Middleware Architecture for Ambient Intelligence in the Networked Home

Nikolaos Georgantas, Valerie Issarny, Sonia Ben Mokhtar, Yerom-David Bromberg, Sebastien Bianco, Graham Thomson, Pierre-Guillaume Raverdy, Aitor Urbieta and Roberto Speicys Cardoso

1 Introduction

With computing and communication capabilities now embedded in most physical objects of the surrounding environment and most users carrying wireless computing devices, the Ambient Intelligence (AmI) / pervasive computing vision [28] pioneered by Mark Weiser [32] is becoming a reality. Devices carried by nomadic users can seamlessly network with a variety of devices, both stationary and mobile, both nearby and remote, providing a wide range of functional capabilities, from base sensing and actuating to rich applications (e.g., smart spaces). This then allows the dynamic deployment of *pervasive applications*, which dynamically compose functional capabilities accessible in the *pervasive network* at the given time and place of an application request.

In the smart home environment, this translates into integrating today's, still mostly distinct, four application domains of the networked home:

- Personal computing, based on the home computers, printers and Internet connection;
- Mobile computing, manifested by the increasing use of home Wi-Fi and Bluetooth networks connecting laptops, PDAs and tiny personal devices;
- Consumer electronics, targeting multimedia in the home; and
- Home automation, adding intelligence to household appliances like washing machines and lighting systems.

Nikolaos Georgantas, Valerie Issarny, Yerom-David Bromberg, Sebastien Bianco, Graham Thomson, Pierre-Guillaume Raverdy and Roberto Speicys Cardoso
INRIA Paris-Rocquencourt, France, e-mail: `firstname.lastname@inria.fr`

Sonia Ben Mokhtar
University College London, UK, e-mail: `s.benmokhtar@cs.ucl.ac.uk`

Aitor Urbieta
Mondragon Unibertsitatea, Spain, e-mail: `aurbieta@eps.mondragon.edu`

Still, easing the development, from design to deployment, of pervasive applications raises numerous challenges. In this chapter, we concentrate on the software engineering aspect of pervasive computing, and more specifically on the adequate abstraction of networked resources, both software and hardware, and supporting software systems, so as to effectively enable pervasive applications to dynamically deploy over the pervasive network. Specifically, pervasive applications should be dynamically composed out of capabilities that are in reach to realize the target functional behavior while also meeting required quality of service (also referred to as non-functional properties). Our aim is to offer a solution that enables exploiting most networked resources, without being restrictive in terms of underlying software and hardware platforms, neither in terms of assumed resources. Indeed, while the development of advanced middleware platforms for pervasive computing has led to significant progress over the last decade [10], those platforms require consumers and providers of resources to agree on a common *syntactic* description of capabilities and distributed runtime environment for them to actually network. This assumption is too restrictive for truly open pervasive environments, despite existing ubiquitous software technologies, and in particular those from the Web. Common syntactic description of capabilities is not achievable on a large scale basis, neither is the usage of a common middleware platform.

Concretely, this chapter introduces the Amigo middleware approach for the open networked home, which was designed as part of the IST FP6 Amigo[1] project on the development of a networked home system towards the ambient intelligence vision. Furthermore in Amigo, we targeted the *extended* home environment, where users get access to applications also between homes, between home and workplace, and potentially anytime, anywhere. This chapter more specifically focuses on the *AmIi*[2] middleware interoperability solution developed at INRIA as part of the Amigo system architecture. As opposed to most existing pervasive system architectures, the Amigo approach does not impose any specific middleware technology, as it allows heterogeneous technologies to be integrated, establishing interoperability at a higher, *semantic*, level.

In Amigo, we exploit *service oriented architectures* [22], which appears as the appropriate architectural paradigm to cope with the above requirements [14]. Therein, networked devices and hosted applications are abstracted as services, which may dynamically be retrieved and composed, based on service discovery protocols as well as choreography and orchestration protocols [23]. We further advocate usage of semantic services. Semantics of an entity encapsulate the *meaning* of this entity by reference to an established vocabulary of terms (*ontology*) representing a specific area of knowledge. In this way, meanings of entities become machine-interpretable, enabling machine reasoning on them. Such concepts come from the Knowledge Representation field and have been applied and further evolved in the

[1] http://www.hitech-projects.com/euprojects/amigo/

[2] Ambient Intelligence interoperability

Semantic Web[3] domain [5]. The Web Ontology Language (OWL)[4] is a recommendation by W3C supporting formal description of ontologies and reasoning on them. A natural evolution to this has been the combination of the Semantic Web and Web Services[5], the currently dominant service oriented technology, into Semantic Web Services [19]. This effort aims at the semantic specification of Web Services towards automating Web services discovery, invocation, composition and execution monitoring. The Semantic Web and Semantic Web Services paradigms address web service interoperability [30, 21]. Nevertheless, our goal in Amigo is wider, that is, to address service interoperability without being tied to any service technology. To this end, we establish semantic service modelling independently of underlying service technologies. Based on such modelling, we empower the networked environment with interoperability mechanisms so as to allow networked services to seamlessly compose, independently of their underlying software technologies.

The remainder of this chapter is structured as follows. Sect. 2 presents the Amigo abstract reference service architecture that targets service interoperability, and further discusses limitations of related work with respect to achieving interoperability in service oriented systems. Sect. 3 then introduces the AmIi solution to interoperable service discovery; it is based on a generic service model on which the various existing service description languages may be mapped and thus compared against each other for the purpose of service matching; it further introduces a repository-based service discovery solution. Sect. 4 then concentrates on the AmIi solution to achieving interoperability among heterogeneous middleware communication protocols; once discovered as in Sect. 3, heterogeneous services can communicate as in Sect. 4. Finally, Sect. 5 concludes with a summary of our contribution and sketches our future work on enablers for pervasive computing.

2 Achieving Interoperability in AmI Environments

The Amigo service oriented system architecture aims to enable integrating diverse technologies in terms of networks, devices and software platforms. Thus, in the design of the Amigo system architecture, only a limited number of technology-specific restrictions are imposed. This further means that existing service platforms (e.g., Web Services[6], UPnP[7], etc.) relevant to the four home application domains are retained. Our intension was not to develop yet another service platform imposing a homogeneous middleware layer on all devices within the Amigo home, but to introduce an *abstract reference service architecture* for the Amigo system which can represent various service platforms by abstracting their fundamental features.

[3] http://www.w3.org/2001/sw/

[4] http://www.w3.org/TR/owl-semantics/

[5] http://www.w3.org/TR/ws-arch/

[6] http://www.w3.org/2002/ws/

[7] http://www.upnp.org

In the following section, we introduce the Amigo architecture, while in Sect. 2.2 and Sect. 2.3 we discuss related work on service platform interoperability.

2.1 The Amigo Service Architecture for Interoperability

The Amigo reference architecture [11] is based on a *typical* service oriented architecture, which follows a general application-middleware-platform layering structure [22]. In this typical architecture, the platform layer offers base system and network support, while the middleware layer includes essential mechanisms for service discovery and service communication. Furthermore, services in the application layer are functionally described based on a common syntactic service description language in order to enable service discovery and invocation independently of service implementation details. The elements of the typical service architecture are depicted on the left-hand side of Fig. 1 in normal typeface.

In Amigo, we enhanced the typical architecture to account for the open AmI environment. Hence, it is assumed that all three layers of the architecture may be heterogeneous and based on diverse technologies. Furthermore, support for context-awareness and quality of service (QoS)-awareness is incorporated, as these are core features in AmI environments. This leads to the enhancement of service description with context and QoS features, and to the establishment of capability- and resource-awareness in the base system and network. The above advanced elements added to the typical service architecture are depicted on the left-hand side of Fig. 1 in boldface.

Furthermore, to deal with the heterogeneity present in this *enhanced* service architecture for AmI, we introduce a common architectural and behavioural description at a higher, technology-independent level, based on semantic concepts. This description aims at enabling interoperability between heterogeneous service platforms in AmI environments, effectively offering *Ambient Intelligence interoperability (AmIi)*; we call this description *AmIi Service Description Model (ASDM)*. ASDM covers the elements identified in the enhanced service architecture, not only in the application layer but also in the underlying middleware and platform layers. This abstraction is illustrated in Fig. 1, where the right-hand layered structure complements the left-hand enhanced service architecture introduced so far, to produce the *Amigo abstract reference service architecture*.

Based on ASDM, we enable deployment of *AmIi interoperability mechanisms* within the reference architecture, which may concern any of the three layers (see Fig. 1). These mechanisms aim at establishing semantic-based service interoperability, and comprise *conformance relations* and *interoperability methods*. Conformance relations aim at checking conformance (matching) between services for assessing their capacity to interoperate. Interoperability methods aim at enabling integration of partially conforming services, thus allowing their seamless discovery, communication and composition.

Building on the above principles, we have developed in Amigo interoperability solutions to all of service discovery, communication and composition. In this chapter, we present the first two solutions; for the third, the interested reader is referred to [2]. In the next two sections, we survey related research work on service discovery and communication interoperability.

