8. The UPnP parser (*ServiceDescr* class described in Section 4.1) receives the UPnP service description and generates the events related to the service description (events are related to the description of the interface provided by the service: it's a list of methods and their descriptions).
9. The events generated in steps 4, 6 and 8 are sent to the SII's Proxy Generator and to the SDI's SLP unit.
10. The Proxy Generator dynamically creates the remote service's proxy and all the other classes required to deploy the proxy on the Proxy.
As the client is based on RMI technology (for more details, see RMI specifications²⁰), the SII provides an RMI Registry that is used by the client to dynamically download the proxy. Registering the proxy on the RMI Registry requires the following classes: the java RMI

¹⁹ <http://www.upnp.org/standardizeddcps/mediaserver.asp>

²⁰ <http://java.sun.com/products/jdk/rmi/>

interfaces provided by the remote service, the proxy implementing all the methods declared by the interfaces and finally the RMI service that provides a void implementation of all the methods declared in the interfaces.

11. When all the classes have been generated in step 10, the Proxy Generator registers the proxy on the RMI Registry and binds the proxy to a specific address (*rmi://ip:port/proxy*).
12. The address bound to the proxy in step 11 (*rmi://ip:port/proxy*) is notified with an event to the SLP unit.
13. The SLP unit dispatches the address received in step 12 to the SLP composer that is in charge of creating (taking into account all the events received from the beginning of the operation) the SLP reply and sending it to the client.
14. The client receives the service address (*rmi://ip:port/proxy*). Since it expects the remote service to be an RMI service using SLP as discovery protocol, it looks up on the RMI Registry at the address received as SLP reply.
15. The RMI Registry on the Amigo interoperable middleware core returns the service proxy to the client application.
16. The client finally invokes a method on the remote service making use of the proxy. The message sent over the network and the reply from the service are SOAP messages and the proxy is in charge of the translation from SOAP to java for the client.

The process described above to enable service discovery and interaction interoperability, and detailed for the special case of DMR, is the same for any other UPnP device in the integrated prototype (e.g., the DMR, the UPnP proxy for plug and the UPnP proxy for lamp). The internal components of the Amigo interoperable middleware core used for these other devices will be the same. There is no specialization required for the SDI and SII components to be used for different devices and services, so they can be reused without any modifications.

5.2.2 Integration of the OSGi-based framework

In this section, we employ the OSGi-based framework introduced in Section 4.3 to develop and deploy a middleware architecture realizing interoperability between the graphical client and the UPnP devices of the integrated prototype. We show how interoperability can be ensured by a set of OSGi "bundles" interacting with each other, each of them dealing with a different aspect of interoperability. First, we present two possible deployment architectures for the prototype, where the bundles ensuring interoperability are either deployed on the client, or on a separate node. Then we detail the role and behaviour of each bundle, and the way bundles interact with each other.

Deployment architecture

The demonstration can be deployed according to two different architectures:

1. Interoperability methods co-localized with the client. In this case, the client is an OSGi bundle that uses the OSGi framework to retrieve instances implementing wished Java interface (MediaServer, MediaRenderer,...). Figure 5-3 summarizes the deployment architecture: the client can discover and interact with the external devices (Media

Renderer, Media Server) as soon as the amigo_interop and upnp base driver bundles are launched. The role of each bundle is explained in the next subsections.

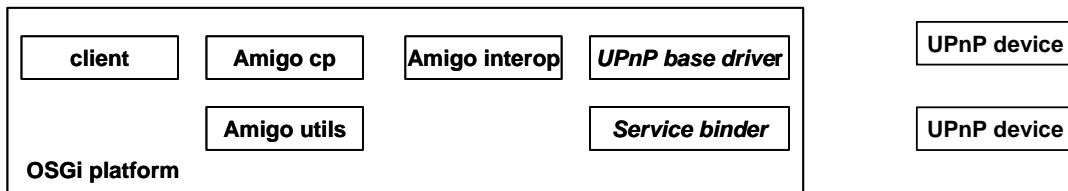


Figure 5-3: First OSGi-based deployment architecture

2. Interoperability methods on a node (called Interoperability node), and remote client on the PDA. We use in this case the same client as in the non-OSGi case presented in Section 5.2.1. Figure 5-4 summarizes the deployment architecture: the same bundles as in the previous architecture (amigo_interop and amigo_utils) are deployed on the OSGi platform, plus an additional bundle (Amigo slp) that allows the publication according to the SLP protocol. A rmi registry must also be running on the same node as the OSGi platform (possibly in a bundle or outside the platform).

