INDISS: Interoperable Discovery System for Networked Services

Yérom-David Bromberg and Valérie Issarny

INRIA-Rocquencourt,
Domaine de Voluceau, 78153 Le Chesnay, France
{David.Bromberg, Valerie.Issarny}@inria.fr

Abstract. The emergence of handheld devices associated with wireless technologies has introduced new challenges for middleware. First, mobility is becoming a key characteristic; mobile devices may move around different areas and have to interact with different types of networks and services, and may be exposed to new communication paradigms. Second, the increasing number and diversity of devices, as in particular witnessed in the home environment, lead to the advertisement of supported services according to different service discovery protocols as they come from various manufacturers. Thus, if networked services are advertised with protocols different than those supported by client devices, the latter are unable to discover their environment and are consequently isolated. This paper presents a system based on event-based parsing techniques to provide full service discovery interoperability to any existing middleware. Our system is transparent to applications, which are not aware of the existence of our interoperable system that adapts itself to both its environment across time and its host to offer interoperability anytime anywhere. A prototype implementation of our system is further presented, enabling us to demonstrate that our approach is both lightweight in terms of resource usage and efficient in terms of response time.

1 Introduction

The home environment now embeds networked devices, possibly wireless, from various application domains, i.e., home automation, consumer electronics, mobile and personal computing domains. The networked home shall then enable an open spontaneous network in which authorized devices are discovered and connected, as in particular investigated in the Amigo IST project [1].

Service discovery protocols enable finding and using networked services without any previous knowledge of their specific location. Several Service Discovery Protocols (SDP), like Jini [2], SLP [3], UPnP [4] and Salutation [5], are now available. With the advent of both mobility and wireless networking, SDPs are taking on a major role in networked environments, and are the source of a major heterogeneity issue across middleware. Furthermore, once services are discovered, applications need to use the same interaction protocol to allow unanticipated connections and interactions with them. Consequently, a second heterogeneity issue appears among middleware. Summarizing, middleware for the networked home environment must overcome two heterogeneity issues to provide interoperability, i.e.: (i) heterogeneity of service discovery protocols, and (ii) heterogeneity of interaction protocols between services. Interoperability is also difficult between devices made by different manufacturers, as they can implement differently a standardized protocol.

Distributed systems for the networked home must provide efficient mechanisms to detect and interpret protocols used by the networked devices, which are not known in advance. Furthermore, detection and interpretation must be achieved without increasing consumption of resources that are limited on a number of devices (e.g., handheld). New techniques must be used to both: (i) offer lightweight systems so that they can be supported by resource-constrained devices, and (ii) support system adaptation according to the dynamics of the open networked environment. Middleware solutions, designed to cope with the above issues, have been introduced, as surveyed in [6]. From this pool of existing middleware, more or less adapted to the constraints of the networked home, reflective middleware seem to be flexible enough to provide interoperability among networked services. However, solutions to interoperability based on reflective techniques, like ReMMoC [7,8], do not bring simultaneously interoperability and high performance, as discussed in [9]. SDP interoperability needs to be revisited to improve efficiency of SDP detection, interpretation and evolution. Moreover, to provide interoperability, we need a fine-grained control over protocols. Our approach is to decouple components from protocols with the use of concepts inherited from software architecture enhanced with event-based parsing techniques [10,11].

The originality of our approach comes from the trade offs achieved among efficiency, interoperability and flexibility. Our interoperability system, called INDISS (INteroperable DIscovery System for networked Services), may further be integrated with any existing middleware platform. Hosting INDISS enables the networked home system to discover and interpret all the services available in the home environment, independent of underlying middleware technologies. One key feature of INDISS is to provide efficient interoperability without altering the existing applications and services.

Based on conceptual similarities among SDPs, we are able to provide a generic mechanism supporting discovery protocol interoperability, as presented in Section 2. According to user activities, the networked home can become a highly dynamic network formed by the random arrival of devices based on different middleware. Whatever the networked home configuration/composition, interoperability must be maintained transparently without requiring to change the applications and/or services. In this context, INDISS must adapt itself to the evolution of the home environment across time. Section 3 discusses both the self-adaptation and context-awareness capabilities of INDISS. To validate the INDISS design, in particular in terms of efficiency, we have developed a first prototype, which is flexible enough to consider several use cases. Section 4 provides performance results, which demonstrate the efficiency of INDISS. Finally, Section 5 summarizes our contribution and discusses our future work on achieving middleware interoperability.