2.2 Service Discovery Interoperability

Service discovery protocols (SDPs) enable services on a network to discover each other, express opportunities for collaboration, and compose themselves into larger collections that cooperate to meet an application's needs. Many academic and industry-supported SDPs have already been proposed such as UDDI or CORBA's Trading Service for the Internet, or SLP and Jini for local and ad hoc networks. Classifications for SDPs [33] distinguish between pull-based and push-based protocols. In pull-based protocols, clients send a request to service providers (distributed pull-based mode) or to a third-party repository (centralized pull-based mode) in order to get a list of services compatible with the request attributes. In push-based protocols, service providers provide their service descriptions to all clients that locally maintain a list of the available networked services. Leading SDPs in pervasive environments use a pull-based approach (Jini, SSDP), often supporting both the cen-

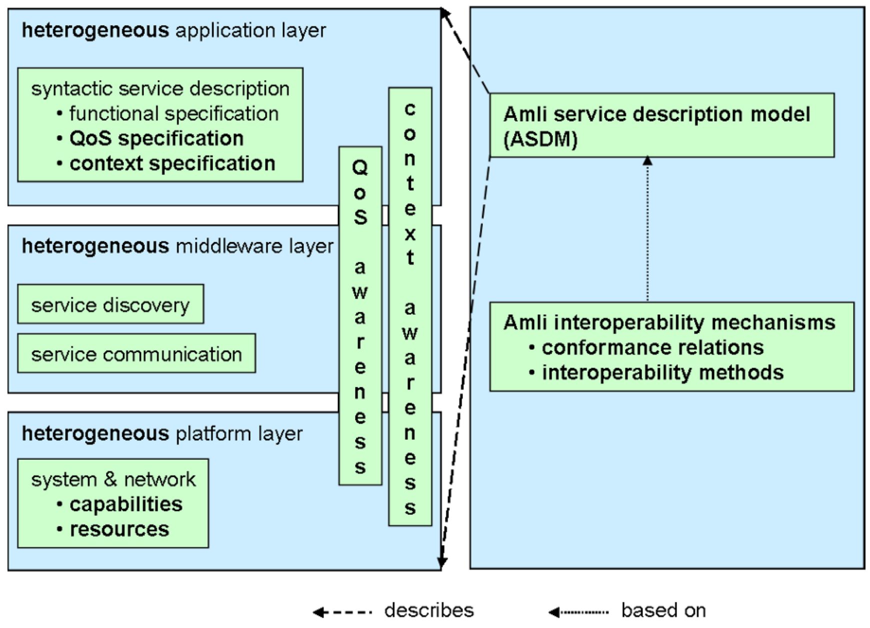

heterogeneous application layer
syntactic service description
• functional specification
• QoS specification
• context specification
heterogeneous middleware layer
service discovery
service communication
heterogeneous platform layer
system & network
• capabilities
• resources
QoS awareness
context awareness
Amli service description model (ASDM)
Amli interoperability mechanisms
• conformance relations
• interoperability methods
describes
based on

Fig. 1 The Amigo abstract reference service architecture

tralized and distributed modes of interaction (SLP, WS-Discovery). In centralized pull-based discovery protocols, one or a few repositories store the descriptions of the available services in the network, and their location is either well-known (e.g., UDDI) or dynamically discovered (e.g., Jini). Repositories are usually kept up to date by requiring explicit sign-off or by removing entries periodically. If multiple repositories exist, they cooperate to distribute the service registrations among them or to route requests to the relevant repository according to pre-established relationships.

Although many SDP solutions with well-proven protocol implementations are now available, middleware heterogeneity raises interoperability issues between the different SDPs (e.g., SLP, SSDP, UDDI) active in the environment. Existing SDPs do not directly interoperate with each other as they employ incompatible formats and protocols for service descriptions or discovery requests, and also use incompatible data types or communication models. In any case, the diverse environment constraints and the de facto standard status of some of the existing protocols make it unlikely for a global, unique SDP to emerge. Several projects have thus investigated interoperability solutions [20, 12, 15], as requiring clients and service providers to support multiple SDPs is not realistic. SDP interoperability is typically achieved using intermediate common representations of service discovery elements (e.g., service description, discovery request) [8] instead of direct mappings [15], as the latter does not scale well with the number of supported protocols. Furthermore, the interoperability layer may be located close to the network layer [8], and efficiently and transparently translate network messages between protocols, or may provide an explicit interface [24] to clients or services so as to extend existing protocols with advanced features such as context management [25].

Furthermore, the matching of service requests and service advertisements is classically based on assessing the syntactic conformance of functional and non-functional properties. However, an agreement on a common syntactic standard is hardly achievable in open environments. Thus, higher-level abstractions, independent of the low-level syntactic realizations specific to the technologies in use, should be employed for denoting service and context semantics [3]. A number of approaches for semantic service specification have been proposed, and in particular for semantic Web services such as OWL-S [18] or SAWSDL. In addition, it has been shown that efficient semantic service discovery can be performed efficiently, which is a key requirement for the resource-limited devices found in pervasive environments [4]; supporting solutions lie in encoding ontology concepts off-line and adequately classifying service descriptions in the repository based on these encoded concepts.

While the essence of the above issues is well understood, and individual solutions have been proposed that may form the foundations of a comprehensive service discovery solution for pervasive environments, a number of problems remain, or arise from such combination. First and foremost, syntactic-based and semantic-based solutions have mostly been considered separately. Indeed, interoperability solutions enabling multi-protocol SDP have focused on syntactic SDPs. At the same time, semantic-based SDPs neither manage protocol nor network heterogeneity, and context-aware SDPs assume the consistent use of a common ontology by all clients and providers. A better integration of the semantic and syntactic worlds is required, which is supported by the AmIi interoperability solution to service discovery, as presented in Sect. 3.

2.3 Service Communication Interoperability

In a dynamic open networked environment, applications/services need to adapt themselves to the context by switching, for instance, on the fly, their communication protocol. This is currently not feasible, as the way service clients and providers are designed depends strongly on the middleware upon which they are developed. Thus, applications can not be decoupled from their underlying middleware. For instance, considering RPC-based communication, a RMI client can not switch on the fly its communication protocol to interact with a CORBA service and *vice versa* unless coupled with some interoperability system.

The above issue outlines the need for a system enabling interoperability among middleware communication protocols. A number of such have been introduced since the emergence of middleware. These include middleware bridges, which can be direct or indirect. Direct bridges (e.g., RMI-IIOP[8], IIOP-.NET[9]) provide interoperability between two fixed middleware, whereas, indirect bridges assume the predominance of one specific middleware that acts as an intermediary [29]. Bridges may appear as an attractive solution to provide interoperability. However, bridges are not suitable for dynamic open networks, and in particular those that are formed in an ad hoc manner, where the communication protocols used are not known in advance. Applications and/or services are strongly coupled to a given bridge and hence are not able to switch on the fly their communication protocol according to the networked environment context. Indeed, direct or indirect bridges are a static mean (i.e., fixed at design or possibly deployment time) to overcome middleware heterogeneity as it is expected to know, in advance, between which heterogeneous communication protocols interoperability is required.

One possible solution to overcome the above constraint is to decouple applications/services from their bridge through proxy-based bridging, which acts as an intermediary. The proxy then encapsulates one or more bridges and hides their implementation details to applications/services, enabling thus to change transparently the bridge used according to the networking context. The most famous implementation of such a mechanism is the Java dynamic proxy. Still, its shortcomings are: (i) proxies are platform-specific; and (ii) if applications/services are not aware in advance of the bridges to be used, they need to have or to dynamically download the code of all possibly needed bridges – a dedicated bridge is needed for each pair of heterogeneous middleware – which can be very resource-consuming.

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

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

Greater flexibility to bridge-based interoperability is brought by Enterprise Service Buses (ESBs), which address the above shortcomings. An ESB [9] is a server infrastructure that acts as an intermediary among heterogeneous middleware through the integration of a set of reusable bridges. Applications/services are freed from the management overhead of interoperability. However, this requires an ESB server to be deployed and configured in the network. It is not reasonable to consider that there exists such a server in each open networked environment joined by mobile devices and in particular in ad hoc networks. The extended home environment that was targeted by Amigo is potentially such open environment. As we can not do any assumption about software infrastructure in such environment, interoperability must possibly be managed by the networked devices themselves through the use of middleware, like e.g., ReMMoC.

ReMMoC is one of the pioneering middleware introducing an interoperability system for open (wireless) networks [12]. In its latest version, ReMMoC is a Web Service-based reflective middleware that uses the Web Services abstraction (abstract part of a WSDL document), to abstract to applications the concrete communication protocol used to invoke remote services. Indeed, REMMoC, thanks to its reflection mechanisms [31], selects dynamically the most appropriate communication protocol according to the context. Although ReMMoC is currently one of the most efficient and innovative middleware to perform interoperability, it is confronted to some constraints. First, client applications must be developed using the ReMMoC middleware. Thereby, interoperability is available only to ReMMoC-based clients. In addition, ReMMoC is dedicated to client applications, excluding thus interoperability to service providers. Providing interoperability to service providers enables clients, which are not interoperable (e.g., not based on ReMMoC) to still interoperate with services that are based on a different communication protocol.

In a way similar to ReMMoC, RMIX is a middleware that permits transparent dynamic binding with multiple communication protocols [16]. The RMIX originality comes from its programming model that is based on RMI. Hence, the reengineering of existing RMI applications, to take benefit of RMIX, is reduced to a minimum. As a result, RMIX is dedicated to Java and uses functionalities inherent to the Java platform. Thus, interoperability is restricted to Java-compliant devices and/or services. The need to embed a Java Virtual Machine (JVM) and to rewrite non-Java applications to be interoperable is a strong limitation. The same applies to OSGi[10], which is a popular Java-based middleware that provides the capability to integrate different communication protocols for OSGi-specific applications.

Summarizing, from the above survey of existing solutions to middleware communication interoperability, there is, to the best of our knowledge, no satisfying solution to middleware interoperability for open dynamic networks, which has led us to develop the AmIi interoperability solution to service communication, as detailed in Sect. 4.

[10] http://www.osgi.org/

3 AmIi Interoperable Service Discovery

In Sect. 2, we identified the AmIi service description model (ASDM) and the thereupon based AmIi interoperability mechanisms (comprising conformance relations and interoperability methods) as the elements that enable interoperability in the Amigo reference architecture. In this section, we detail our AmIi interoperable service discovery solution (*AmIi-SD*), which includes: (i) the design of the ASDM conceptual model and a concrete realization of the model, the *AmIi Service Description Language (ASDL)*; (ii) conformance relations for matching heterogeneous services based on ASDL; and (iii) a repository-based service discovery mechanism.

The ASDM model serves as basis for enabling mapping between heterogeneous service description languages, including both syntactic- and semantic-based languages, used by existing service discovery protocols (SDPs). Specifically, service descriptions given using syntactic-based languages (e.g., UPnP, SLP, WSDL) and semantic-based languages (e.g., SAWSDL, OWL-S, WSMO) can be translated to ASDM-based (more specifically ASDL-based) descriptions. As a result, a service request may be matched against a service description, even if they are expressed in different service description languages.