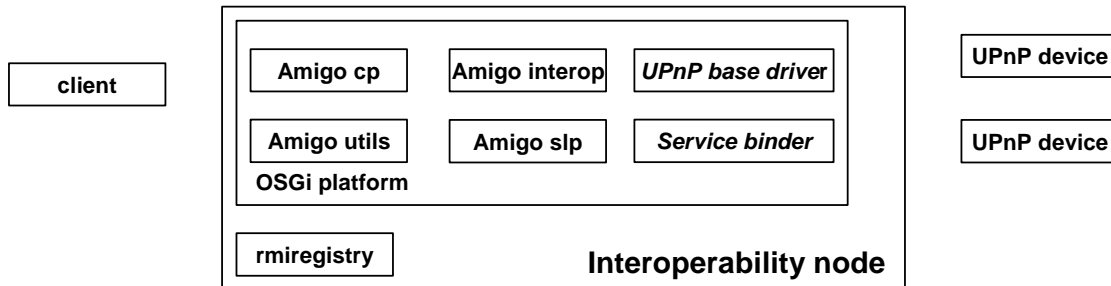


Figure 5-4: Second OSGi-based deployment architecture

SLP/RMI client

The SLP/RMI client is described in Section 5.2.1.

Local client

The local client is based on the same code as the SLP/ RMI client. The differences lie in the bootstrap process (the functional components are instantiated by a main program in the case of SLP/ RMI client, by the bundle's activator in the case of the OSGi client) and in the lookup service (the implementation of ServiceLookup and ServiceReference are based in one case on SLP/ RMI registry, and on the OSGi framework local lookup in the other case).

When the bundle is started, the client starts looking for local instances of MediaRenderer and MediaServer.

UPnP base driver bundle

This bundle is specified by the OSGi R3 specifications. The specification defines Java interfaces (UPnPDevice, UPnPService, UPnPAction) that corresponds to the UPnP abstractions. Also, it specifies the behaviour that a bundle must provide in order to be a conformant "UPnP base driver".

Among others, a conformant UPnP base driver provides implementations of the UPnPxxx interfaces that allow interacting with a UPnP device. These UPnPDevice implementations provide all necessary methods to interact with a real UPnP device: obtain the list of services provided by the device, as a set of UPnPService instances, the list of actions provided by each service, invoke an action etc.). The UPnP base driver manages the UPnP stack, builds a local instance of the "UPnPDevice" class as soon as a device is announced on the network and announces these instances using the OSGi local lookup.

Several implementations of the UPnP base driver exist (open source or proprietary). We use the open source implementation provided by domoware²¹.

Service binder bundle

The "service binder"²² bundle eases the interaction between bundles by introducing the concept of *service component* to the OSGi framework. A service component is similar to the concept of a logical bundle but the difference is that multiple service components can be deployed inside a single physical bundle.

A service component declares a set of provided service interfaces, and a set of required service interfaces. During execution, an instance of a service component implements the provided services and is connected to other instances that implement the required interfaces.

The service binder is an open source development.

Amigo_util bundle

This bundle provides generic classes (packages rmiholders and standard) described in chapter 4.2.

SLP bundle

This bundle publishes a service which offers a programmatic interface to the SLP protocol Service Agent functions.

Amigo_interop bundle

This bundle provides Java interfaces and classes generated from the XML description of UPnP devices. It provides also the interoperability methods between UPnP and OSGi local clients, and between UPnP and RMI/SLP client. The latter is achieved only if the SLP bundle is deployed on the OSGi platform.

An extension of the proxy generator presented in Section 5.2.1 has been developed. This OSGi code generator parses a set of UPnP xml descriptions and generates classes that implement the interfaces expected by the client. Generated classes extend `java.rmi.UnicastRemoteObject` and rely on `UPnPDevice`, `UPnPService` and `UPnPAction` (as defined by the OSGi standard). More precisely, an instance of a generated class (say, `MediaRendererOSGi_Impl`) is associated to an instance of `UPnPDevice`, and a set of `UPnPService` and `UPnPAction` instances. Invoking a specific method on a `MediaRendererOSGi_Impl` results in building a set of arguments for the corresponding `UPnPAction`, calling the "invoke" method (which transparently results into a UPnP network invocation), retrieving the result and filling the corresponding rmi holders.

²¹ <http://domoware.isti.cnr.it/>

²² <http://gravity.sourceforge.net/servicebinder/servicebinder-index.html>

The role of the Amigo_interop bundle is to create instances of these generated classes as soon as UPnPDevice instances appear, and publish these instances so that the client can discover and use them.

The interoperability bundle code is not specific to the devices/services supported by this prototype and could be reused in another prototype with different ones.

Interaction between bundles

Bundles interact by means of the service binder described above. As an example, the interoperability bundle declares that it contains a component (called AmigoInteropComponent) and that this component must be warned each time an instance of the "UPnPDevice" class is published on the OSGi lookup. The service binder instantiates AmigoInteropComponent, and tracks registrations of instances of UPnPDevice.

The diagram of Figure 5-5 illustrates the discovery of a UPnP device. The UPnP base driver handles the UPnP stack. As soon as a device is announced on SSDP, it creates an UPnPDevice instance and publishes it by the OSGi lookup. The service binder is informed (2.1) and calls the corresponding method (2.1.1) on Amigo Interop Component. The interop component asks the device its description (2.1.1.1), extracts the UPnP device type from the description and searches which Java type correspond to the UPnP type of the UPnPDevice (2.1.1.2), in this case it is MediaRendererOSGi_Impl. It instantiates this class (2.1.1.3) and links the instance with the UPnPDevice (2.1.1.4). Finally, it registers it as a MediaRenderer, so that the local client can discover and use it.

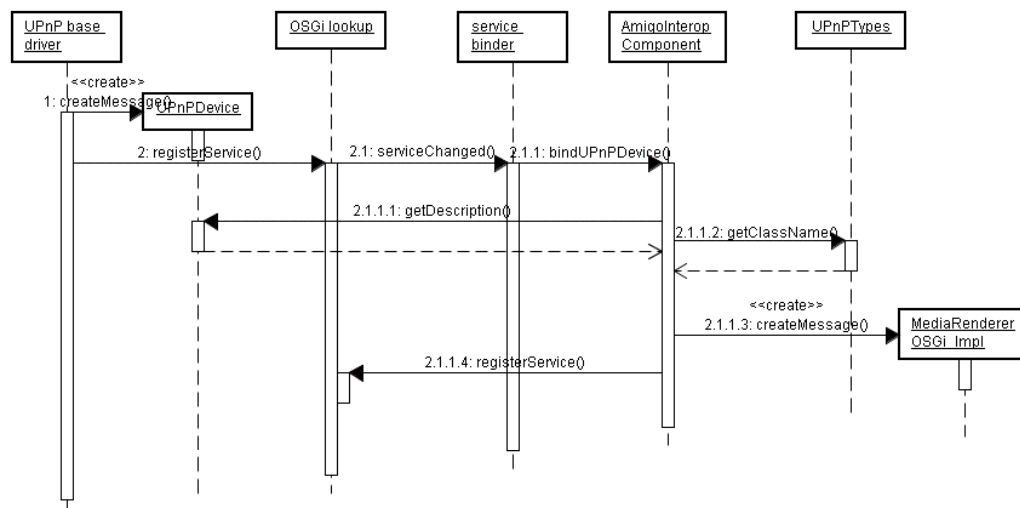


Figure 5-5: Sequence for discovery of a UPnP device

If an SLP service is present (not shown on the diagram for the sake of simplicity), the additional actions are performed:

- Export the MediaRendererOSGi_Impl using rmi's UnicastRemoteObject exportObject method
- Generate a uuid
- Bind the object to the local rmi registry using the generated uuid
- Register the rmi URL using SLP.