Interoperable service discovery is then achieved by deploying multi-SDP repositories in the network. Repositories run plugins associated with legacy SDPs and are thus able to serve requests and catch service advertisements from the various SDPs. Concretely, as depicted in Fig. 2, the *AmIi service repository* is a (logically) centralized repository that enables service discovery within the pervasive environment. Specific *legacy SDP plugins* register with the active SDPs in the network, and translate requests and advertisements from legacy formats to ASDL (e.g., UPnP in Fig. 2). Depending on the specific SDP, the legacy plugin either directly performs service discovery (i.e., pull-based only protocols) or registers for service advertisements (i.e., push-based and hybrid protocols). In the latter case, the ASDL description generated from a service announcement is sent to the *ASDL descriptions directory* for storage. Additionally, the AmIi repository provides an explicit

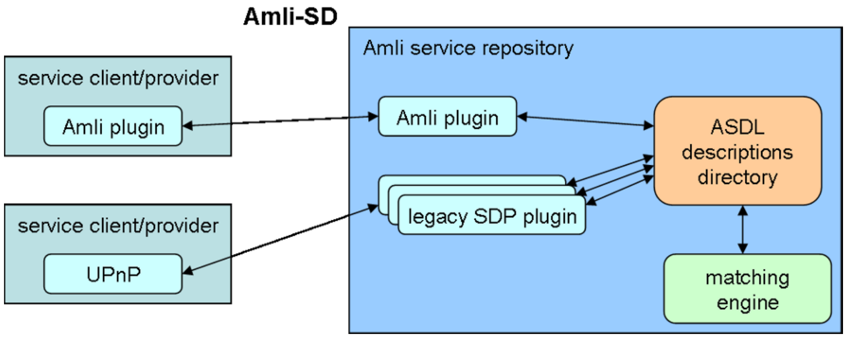


Fig. 2 AmIi interoperable service discovery

API supported by the *AmIi plugin* that enables clients (resp. providers) in a network to issue service requests (resp. advertisements) directly in the ASDL format, thus benefiting from all its advanced semantic features. In this case, the ASDL directory stores the ASDL descriptions issued by the AmIi plugin. Finally, the *matching engine* combines various conformance relations to support both syntactic-based and semantic-based service descriptions (included in requests or advertisements), and thus provides interoperability between SDPs. When there is an incoming service request, it is translated by the appropriate plugin, and then the matching engine matches it against the ASDL directory.

In the next sections, we present the ASDM model (Sect. 3.1) and its ASDL instantiation (Sect. 3.2). We then detail our interoperable matching relations (Sect. 3.3) and method for ranking the matching results (Sect. 3.4).

3.1 ASDM: A Model for Semantic and Syntactic Service Specification

The design of ASDM results from the analysis of many existing service description languages; it further builds upon two models coming from our previous work: (i) the MUSDAC service model [25], which supports interoperability between syntactic-based languages (e.g., UPnP, SLP and WSDL); and (ii) the EASY service model [4] for the semantic specification of service functional and non-functional capabilities. ASDM then extends these two models to account for both semantic- and syntactic-based descriptions, as well as non-functional properties associated with services. In this chapter, due to the lack of space, we do not present features related with the complete specification and matching of service non-functional properties; the interested reader is referred to [1].

In ASDM, a *service* description is composed of two parts: a *profile* and a *grounding* (see corresponding UML diagram in Fig. 3). The service profile is described as a non-empty set of *capabilities* and a possibly empty set of *non-functional properties*, while the service grounding prescribes the way of accessing the service. A service capability is any functionality that may be provided by a service and sought by a client. It is described with: its *name*; its interface comprising a possibly empty set of *inputs* and a possibly empty set of *outputs*; a potential *conversation*; and a possibly empty set of non-functional properties. Capabilities that do not have any associated input/output descriptions are those retrieved by legacy SDPs that simply use names to characterize capabilities (e.g., a native SLP service). Inputs associated with a capability are the information necessary for the execution of the capability, while outputs correspond to the information produced by the capability. Inputs and outputs of capabilities are described with their *names*, *types*, and a possible *semantic annotation* that is a reference to a concept in an existing ontology. Capabilities may themselves have a semantic annotation. Capabilities that do not have semantic annotations associated to them or to their inputs and outputs are those provided by legacy services without an enriched semantic interface (e.g., a native UPnP service).

A conversation associated with a capability prescribes the way of realizing this capability through the execution of other capabilities. This conversation is described as a workflow of activities that may correspond either to elementary or composite capabilities. Elementary capabilities are those that do not have a conversation, while composite capabilities are those that are themselves composed of other capabilities. In concrete terms, an elementary capability represents a basic interaction with a service. For instance, in the case of SLP services, it corresponds to the invocation of the whole service, whereas in the case of UPnP or Web Services it corresponds to the invocation of one of the service operations. Conversations are said to be semantic if they involve capabilities that have associated semantic annotations. They are said to be syntactic otherwise.

Further in ASDM, non-functional properties are related with context and QoS information of services. They are expressed at two levels: at the service (profile) level and at the capability level. Non-functional properties defined at the service level are those that apply to all the capabilities of the service. For instance, a property given at the service level and describing that the service encrypts its messages using a particular encryption algorithm means that all the capabilities of the service employ the same encryption algorithm. On the other hand, a property given at the capability level and expressing a latency property, for instance, concerns only the capability itself. Non-functional properties have semantic annotations associated to them.

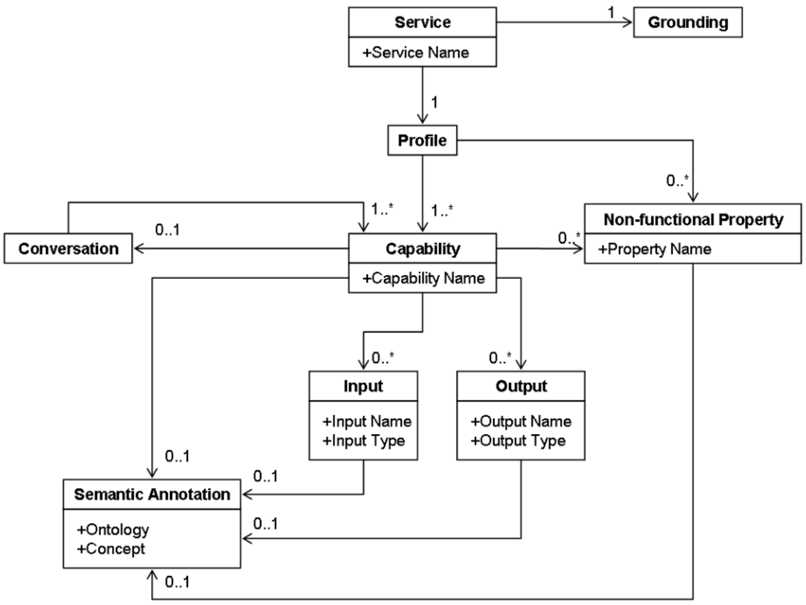
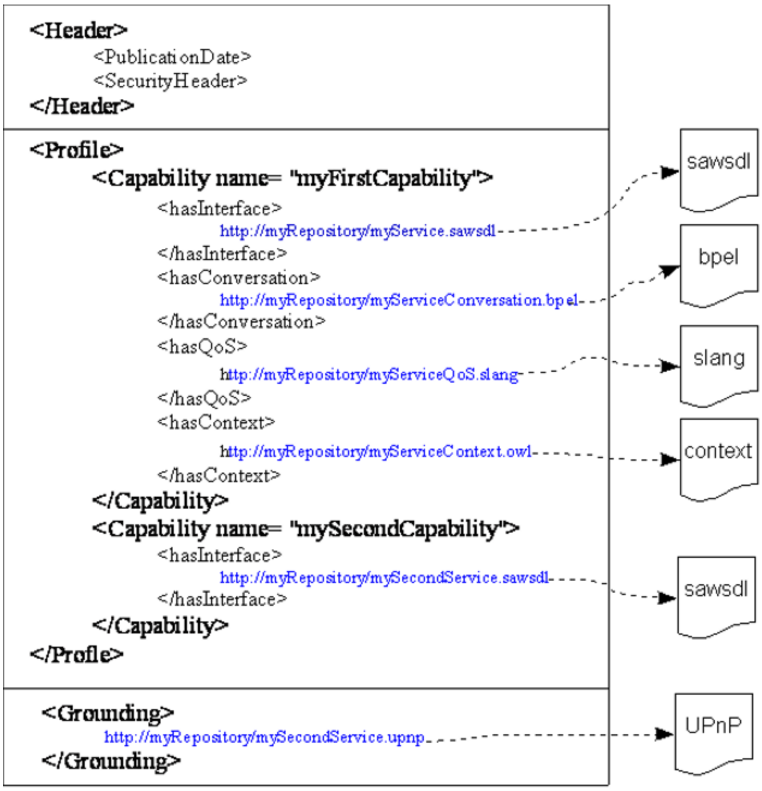


Fig. 3 The ASDM model

3.2 ASDL: A Language for Semantic and Syntactic Service Specification

Next to the ASDM conceptual model, we introduce the AmIi Service Description Language (ASDL) as a concrete realization of the model. For the implementation of ASDL, we opted for an XML-based schema defining a container, which is combined with the two emergent standard service description languages, namely SAWSDL and WS-BPEL. The ASDL description acts primarily as a top-level container for additional files describing facets of the service. SAWSDL is used to describe the capability interfaces, while WS-BPEL is used to express conversations associated with capabilities. We employ SAWSDL for the definition of capability interfaces because it supports both semantic and syntactic specification of service attributes (e.g., inputs, outputs). Thus, both legacy syntactic descriptions and rich semantic descriptions can be translated to SAWSDL. On the other hand, WS-BPEL is a comprehensive language for workflow specification, which is adequate for conversation specification. It has largely been adopted both in the industrial community and in academia.



```
<Header>
        <PublicationDate>
        <SecurityHeader>
</Header>
<Profile>
        <Capability name= "myFirstCapability">
                <hasInterface>
                        http://myRepository/myService.sawsdl
                </hasInterface>
                <hasConversation>
                        http://myRepository/myServiceConversation.bpel
                </hasConversation>
                <hasQoS>
                        http://myRepository/myServiceQoS.slang
                </hasQoS>
                <hasContext>
                        http://myRepository/myServiceContext.owl
                </hasContext>
        </Capability>
        <Capability name= "mySecondCapability">
                <hasInterface>
                        http://myRepository/mySecondService.sawsdl
                </hasInterface>
        </Capability>
</Profle>
  <Grounding>
        http://myRepository/mySecondService.upnp
  </Grounding>
```

Fig. 4 An ASDL description

WS-BPEL supports only syntactic conversation specification, however, if combined with SAWSDL, semantic conversations can be defined. Additional files may be optionally linked to the ASDL container to describe a service's non-functional properties using existing QoS and context models (e.g., SLAng [17], EASY [4]). Fig. 4 shows an example of an ASDL description where the service is composed of two capabilities. The first capability has a functional and non-functional description that comprises a reference to a SAWSDL file defining the capability interface, a WS-BPEL description that defines the conversation associated with the capability, as well as QoS and context descriptions given in a SLAng and an OWL file respectively. The second capability of this service is only given with an interface description defined in a SAWSDL file.