The Amigo interoperability bundle and the UPnP base driver play complementary roles in the interoperability middleware: the Amigo interoperability bundle does not deal directly with the UPnP protocol, but rather uses the abstractions provided by the OSGi R3 specification. It provides model interoperability (match the generic model of UPnP artefacts onto a specific Java interface) inside the OSGi platform, whereas the UPnP base driver provides interoperability between the Java world and UPnP devices.

5.2.3 Integration of domotic interoperability

In this section, we detail how the Amigo domotic interoperable architecture makes domotic devices (which use service discovery/interaction protocols not supported by the Amigo system) available to both middleware realizations: the interoperable middleware core (SDI and SII) and the OSGi-based middleware.

UPnP proxies of the domotic devices are provided by the domotic interoperable architecture, by means of the components described in Section 4.4 running on the Domotic laptop described in Section 5.1.

The BDF Plug is based on BDF protocol for discovery and communication. Thus, the objective is to instantiate a UPnP proxy, useful for the middleware prototype, representing the physical BDF device. BDF protocol's physical layer is power line. The BDF plug is, then, only connected to the power line and requires no further connection. All the BDF messages are sent to and received from the power line. First of all, a BDF bus listener is required, so that BDF messages through the power line can be managed. A RS232-Powerline bridge enables the passing of messages from the laptop to the power line and back. The bus listener is a software component running on the laptop so, with the described setup, this component can listen to the BDF messages in the power line. It's implemented as an OSGi bundle, and runs over Oscar²³.

We recall here, the behaviour of the domotic interoperability explained in detail in Section 4.4.2.2. When a BDF plug is discovered by the bus listener in the power line, another bundle (a *BDFServiceFactory* bundle) is notified about the new physical device, and instantiates a *BDFService* (a java object to access the real device). This *BDFService* is registered in the OSGi framework, advertising it and making it available to every component in the framework. A *BDFModelFactory* (listening to *BDFService* registrations in the framework) instantiates a generic *ServiceModel* java object that describes the methods and properties of the *BDFService*, but decoupling it from the BDF underlying technology. This *ServiceModel* is a generic service that is also registered and advertised within the OSGi framework. A UPnPDevice factory component receives the notification of the availability of the *ServiceModel* in the framework, and instantiates the corresponding UPnP device. Thus, we provide a UPnPDevice proxy to handle the physical BDF plug. UPnP service discovery/interaction messages from the interoperable middleware core and from the OSGi based middleware are managed by the proxy are finally sent to the BDF plug.

The RS232 Lamp has a RS232 connector to access the functionalities of the lamp: getting and setting its status. Thus, the objective is to instantiate a UPnP proxy, representing the physical device, and providing for the middleware prototype with access to the lamp. In this case, .NET technology has been used for the required implementation. A RS232 Monitor running on the laptop is continually checking the presence of the lamp in the serial port. When the lamp is detected (responding to a predefined protocol), a lamp model is built and a UPnP proxy of the lamp is instantiated. By means of this proxy, the interoperable middleware core and the OSGi based middleware can interchange messages with the lamp using the RS232 interface.

²³ <http://oscar.objectweb.org/>

5.3 Integrated prototype visualization

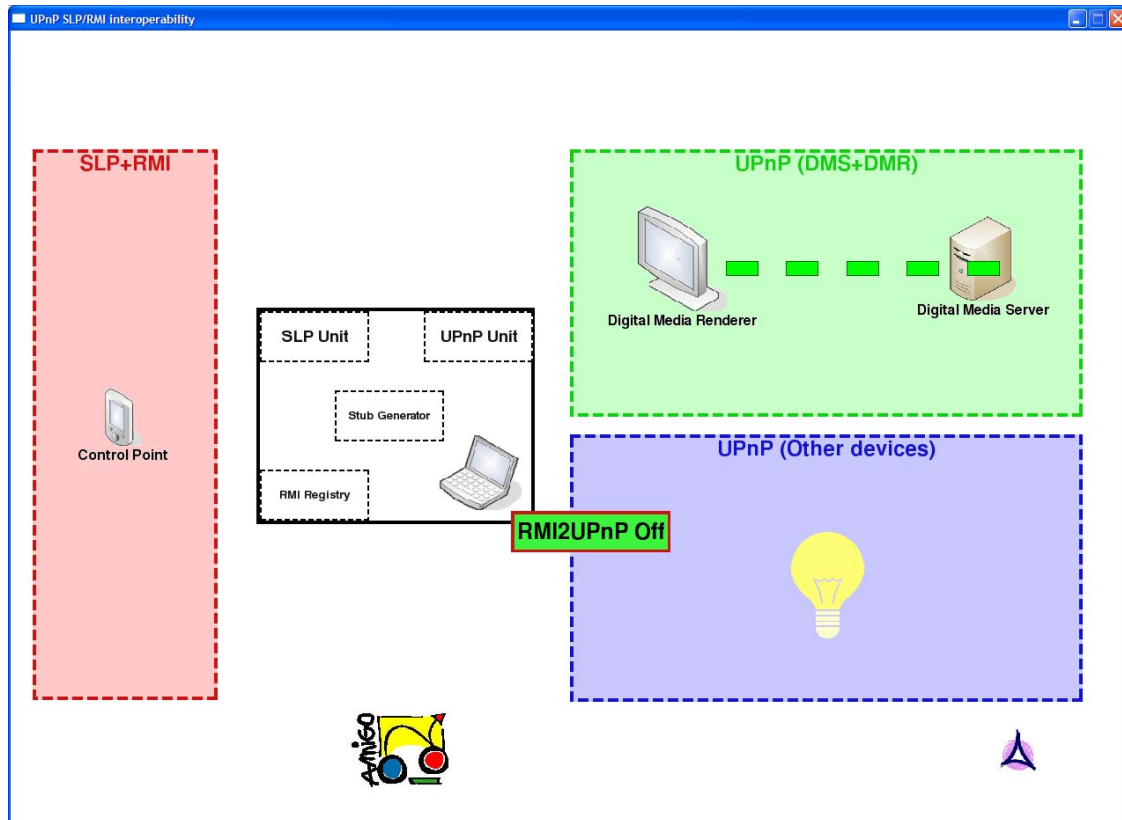


Figure 5-6: Preliminary version of the Amigo integrated prototype visualisation.

The viewer application is written in the Python language²⁴. It utilizes a gaming library called PyGame²⁵ for displaying and moving images. PyGame is a set of Python modules designed for writing games. It is written on top of the Simple Directmedia Library (SDL)²⁶, allowing the creation of fully featured games and multimedia programs in the python language. PyGame is highly portable and runs on nearly every platform and operating system.

The viewer is constructed by a number of entities such as Nodes, Messages, Lines, Actions, and Layers. These can be combined to form a visualisation such as the one shown in Figure 5-6. Nodes, Messages, and Lines can be animated (e.g. resized, moved, rotated, etc).

The triggers for these animations can come from different types of input. Typically this input will be the keyboard keys, for example to start a pre-programmed scenario, or from the 'sniffer' input that allows triggers to be based on specific packets sensed on the network.

Essentially the sniffer input is a python object wrapper around a Libpcap²⁷ (on Linux) or winpcap²⁸ (on Windows) library. The standard libpcap syntax is used for defining filters for these sniffer inputs.

²⁴ <http://www.python.org>

²⁵ <http://www.pygame.org>

²⁶ <http://www.libsdl.org/index.php>

²⁷ <http://www.tcpdump.org/>

²⁸ <http://www.winpcap.org/>

A server socket can also be used as an input allowing arbitrary clients to send events of interest to the viewer application. The form of these events is plain ASCII text, allowing maximum flexibility in defining events and actions.

All animations are placed in a buffer from which they are retrieved one after the other for playing. The rate at which they are retrieved and played can be varied, making the viewer application well-suited for illustrating protocols and their message exchanges between different computers.

A client socket in Java is provided as well to allow the Amigo middleware to send events of interest to the viewer application.

6 Conclusion

In this report, we have elaborated on the Amigo Interoperable Middleware Core and on Amigo-aware Service Specification, refining the Amigo abstract middleware architecture presented in Deliverable D2.1. We have considerably advanced our work on the middleware core by providing detailed design and prototype implementation of essential functionalities. Moreover, prototype implementations related to different application domains have been linked into an early, proof-of-concept, integrated prototype. In the next phase of the project, we will proceed towards a full prototype implementation of the Amigo middleware core.