Given ASDL, heterogeneous service descriptions can be translated into reference ASDL descriptions by using translators associated with service description languages that come along with legacy SDPs, e.g., UPnP2ASDL (see legacy SDP plugins in Fig. 2). It is also possible that a service provider makes available a rich semantic description of a service in ASDL, thus exploiting the expressiveness of the language set included in the ASDL container (see AmIi plugin in Fig. 2). ASDL descriptions are used to assess the conformance of services against service requests.

Fig. 5 gives an overview of how various legacy service descriptions are translated into ASDL, where we focus on functional aspects of services. In this figure, five different scenarios are identified:

1. A legacy service specified with the name of its provided functionality (e.g., a SLP service). The produced ASDL description contains the SLP grounding information and links to a SAWSDL description that contains a single operation having as name the name of the SLP service without any input or output specification.
2. A service that provides a list of operations described syntactically with their signatures, as it is the case for UPnP services or web services. The produced ASDL description links to a SAWSDL description that comprises a list of WSDL operations corresponding to the operations specified in the legacy description without semantic annotations.
3. A service described as a set of semantically annotated operations (e.g., given as a SAWSDL description). The produced ASDL description can link directly to the given SAWSDL description file or, in the case of a description other than SAWSDL, after mapping the given description to SAWSDL.
4. A syntactic capability described with an associated conversation of operations (e.g., a service described as a WSDL operation that is realized through the execution of a WS-BPEL conversation). The generated ASDL description contains the specification of both an interface and a conversation. The interface points to a SAWSDL description that contains a single operation without semantic specification and is used to describe the capability. The conversation links to a WS-BPEL description that describes the conversation associated with the operation. This WS-BPEL description uses itself another WSDL file that specifies the operations used in the conversation.

5. A semantic capability having an associated conversation of semantic operations (e.g., an OWL-S service with a profile that describes the semantic capability and a process model that describes the associated conversation). The generated ASDL description comprises both an interface and a conversation, as in the previous scenario. However, contrary to the previous scenario, the SAWSDL description used to describe the capability includes semantic annotations of the capability elements (i.e., inputs, outputs). Furthermore, the WS-BPEL file describing the conversation associated with the capability uses another SAWSDL description in which the operations used in the conversation are also semantically annotated.

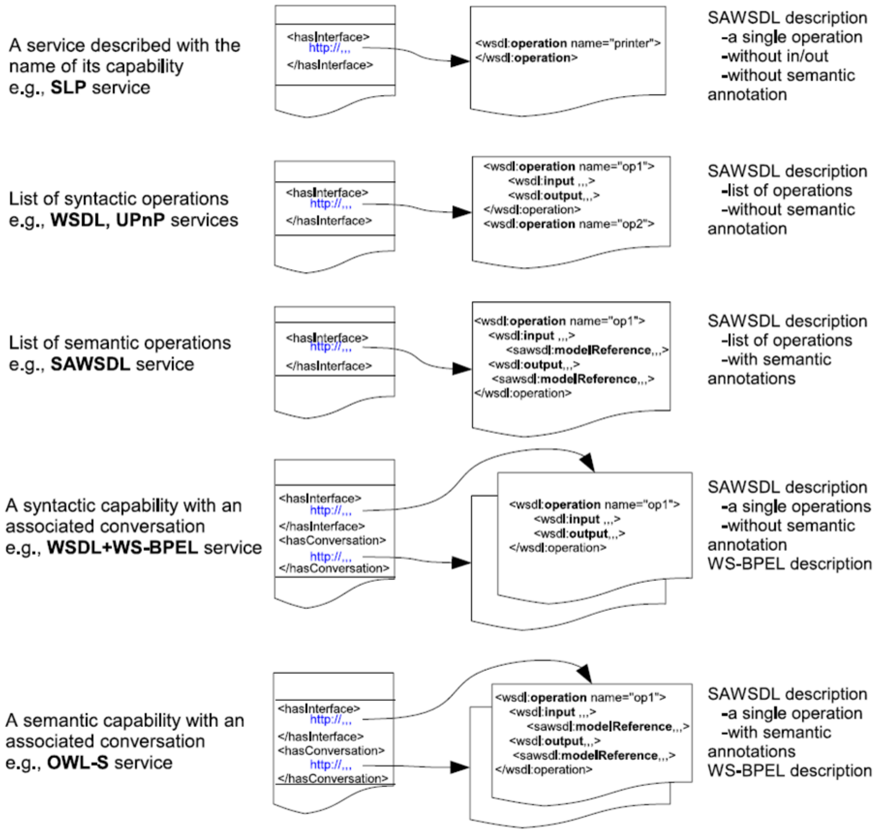

A service described with the name of its capability e.g., SLP service
<hasInterface> http://,,, </hasInterface>
<wsdl:operation name="printer"> </wsdl:operation>
SAWSDL description -a single operation -without in/out -without semantic annotation
List of syntactic operations e.g., WSDL, UPnP services
<wsdl:operation name="op1"> <wsdl:input ,,,> <wsdl:output,,,> </wsdl:operation> <wsdl:operation name="op2">
SAWSDL description -list of operations -without semantic annotation
List of semantic operations e.g., SAWSDL service
<wsdl:operation name="op1"> <wsdl:input ,,,> <sawsdl:modelReference,,,> <wsdl:output,,,> <sawsdl:modelReference,,,> </wsdl:operation>
SAWSDL description -list of operations -with semantic annotations
A syntactic capability with an associated conversation e.g., WSDL+WS-BPEL service
<hasConversation> http://,,, </hasConversation>
SAWSDL description -a single operations -without semantic annotation WS-BPEL description
A semantic capability with an associated conversation e.g., OWL-S service
SAWSDL description -a single operation -with semantic annotations WS-BPEL description

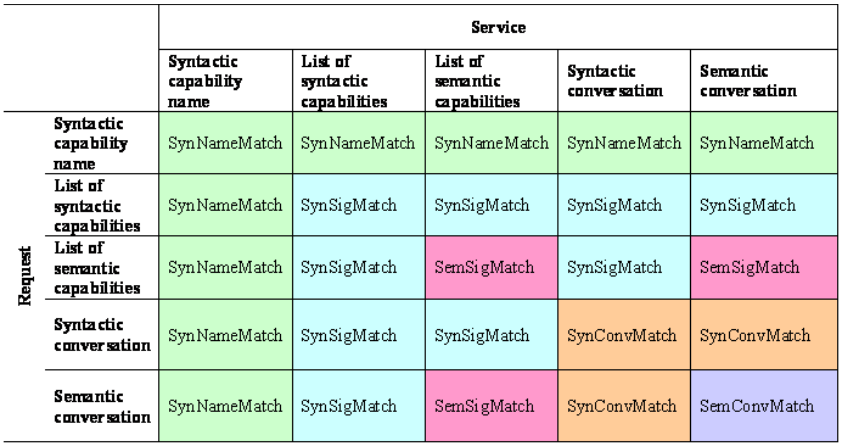
Fig. 5 Various legacy service descriptions translated into ASDL

3.3 Interoperable Matching of Service Capabilities

Based on their common mapping to ASDL descriptions, heterogeneous service descriptions (a service request and a service advertisement) can be matched. We introduce a set of conformance relations for matching the various cases of heterogeneous service descriptions, inspired from the scenarios of the previous section. These conformance relations enable different degrees of matching, from basic capability name matching to advanced semantic conversation matching. The choice of the appropriate conformance relation depends on how much information is given in the service descriptions. Obviously, in each case, the highest degree of matching possible is limited by the greatest common denominator of information given in the two service descriptions. For instance, comparing a service request described as a syntactic capability name (first scenario) with a rich semantic service description (third scenario) requires ignoring the semantic service annotations as well as the input and output information and performing a syntactic comparison of the request with the names of the service capabilities.

The different cases of matching of heterogeneous service descriptions are outlined in the table depicted in Fig. 6. In this table, following the five scenarios of the previous section, a service request and a service advertisement can be described as: (1) a syntactic capability name; (2) a list of syntactic capabilities; (3) a list of semantic capabilities; (4) a syntactic capability with an associated syntactic conversation; and (5) a semantic capability with an associated semantic conversation.