Further, we have provided an informal definition of a declarative language for semantic service specification and associated conformance relation mechanisms. We intend to proceed to a detailed, formal definition of the language and elaborate associated online tools for service matching and interoperability.

Finally, we have identified the different classes of services and interoperability levels in the Amigo home environment. Co-existence of legacy services, middleware interoperable services and Amigo-aware services is inevitable in the open Amigo environment. We will further elaborate all required mechanisms that will make it possible for services of different classes to integrate and interoperate.

Acronyms

ADSL	Asymmetric Digital Subscriber Line
API	Application Programming Interface
BDF	Bus Domotico Fagor (Fagor Domotic Bus)
CDC	Connected Device Configuration
CE	Consumer Electronics
CEA	Consumer Electronics Association
CORBA	Common Object Request Broker Architecture
DHCP	Dynamic Host Configuration Protocol
DIDL	Digital Item Declaration Language
DLNA	Digital Living Network Alliance
DMR	Digital Media Renderer
DMS	Digital Media Server
EHS	European Home System
EIB	European Installation Bus
FTTH	Fiber To The Home
GUI	Graphical User Interface
HAVi	Home Audio Video interoperability
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
IETF	Internet Engineering Task
IOPE	Outputs, Pre-conditions and Effects
IP	Internet Protocol
ISMA	Internet Streaming Media Alliance
J2ME	Java 2 Micro Edition
JDK	Java Developers Kit
JPEG	Joint Photographic Experts Group
JXTA	Juxtapose
LPCM	Linear Pulse Code Modulation
MPEG	Moving Pictures Experts Group
OSGi	Open Service Gateway Initiative
OWL	Ontology Web Language
OWL-DL	Ontology Web Language Description Logics
OWL-S	Ontology Web Language for Services
PC	Personal Computer
PP	Personal Profile

PRC	Personal Remote Control
QoS	Quality of Service
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RSVP	Resource Reservation Protocol
RTCP	Real-Time Control Protocol
RTP	Real-Time Transport Protocol
RTSP	Real-Time Streaming Protocol
SCART	Syndicat des Constructeurs d'Appareils Radiorécepteurs et Téléviseurs
SD	Service Discovery
SDI	Service Discovery Interoperability
SDL	Simple Directmedia Library
SDP	Service Discovery Protocol
SDS	Service Discovery Service
SDSL	<i>symmetric digital subscriber line</i>
SII	Service Interaction Interoperability
SLP	Service Location Protocol
SMC	State Machine Compiler
SOAP	Simple Object Access Protocol
SOFA	Simple Ontology Framework API
SSDP	Simple Service Discovery Protocol
TCP	Transmission Control Protocol
TV	Television
UDDI	Universal Description, Discovery and Integration
UDP	User Datagram Protocol
UML	Unified Modeling Language
UPnP	Universal Plug & Play
UPnP AV	Universal Plug & Play Audio/Video
VDSL	Very high data rate Digital Subscriber Line
WSDL	Web Service Definition Language
WLL	Wireless Local Loop
W3C	World Wide Web Consortium
XML	eXtensible Markup Language