Fig. 6 shows twenty-five cases of matching a service request with a service advertisement. Among these cases, we can identify certain redundancy by applying the greatest common denominator rule discussed above. This leads us to introduce the following five matching relations:



		Service				
		Syntactic capability name	**List of syntactic capabilities**	**List of semantic capabilities**	**Syntactic conversation**	**Semantic conversation**
Request	**Syntactic capability name**	SynNameMatch	SynNameMatch	SynNameMatch	SynNameMatch	SynNameMatch
	List of syntactic capabilities	SynNameMatch	SynSigMatch	SynSigMatch	SynSigMatch	SynSigMatch
	List of semantic capabilities	SynNameMatch	SynSigMatch	SemSigMatch	SynSigMatch	SemSigMatch
	Syntactic conversation	SynNameMatch	SynSigMatch	SynSigMatch	SynConvMatch	SynConvMatch
	Semantic conversation	SynNameMatch	SynSigMatch	SemSigMatch	SynConvMatch	SemConvMatch

Fig. 6 Interoperable matching of services capabilities

1. Syntactic matching of capability names, noted **SynNameMatch()**. It applies when the service request or the service advertisement is described only as a syntactic capability name. This function is defined as follows:

$$\begin{aligned} &SynNameMatch(C_{adv}, C_{req}) = \\ &C_{adv}.CapabilityName = C_{req}.CapabilityName \end{aligned}$$

2. Syntactic signature matching, noted **SynSigMatch()**. It applies – to the remaining cases – when the request or the advertisement is described as a list of syntactic capabilities, or when additionally one of them is described as a syntactic conversation but the other side provides no conversation. The SynSigMatch() function is based on syntactic matching of the capability names and of the inputs and outputs of the corresponding capabilities of the request and the advertisement; it is defined as follows:

$$\begin{aligned} &SynSigMatch(C_{adv}, C_{req}) = \\ &C_{adv}.CapabilityName = C_{req}.CapabilityName \\ &\forall in_{adv} \in C_{adv}.Input, \exists in_{req} \in C_{req}.Input : \\ &in_{adv}.Name = in_{req}.Name \wedge in_{adv}.Type = in_{req}.Type \\ &\forall out_{req} \in C_{req}.Output, \exists out_{adv} \in C_{adv}.Output : \\ &out_{req}.Name = out_{adv}.Name \wedge out_{req}.Type = out_{adv}.Type \end{aligned}$$

3. Semantic signature matching, noted **SemSigMatch()**. It applies when both the request and the advertisement are described as a list of semantic capabilities, or when additionally one of them is described as a semantic conversation but the other side provides no conversation. The SemSigMatch() function is based on semantic matching of the capabilities and of the inputs and outputs of the corresponding capabilities of the request and the advertisement; it is defined below. It is based on the **ConceptMatch()** function, which is used to check whether two concepts are related in an ontology, i.e., if they are equivalent or one is more generic than the other [4]. If semantic conformance is established, syntactic adaptation should then be performed between the syntactic signatures of the capabilities.

$$\begin{aligned} &SemSigMatch(C_{adv}, C_{req}) = \\ &ConceptMatch(C_{adv}.CapabilityName, C_{req}.CapabilityName) \\ &\forall in_{adv} \in C_{adv}.Input, \exists in_{req} \in C_{req}.Input : \\ &ConceptMatch(in_{adv}.SemanticAnnotation, in_{req}.SemanticAnnotation) \\ &\forall out_{req} \in C_{req}.Output, \exists out_{adv} \in C_{adv}.Output : \\ &ConceptMatch(out_{req}.SemanticAnnotation, out_{adv}.SemanticAnnotation) \end{aligned}$$

4. Syntactic conversation matching, noted **SynConvMatch()**. It applies when both the request and the advertisement are described as conversations, but at least one

of them is only syntactic. First, the SynSigMatch() function is used to match the request and the advertisement based on their signature. Then, matching of conversations is performed. As WS-BPEL can be attributed formal semantics based on process algebras, conformance between the requested and provided conversations can be assessed using process bi-simulation. In the case where no service that provides an equivalent conversation to the request is found, we may employ mechanisms for dynamic composition of heterogeneous services to reconstruct the requested conversation, as addressed by our previous work reported in [2].

5. Semantic conversation matching, noted **SemConvMatch()**. It applies when both the request and the advertisement are described as semantic conversations. This case is similar to the previous one, except that we employ the SemSigMatch() function for the initial matching.

3.4 Ranking Heterogeneous Matching Results

As discussed in the previous sections, service descriptions are heterogeneous in terms of their expressiveness, going from very simple syntactic definitions to very rich semantic definitions with associated conversations. Thus, we defined five matching relations to assess the conformance of heterogeneous service advertisements with respect to a particular service request.

In the case of having multiple service advertisements matching a service request, we should further be able to select the service that best matches the request. This requires being able to rank the heterogeneous matching results. Ranking such results is dependent upon the expressiveness of service requests and service advertisements. For instance, services that have semantic annotations are preferred to syntactic services when the request is given with semantic annotations. Furthermore, semantic services that match a semantic request should be ranked with respect to their *semantic distance* to the request. Semantic distance allows evaluating the degree of conformance of a semantic service capability with respect to a request. We present in this section our mechanism for ranking service advertisements with respect to a service request. First, according to the degree of expressiveness of the service request, results coming from our five matching relations are ranked according to the table in Fig. 7; where $m_1 > m_2$ means that matching m_1 is more accurate than m_2 and thus preferred.

Second, the results of the semantic signature matching function, i.e., SemSigMatch(), are themselves classified according to their degree of conformance to the given service request. The degree of conformance between a semantic request and a semantic service advertisement is evaluated using the function **SemSigDegreeOfMatch()**, which sums the results of **ConceptDegreeOfMatch()** for all concepts matched in the matching phase for SemSigMatch(), as follows:

$$SemSigDegreeOfMatch(C_{adv}, C_{req}) = \sum_{i=1}^{n} ConceptDegreeOfMatch(c_i, c_i'), \forall c_i, c_i' : ConceptMatch(c_i, c_i')$$

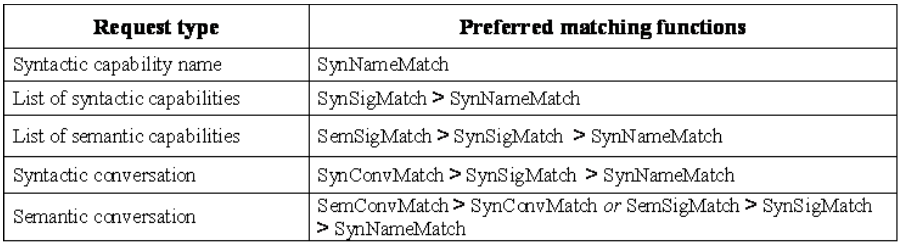
Where $ConceptDegreeOfMatch(c_1, c_2)$ returns the number of levels that separate the concepts c_1 and c_2 in the ontology hierarchy that contains them [4].

4 AmIi Interoperable Service Communication

Services located through the AmIi interoperable service discovery (AmIi-SD) may embed various middleware communication protocols. It is then necessary to overcome such heterogeneity, so that interaction with these services may be possible independently of the middleware technologies embedded on the client's and provider's sides. In this section, we present our AmIi solution to interoperable service communication (*AmIi-COM*) [7], which is based on runtime protocol translation. We more specifically focus on interoperability achieved among heterogeneous RPC protocols, as this is an essential communication model in service oriented computing. In the following, we first discuss the relation between AmIi-COM and AmIi-SD (Sect. 4.1). We then detail our AmIi-COM solution. More specifically, we recall the characteristics of RPC communication protocols (Sect. 4.2), in order to introduce efficient event-based techniques to overcome RPC protocol heterogeneity (Sect. 4.3). Finally, Sect. 4.4 discusses the deployment and configuration of AmIi-COM.

4.1 Interoperable Service Discovery and Communication

In the previous sections, we already discussed discovery of services. We herein recall some of the elements of this process and see how it is coupled with service communication.



Request type	**Preferred matching functions**
Syntactic capability name	SynNameMatch
List of syntactic capabilities	SynSigMatch > SynNameMatch
List of semantic capabilities	SemSigMatch > SynSigMatch > SynNameMatch
Syntactic conversation	SynConvMatch > SynSigMatch > SynNameMatch
Semantic conversation	SemConvMatch > SynConvMatch *or* SemSigMatch > SynSigMatch > SynNameMatch

Fig. 7 Ranking heterogeneous matching results

In order to interact with services in open networked environments, clients must first find remote services using some SDP. Then, they rely on specific information that they get about the discovered services to actually interact with them. Clients look up the information needed to interact with services in service repositories, which are logical centralization points for such information. Each RPC communication middleware depends on a dedicated repository. For instance, Web Services, which are based on the SOAP protocol, use UDDI[11], whereas the RMI and CORBA middleware use repositories respectively called rmiregistry and CORBA Naming/Trading Service.

Hence, remote services must first publish/export their description to a repository to be accessed by clients (Fig. 8, Step 1). Through the export process, services advertise their interface, communication protocol and unique reference/address. The former is a set of methods describing the service's communication contract, whereas the latter two provide a mean to locate and access the service's instance. This data enables producing a client-side stub that acts as a proxy for the remote service. Clients then use the stub as a handle to make method calls to the remote service. The way stubs are produced and obtained by clients may differ from one middleware to another. Stubs can be obtained statically or dynamically. In the former case, stubs are generated at development-time, so that clients do not need to get them at run-time; still, clients retrieve at run-time from the repository the service's reference/address, which they feed into the static stub. In the latter case, stubs are transparently created by the export process and registered to the repository (Fig. 8, Step 2); in this case, clients retrieve the whole stub. In the following, we consider only the dynamic stub case as representative of both cases. The repository's location is either known in advance by the client or dynamically discovered using some SDP. Once the client gets the stub (Fig. 8, Step 3), it can interact with the desired service. To invoke a method on the remote service, the client makes a local call on the stub (Fig. 8, Step 4). The latter first marshals the call into a request message according to the communication protocol used by the middleware (e.g., IIOP for CORBA, JRMP for RMI) and then sends the message to the remote service (Fig. 8, Step 5). Hence, clients are not aware of the implementation specifics of services; stubs abstract their location, programming language and communication protocol. Finally, on the ser-

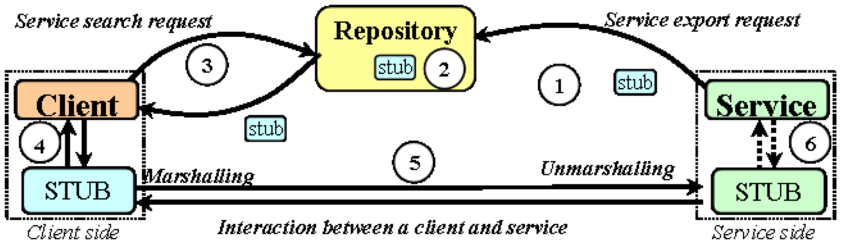


Fig. 8 RPC-based middleware architecture

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

vice side, the incoming request message is unmarshalled by the service-side stub into a local call (Fig. 8, Step 6). Stubs are not always mandatory; some clients may dynamically generate method calls to the remote service (e.g. CORBA DII).

In Amigo, we extended the above essential mechanism to deal with interoperability among heterogeneous discovery and communication protocols. As introduced in Sect. 3, the AmIi service repository assumes the role of a universal repository for different SDPs. Depending on the specific mechanism of each supported SDP and RPC communication middleware coupled with it, the AmIi repository enables export and retrieval of stubs as depicted in Fig. 9 (Steps 1 and 2). More specifically, by using its native SDP, the service exports its stub, which conforms to (and indicates) its native communication middleware. On its side, by using its native SDP, the client looks up the service. The native communication middleware of the client is inferred from its SDP – as indicated above, a communication middleware employs a standard SDP. Then, a translation is performed between the stub exported by the service and the stub to be retrieved by the client, as either stub conforms to its native communication middleware. The above procedure allows providing the appropriate stub to the requesting client. The retrieved stub has been further customized – once used – to tunnel interaction between the client and service through the AmIi communication interoperability mechanism. AmIi-COM deals with communication protocol translation between the client and the service by employing appropriate protocol units (see Fig. 9, Steps 3 and 4). To enable AmIi-COM to configure appropriate protocol units, the AmIi repository dispatches to the former its knowledge about the communication protocols and also the references/addresses of the client and the service. AmIi-COM may be located on the client, the service, or even a gateway device. AmIi-COM instances register themselves with the AmIi repository in their vicinity, which allows the AmIi repository to select the appropriate AmIi-COM instance. The whole process (AmIi-COM deployment and use) is done in a decentralized and totally transparent way for the client and the service.

In the following, we detail the AmIi-COM mechanism, starting with RPC basics in the next section.

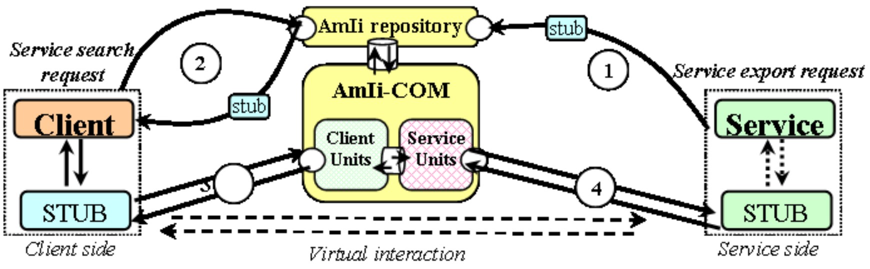


Fig. 9 Coupling between AmIi-SD and AmIi-COM

4.2 RPC Communication Stack

According to the OSI model, RPC communication protocols can be decomposed into layers, providing a functional division of the tasks required to enable successful interaction. As depicted in Fig. 10, RPC communication protocols decompose into 5 layers, defining a reference RPC communication stack. The *network* and *transport* layers are similar to the OSI ones. The former determines how data are transferred between networked devices whereas the latter specifies how to manage end-to-end message delivery among networked entities. The *invocation* layer, refining the OSI session layer, defines how to manage sessions with remote services across the network and then specifies the types of messages exchanged during an open session. Then, the *serialization* layer, refining the OSI presentation layer, encodes messages according to a format specification. Finally, the *application* layer provides to applications an interface to perform remote procedure calls.

For illustration, consider a device A hosting an RMI communication stack and another device B hosting a Web Services stack. As depicted in Fig. 11, assume an RMI-based application of A wishes to invoke a Web service of B (Fig. 11, Step 1). The corresponding request message passes through the 5 layers of the stack hosted on A (Fig. 11, Step 2). Specifically, the request is first passed to the application layer, which adds a header to the data. The resulting message is passed to the serialization layer, which adds its own header to the message it just received from its upper layer and so on, all the way down to the IP network layer. At the IP layer, the resulting message is transmitted through the network medium to B (Fig. 11, Step 3). The message should then traverse the 5 layers of the communication stack hosted on B (Fig. 11, Step 4). Each crossed layer shall extract its corresponding header and pass the message payload to the next layer and so on, all the way up to the topmost layer. Each added header contains information dedicated to the crossed layer and thus enables a direct layer-to-layer communication between the two stacks that are respectively hosted on A and B. However, although the communication stacks have a similar design, interoperability is not supported as the stacks of A and B are bound to specific message types and data format.

In the RMI stack, the serialization layer offers functions to encode/decode application data in binary format according to the Java Object Serialization Stream Protocol[12] (JOSSP) specification. In the Web Services stack, the same layer en-

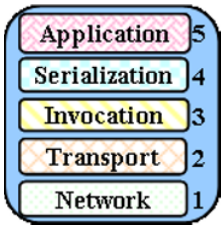


Fig. 10 RPC communication protocol stack

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

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

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

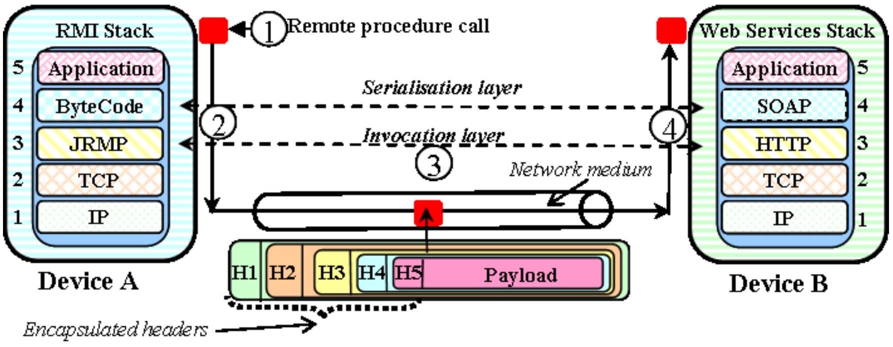


Fig. 11 Layer-to-layer communication

[13] Depending on the point of view, SOAP can be also considered as an invocation protocol

4.3 Event-based Interoperability

Following our previous work on the design of the INDISS interoperability system [8], interoperability for one layer of the communication protocol stack is the result of the composition of a *protocol parser* with a *protocol composer*. Specifically, a protocol parser generates semantic events according to input protocol messages and the protocol composer does the inverse process, either one for a specific protocol. Cooperation between a parser and composer is achievable because the parsed and produced protocols share similar functions, which are abstracted as events. An *interoperability process* is then a translation process, resulting from the composition of a parser and a composer. Thus, for each layer, we have an interoperability process based on the set of events abstracting the functionalities of the layer that are common across heterogeneous communication protocols.

According to the RPC communication stack, at least 5 interoperability processes are required to enable interoperability between two middleware based on different communication protocols. However, these processes cannot always be completely known in advance (e.g., an RMI stack may use JRMP or HTTP for the invocation layer). In this particular case, AmIi-COM can dynamically discover the structure of a remote protocol stack to select the appropriate parsers in order to create and chain the interoperability processes; and this, as long as AmIi-COM receives a message from a remote protocol stack. This is enabled by the structure of the network-layer message, as illustrated in Fig. 11. More specifically, every network message embeds the headers corresponding to the layers previously crossed. The set of headers is therefore a signature that reveals the composition of the protocol stack. Furthermore, by definition, a header always contains a magic number and/or a field to specify the current protocol used and/or the protocol expected in the next upper layer. Hence, this property enables chaining progressively the adequate parsers belonging to the different layers to generate a stream of events that semantically represents the RPC message.

The chaining of interoperability processes is depicted in Fig. 12. In Step 1, the RPC call from device A is first parsed by the *network parser*. The parser decomposes the message into two distinct parts: the header and the payload. The former is transformed into an event stream that is forwarded to the *network composer* and the payload is passed to the *transport parser*, which is the next parser in the chain. Recursively, the transport parser extracts from the received payload a new header translated into events that are sent to the *transport composer* and a new payload that is directed to the *invocation parser* and so on, all the way down to the *application parser* that finally translates the data of the RPC call into an event stream. Events from each parser are sequentially forwarded to composers (Fig. 12, Step 2). However, composers are not able to generate a message until the last parser of the chain has parsed the last payload. In fact, the composer from the bottom level generates the payload that is required for the composer of the level immediately above and so on, all the way up to the network level (Fig. 12, Step 3). The resulting message is finally compliant to the protocol stack of device B (Fig. 12, Step 4). A similar process applies to the RPC reply from B to A. Therefore, to provide bidirectional communication between two different communication protocol stacks, to each protocol layer corresponds a *protocol unit*, which embeds the protocol parser and composer for the specific protocol layer as depicted in Fig. 13. Further details about protocol units are introduced in [8] in the context of service discovery protocols. In practice, as layers are not always fully independent (e.g., SOAP defines both serialization and invocation protocols), the flow of events generated from the chain of parsers are forwarded to the chain of composers since each one of them is free to handle or ignore incoming events.

In a way similar to Horus [26, 27], Ensemble [13], Coyote [6], and Microsoft Indigo[14], protocol units are independent protocol modules or blocks that are stacked on top of each other to constitute a vertical protocol stack. However, we further introduce a dynamic composition of protocol stacks that is both vertical and horizontal. Vertical stack composition (i.e., vertical unit chaining) enables translating a RPC call to a stream of semantic events, whereas the horizontal stack composition (i.e., horizontal unit chaining) translates the stream of semantic events to another protocol (See Fig. 13). In contrast to traditional protocol stacks, the combined vertical and horizontal stack composition of AmIi-COM enables translating one communication protocol to another but does not interpret them.