References

- [Amigo-D2.1] Amigo Consortium. Deliverable D2.1: Specification of the Amigo Abstract Middleware Architecture. February 2005.
- [Amigo-D3.1a] Amigo Consortium. Deliverable D3.1a: Detailed Design of the Amigo Middleware Core – Service Modelling for Composability. September 2005.
- [Amigo-D3.1c] Amigo Consortium. Deliverable D3.1a: Detailed Design of the Amigo Middleware Core – Security & Privacy, Content Distribution, Data Storage. September 2005.
- [BCMS03] P. Bellavista, A. Corradi, R. Montanari, C. Stefanelli, Context-Aware Middleware for Resource Management in the Wireless Internet, IEEE transactions on Software, 2003
- [BiWe] [Developers Guide to Semantic Web Toolkits for Different Programming Languages. Chriz Bizer, Freie Universität Berlin, Germany. Daniel Westphal, Freie Universität Berlin, Germany. <http://www.wiwiss.fu-berlin.de/suhl/bizer/toolkits/>.
- [BH98] R. Braden and D. Hoffman, "RAPI-An RSVP Application Programming Interface version 5", Internet Draft, August 11, 1998.
- [BPSK04] T. Broens, S. Pokraev, M. van Sinderen, J. Koolwaaij, and P. Dockhorn Costa, "Context-aware, ontology-based service discovery", in 2nd European Symposium on Ambient Intelligence (EUSAI'04). Eindhoven, the Netherlands: Springer Lecture Notes 3295, 2004.
- [CEA] Consumer Electronics Association. <http://www.ce.org/>
- [DeMi] Ontology Editor Survey 2004. Michael Denny. http://www.xml.com/2004/07/14/examples/Ontology_Editor_Survey_2004_Table_-_Michael_Denny.pdf
- [DLNA] Digital Living Network Alliance. <http://www.dlna.org>
- [GKRST] Google Directory – Reference > Knowledge Management > Knowledge Representation > Ontologies > Software and Tools.http://directory.google.com/Top/Reference/Knowledge_Management/Knowledge_Representation/Ontologies/Software_and_Tools/
- [HAVi] Home Audio / Video Interoperability. <http://www.havi.org/>
- [ISMA] Internet Streaming Media Alliance. <http://www.isma.tv/>
- [JRGL05] Michael C. Jaeger, Gregor Rojec-Goldmann, Christoph Liebetrueth, Gero Mühl and Kurt Geihs: Ranked Matching for Service Descriptions Using OWL-S. KIVS 2005: 91-102. <http://user.cs.tu-berlin.de/~michi/resources/kivs05-jaegeretal-owlsmatchmaker.pdf>
- [KIF] Knowledge Interchange Format: Draft proposed American National Standard (dpans). Technical Report 2/98-004, ANS, 1998. Also at <http://logic.stanford.edu/kif/dpans.html>.
- [PASS03] Massimo Paolucci, Anupriya Ankolekar, Naveen Srinivasan and Katia Sycara. The DAML-S Virtual Machine. In *Proceedings of the Second International Semantic Web Conference (ISWC)*, 2003, Sandial Island, Fl, USA, October 2003, pp. 290-305.
- [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.
- [PKPS02] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Semantic Matching of Web Services Capabilities. In *Proceedings of the 1st International Semantic Web Conference (ISWC 02)*, 2002.
- [PoKW03] S. Pokraev, J. Koolwaaij, M. Wibbels, Extending UDDI with Context-aware Features based on Semantic Service Descriptions, in: *Proceedings of the 1st International Conference on Web Services (ICWS'03)*, Las Vegas (USA), June 2003
- [Rals05] P.-G. Raverdy, V. Issarny, Context-aware Service Discovery in Heterogeneous networks, in: *World of Wireless Mobile and Multimedia Networks 2005 (WoWMoM 2005)*, Toarmina (Greece), June 2005
- [Saun04] Steven Saunders. *Home Network Quality of Service*. http://www.itu.int/ITU-T/worksem/hnhs/conclusions/S6_SS_conclusion.ppt
- [SWRL] SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Version 0.5 of 19 November 2003. Ian Horrocks, Department of Computer Science, University of Manchester. Peter F. Patel-Schneider, Bell Labs Research, Lucent Technologies. Harold Boley, National Research Council of Canada. Said Tabet, Macgregor, Inc. Benjamin Grosz, Sloan School of Management, MIT. Mike Dean, BBN Technologies. <http://www.daml.org/2003/11/swrl/>
- [UPnPAV] *UPnP AV Architecture 0.83*. <http://upnp.org/standardizeddcps/documents/UPnPvArchitecture0.83.pdf>
- [UPnPCD] *ContentDirectory:1 Service Template Version 1.01*. <http://www.upnp.org/standardizeddcps/documents/ContentDirectory1.0.pdf>
- [UPnPF] *UPnP Forum*. <http://www.upnp.org>
- [UPnPQoS] UPnP QoS Standards. <http://upnp.org/standardizeddcps/qualityofservice.asp>