Also, note that, contrary to the above systems that provide reconfiguration of protocol stacks [26, 6, 13], with AmIi-COM, applications and services are not aware of the reconfiguration of protocol compositions and are therefore not bound to the AmIi-COM system. The latter acts at the network layer on top of the operating

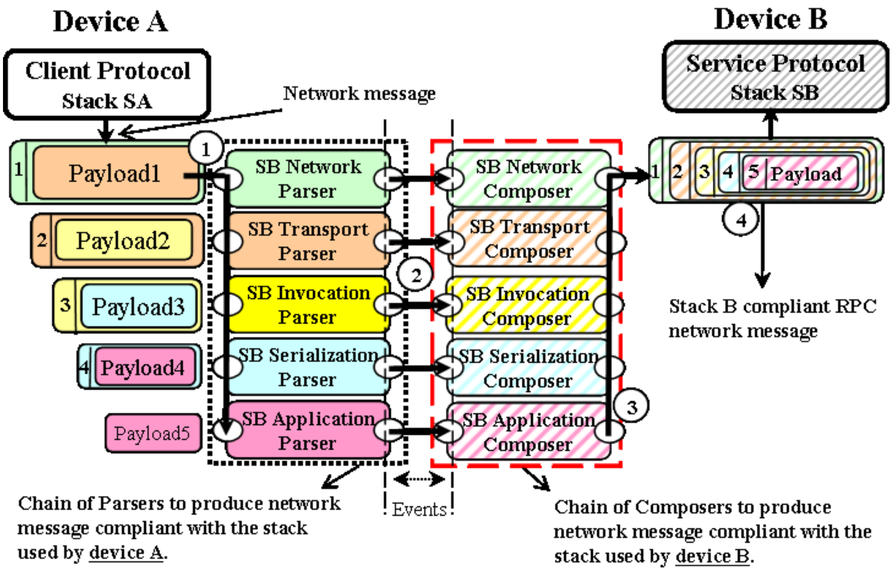


Fig. 12 Event-based interoperability

[14] http://windowscommunication.net/Default.aspx

system and below legacy middleware (See Fig. 14). Further, AmIi-COM needs only to be deployed on one of the nodes involved in the communication, whether the client or the service host, or even a gateway.

4.4 AmIi-COM Instances

In practice, there are not as many units to compose as protocol stack layers. In general, protocol stacks share a number of identical layers, thus reducing the number of units involved in protocol interoperability. For instance, as illustrated in Fig. 11, a majority of RPC protocols are based on TCP/IP, simplifying the interoperability system, which works only from Layer 3 to 4 (i.e., invocation to serialization layers). The TCP/IP drivers of the operating system act as units dedicated to Layers 1 and 2. However, if the latter are heterogeneous, our system can also enable dynamic, through adequate units, interoperability among different networks.

AmIi-COM is built around the concept of vertical and horizontal, dynamic unit chaining. However, dynamic chaining is not without cost in terms of resource con-

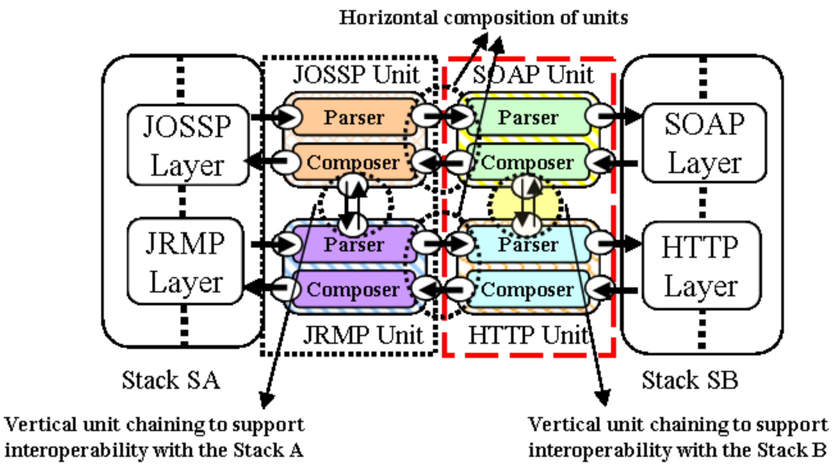

Horizontal composition of units
JOSSP Unit
SOAP Unit
JOSSP Layer
Parser
Composer
Parser
Composer
SOAP Layer
JRMP Layer
Parser
Composer
Parser
Composer
HTTP Layer
JRMP Unit
HTTP Unit
Stack SA
Stack SB
Vertical unit chaining to support interoperability with the Stack A
Vertical unit chaining to support interoperability with the Stack B

Fig. 13 Vertical and horizontal chain composition to provide interoperability

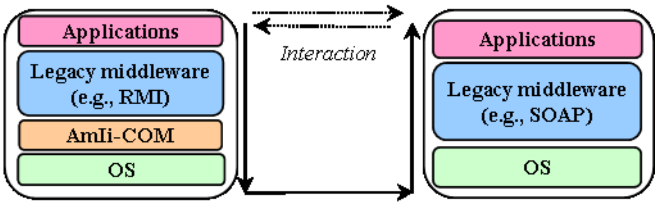


Fig. 14 Localisation of the AmIi-COM system

sumption, is not always required, and is not always possible (as discussed above, it is based on the analysis of incoming messages). In these cases, the vertical composition of protocol units for each supported protocol stack can be achieved statically. Specifically, the service discovery process (the AmIi repository of AmIi-SD) enables AmIi-COM to select the adequate vertical stack, which is statically composed (see Sect. 4.1). However, interoperability among heterogeneous stacks is still dynamic, as is the horizontal composition of protocol units.

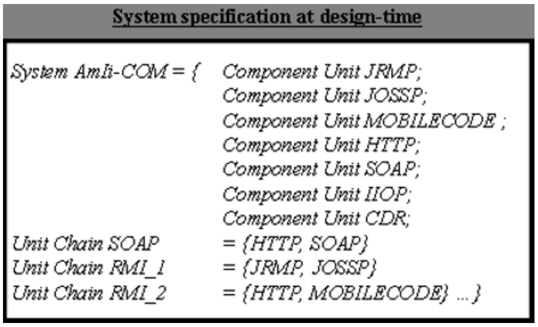
More specifically, the AmIi-COM interoperability system is defined as a set of protocol units that can be either statically or dynamically composed. As illustrated in Fig. 15, the specification of an AmIi-COM instance defines the supported units (for invocation and serialization layers) and the vertical protocol stacks that are statically composed. However, at run-time (See Fig. 16), AmIi-COM may still dynamically create new vertical stacks, or reconfigure the existing stacks, which were statically composed, by adding, removing or changing one protocol unit by another, according to the context.

Protocol units are not necessarily specific to one communication protocol and may be stacked in various ways. For instance, the vertical stack named RMI_2 in Fig. 15, which handles mobile code of RMI-based clients/services, depends on the HTTP unit, which is also used by the SOAP stack.

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

5 Conclusion

Ambient intelligence / pervasive computing environments, such as the smart networked home, should integrate networked devices, possibly wireless, from various application domains, e.g., the home automation, consumer electronics, mobile and personal computing domains. Such environments have introduced new challenges for middleware. Devices need to dynamically detect services offered by other de-


System specification at design-time

```
System AmIi-COM = {   Component Unit JRMP;
                      Component Unit JOSSP;
                      Component Unit MOBILECODE ;
                      Component Unit HTTP;
                      Component Unit SOAP;
                      Component Unit IIOP;
                      Component Unit CDR;
Unit Chain SOAP       = {HTTP, SOAP}
Unit Chain RMI_1      = {JRMP, JOSSP}
Unit Chain RMI_2      = {HTTP, MOBILECODE} ... }
```

Fig. 15 Specification of an AmIi-COM instance

vices available in the open networked environment and adapt their communication protocols to interact with them, as services are implemented on top of diverse middleware (e.g., UPnP used in the home, Java RMI in the mobile domain). Addressing such a requirement calls for enabling interoperability among networked devices throughout their system architecture, i.e., their application, middleware and platform layers. Application layer interoperability should allow networked services to meet and coordinate according to applications to be provided to users, whereas they have been developed independently without a priori knowledge of their respective specific functionalities. Significant helpers towards dealing with this requirement are the Semantic Web and in particular Semantic Web Services, which allow high level description of service functionalities and rigorous reasoning about them. However, being tied to a single service technology, such as Web Services is also restrictive; application layer interoperability should be enabled independently of service platforms. This then further calls for middleware layer interoperability (which may also include platform layer aspects), as each service technology/platform employs a specific middleware technology/platform supporting the execution and networking of services; it cannot be assumed that all networked devices in the open AmI environment will eventually converge to a unique middleware technology.

In the above context, the IST FP6 Amigo project aimed at the development of an extended home AmI environment. As part of our work in the INRIA ARLES[15] group on the development of distributed systems enabling the AmI vision, we devise enablers for pervasive services. In this chapter, we specifically concentrated on our AmIi interoperability solution developed within Amigo. We introduced a reference system architecture that has three key features: (i) it is an enhanced service architecture imposing no specific technologies; (ii) it has a full semantic description at a higher, technology-independent level; and (iii) it includes interoperability mechanisms as first-class entities based on this semantic description. Relying on these principles, we detailed in this chapter two interoperability mechanisms: (i) the AmIi interoperable service discovery (AmIi-SD), enabling service discovery across heterogeneous, both syntactic- and semantic-based, discovery protocols; and (ii) the AmIi interoperable service communication (AmIi-COM), enabling communication

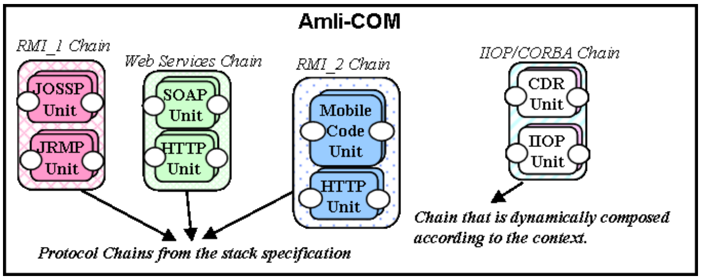


Fig. 16 AmIi-COM instances

[15] http://www-rocq.inria.fr/arles/

across heterogeneous RPC protocols. AmIi-SD and AmIi-COM work together to support complete interoperability among heterogeneous devices in the open AmI environment.

Prototypes of the related software components have been implemented as part of the Amigo project, and are available under open source software license on the Amigo open source software Web site[16] and our group's Web site[17]. Due to the lack of space, we cannot detail these implementations or report on our performance evaluation results in terms of both resource consumption and response time. Details may be found in the references included in the chapter. Performance evaluation has shown that our solutions comply with the requirements of the pervasive computing environment, i.e., resource constraints and high interactivity.

The Amigo project and the presented AmIi interoperability approach was a step forward in enabling the AmI / pervasive vision. In our future work, we aim at further removing the technology barriers that constrain this vision. AmIi, although dynamic, is based on protocol translators that are statically designed and developed to address existing service oriented architectures. In our view, the networking of systems should be completely agnostic to their specific technologies. We envisage systems that network behaviorally, as opposed to networking technologically. Systems should be able to unambiguously specify their networked functional and non-functional behavior based on adequate theoretical foundations. This should allow automated reasoning for behavioral matching and further adaptation of systems, enabling them to interact irrespectively of their underlying technologies. More specifically, building upon work in the software architecture domain, we aim to investigate formal specification of connectors that allows reasoning upon and adaptation of system networked behavior at runtime. This adaptation will be performed through on-the-fly synthesis of connectors customized for the communicating networked systems. This approach envisages "eternal systems" that can seamlessly join any networked environment, existing or possibly even future one.

References

[1] Ben Mokhtar, S.: Semantic middleware for service-oriented pervasive computing. Ph.D. thesis, University of Paris 6, France (2007)

[2] Ben Mokhtar, S., Georgantas, N., Issarny, V.: Cocoa: Conversation-based service composition in pervasive computing environments with qos support. J. Syst. Softw. **80**(12), 1941–1955 (2007). DOI http://dx.doi.org/10.1016/j.jss.2007.03.002

[3] Ben Mokhtar, S., Kaul, A., Georgantas, N., Issarny, V.: Efficient semantic service discovery in pervasive computing environments. In: Middleware '06: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Mid-

[16] https://gforge.inria.fr/projects/amigo/

[17] http://www-rocq.inria.fr/arles/

dleware, pp. 240–259. Springer-Verlag New York, Inc., New York, NY, USA (2006)

[4] Ben Mokhtar, S., Preuveneers, D., Georgantas, N., Issarny, V., Berbers, Y.: Easy: Efficient semantic service discovery in pervasive computing environments with qos and context support. J. Syst. Softw. **81**(5), 785–808 (2008). DOI http://dx.doi.org/10.1016/j.jss.2007.07.030

[5] Berners-Lee, T., Hendler, J., Lassila, O.: The semantic web. Scientific American **284**(5), 34–43 (2001)

[6] Bhatti, N.T., Hiltunen, M.A., Schlichting, R.D., Chiu, W.: Coyote: A system for constructing fine-grain configurable communication services. ACM Transactions on Computer Systems **16**, 321–366 (1998)

[7] Bromberg, Y.D.: Résolution de l'hétérogénéité des intergiciels d'un environnement ubiquitaire. Ph.D. thesis, University of Versailles-Saint Quentin en Yvelines, France (2006)

[8] Bromberg, Y.D., Issarny, V.: Indiss: Interoperable discovery system for networked services. In: Proceedings of the ACM/IFIP/USENIX 6th International Middleware Conference, pp. 164–183. Grenoble, France (2005)

[9] Chappell, D.A.: Enterprise Service Bus. O'Reilly Media (2004)

[10] Georgantas, N., Inverardi, P., Issarny, V.: Software platforms. In: E.H.L. Aarts, J.L. Encarnacao (eds.) True Visions: The Emergence of Ambient Intelligence, pp. 151–170. Springer Berlin Heidelberg (2006)

[11] Georgantas, N., Mokhtar, S.B., Bromberg, Y.D., Issarny, V., Kalaoja, J., Kantarovitch, J., Gerodolle, A., Mevissen, R.: The amigo service architecture for the open networked home environment. In: WICSA '05: Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture, pp. 295–296. IEEE Computer Society, Pittsburgh, Pennsylvania (2005). DOI http://dx.doi.org/10.1109/WICSA.2005.71

[12] Grace, P., Blair, G.S., Samuel, S.: A reflective framework for discovery and interaction in heterogeneous mobile environments. SIGMOBILE Mob. Comput. Commun. Rev. **9**(1), 2–14 (2005). DOI http://doi.acm.org/10.1145/1055959.1055962

[13] Hayden, M., van Renesse, R.: Optimizing layered communication protocols. In: Proceedings of the Sixth IEEE International Symposium on High Performance Distributed Computing, pp. 169–177. Portland, Oregon (1997). DOI 10.1109/HPDC.1997.626686

[14] Issarny, V., Sacchetti, D., Tartanoglu, F., Sailhan, F., Chibout, R., Levy, N., Talamona, A.: Developing ambient intelligence systems: A solution based on web services. Automated Software Engg. **12**(1), 101–137 (2005). DOI http://dx.doi.org/10.1023/B:AUSE.0000049210.42738.00

[15] Koponen, T., Virtanen, T.: Service discovery: a service broker approach. In: Proceedings of the 37th Annual Hawaii International Conference on System Sciences (2004). DOI 10.1109/HICSS.2004.1265669

[16] Kurzyniec, D., Wrzosek, T., Sunderam, V., Slominski, A.: Rmix: a multiprotocol rmi framework for java. In: Proceedings of the International Parallel

and Distributed Processing Symposium (2003). DOI 10.1109/IPDPS.2003. 1213269

[17] Lamanna, D.D., Skene, J., Emmerich, W.: Slang: A language for defining service level agreements. In: FTDCS '03: Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems, pp. 100–106. IEEE Computer Society, Washington, DC, USA (2003)

[18] Martin, D., Paolucci, M., Mcilraith, S., Burstein, M., Mcdermott, D., Mcguinness, D., Parsia, B., Payne, T., Sabou, M., Solanki, M., Srinivasan, N., Sycara, K.: Bringing semantics to web services: The owl-s approach. In: J. Cardoso, A. Sheth (eds.) SWSWPC 2004, *LNCS*, vol. 3387, pp. 26–42. Springer (2004)

[19] McIlraith, S.A., Martin, D.L.: Bringing semantics to web services. IEEE Intelligent Systems **18**(1), 90–93 (2003). DOI http://dx.doi.org/10.1109/MIS.2003. 1179199

[20] Nakazawa, J., Tokuda, H., Edwards, W., Ramachandran, U.: A bridging framework for universal interoperability in pervasive systems. In: Proceedings of ICDCS'06: The 26th IEEE International Conference on Distributed Computing Systems. Lisboa, Portugal (2006). DOI 10.1109/ICDCS.2006.5

[21] O'Sullivan, D., Lewis, D.: Semantically driven service interoperability for pervasive computing. In: MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access, pp. 17–24. San Diego, CA, USA (2003). DOI http://doi.acm.org/10.1145/940923.940927

[22] Papazoglou, M.P., Georgakopoulos, D.: Service-oriented computing. Commun. ACM **46**(10), 24–28 (2003). DOI http://doi.acm.org/10.1145/944217. 944233

[23] Peltz, C.: Web services orchestration and choreography. IEEE Computer **36**(10), 46–52 (2003). DOI http://dx.doi.org/10.1109/MC.2003.1236471. URL `http://dx.doi.org/10.1109/MC.2003.1236471`

[24] Raverdy, P.G., Issarny, V., Chibout, R., de La Chapelle, A.: A multi-protocol approach to service discovery and access in pervasive environments. In: Proceedings of the 3rd Annual International Conference on Mobile and Ubiquitous Systems, pp. 1–9. San Jose, CA, USA (2006). DOI http://doi. ieeecomputersociety.org/10.1109/MOBIQ.2006.340448

[25] Raverdy, P.G., Riva, O., de La Chapelle, A., Chibout, R., Issarny, V.: Efficient context-aware service discovery in multi-protocol pervasive environments. In: MDM '06: Proceedings of the 7th International Conference on Mobile Data Management. IEEE Computer Society, Nara, Japan (2006). DOI http://dx.doi. org/10.1109/MDM.2006.78

[26] van Renesse, R., Birman, K.P., Friedman, R., Hayden, M., Karr, D.A.: A framework for protocol composition in horus. In: PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, pp. 80–89. ACM, New York, NY, USA (1995). DOI http://doi.acm. org/10.1145/224964.224974

[27] van Renesse, R., Birman, K.P., Maffeis, S.: Horus: a flexible group communication system. Commun. ACM **39**(4), 76–83 (1996). DOI http://doi.acm.org/ 10.1145/227210.227229

[28] Satyanarayanan, M.: Pervasive computing: Vision and challenges. IEEE Personal Communications **8**, 10–17 (2001)

[29] Slominski, A., Govindaraju, M., Gannon, D., Bramley, R.: Design of an XML based Interoperable RMI System : SoapRMI C++/Java 1.1. In: Proceedings of PDPTA, pp. 1661–1667 (June 25-28, 2001)

[30] Tsounis, A., Anagnostopoulos, C., Hadjiefthymiades, S.: The role of semantic web and ontologies in pervasive computing environments. In: Proceedings of the Mobile and Ubiquitous Information Access Workshop, Mobile HCI '04. Glasgow, UK (2004)

[31] Van Engelen, R.A., Gallivan, K.A.: The gsoap toolkit for web services and peer-to-peer computing networks. In: CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid, p. 128. IEEE Computer Society, Washington, DC, USA (2002)

[32] Weiser, M.: Some computer science issues in ubiquitous computing. Commun. ACM **36**(7), 75–84 (1993). DOI http://doi.acm.org/10.1145/159544.159617

[33] Zhu, F., Mutka, M.W., Ni, L.M.: Service discovery in pervasive computing environments. IEEE Pervasive Computing **4**(4), 81–90 (2005). DOI http://dx.doi.org/10.1109/MPRV.2005.87