2 Service Discovery Protocol Interoperability

According to the architectural style of service-oriented computing systems, a majority of SDPs support the concepts of *client* and *service*. 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 services information from service advertisements. The latter is used when either the repository's location is not known or there does not exist any repository in the environment. Similarly, services may announce themselves with either unicast or multicast advertisement, depending on whether a repository is present or not. From the aforementioned approaches, two SDP models are identified, irrespectively of the repository's existence: (i) the passive discovery model, and (ii) the active discovery model. When a repository exists in the network 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 [3]. In this context, 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 multicast advertisements from a repository. On the contrary, with an active discovery model, clients and services send multicast requests to discover a repository that 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, which is specific to the SDP that is used to discover services. Obviously, the latter periodically send out 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 send unicast responses directly to the requester only if they match the requested service. Summarizing, most SDPs support both passive and active discovery with either optional or mandatory centralization points. The following details our solution to SDPs interoperability, which is compatible with both the passive and active discovery models.

The following sections introduce the architectural principles of INDISS that builds on [9] and decomposes into mechanisms for: (i) SDP detection (§2.1) and (ii) SDP interoperability (§2.2). Specifically, SDP interoperability is achieved through translation of SDP functions in terms of events coordination (§2.3). This translation process is then outlined through a concrete example (§2.4).

2.1 SDP Detection

All SDPs use a multicast group address and a UDP/TCP port that must 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. Typically, SDPs are detected through the use of their assigned address and port. These two properties form a unique pair and may be interpreted as a permanent SDP identification tag. Furthermore, it is important to note that an entity may subscribe to several multicast groups simultaneously. These only two characteristics are sufficient to provide simple but efficient environmental SDP detection. We discover passively the environment by listening to the well-known SDP multicast groups. In fact, we learn the SDPs that are currently used from both services' multicast announcements and clients' multicast service requests. To achieve this feature, a component, called *monitor component*, embeds two major behaviours:

– The ability to subscribe to several SDP multicast groups, and
– The ability to listen to all their respective ports.

Figure 1 depicts the mechanism used to detect active and passive SDPs in a repository-less context. The *monitor component,* which may be deployed on the client side and/or service side, joins both the SDP1 and SDP2 multicast groups and listens to the corresponding registered UDP/TCP ports. We assume that SDP1 is based on an active discovery model. Hence, SDP1 clients perform multicast requests to the SDP1 multicast group to discover services in their vicinity. The *monitor component*, as a member of the SDP1 multicast group, receives client requests and thus is able to detect the existence of SDP1 in the environment as data arrival on the SDP1-dedicated UDP/TCP port identifies the discovery protocol. Assuming SDP2 is based on a passive discovery model, SDP2 services advertise themselves to the SDP2 multicast group to announce their existence to their vicinity. Similarly to SDP1, as soon as data arrive at the SDP2-dedicated UDP/TCP port, the *monitor component* detects the SDP2 protocol. 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 existence at the specified UDP/TCP ports inside the corresponding groups.

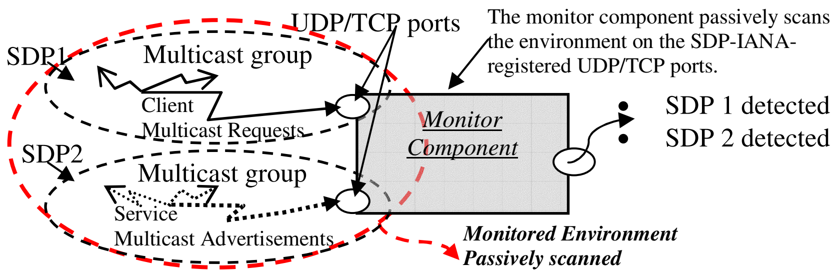


Fig. 1. Detection of active and passive SDPs through the monitor component

The monitor component is easy to implement, as both subscription and listening are solely IP features. Hence, any middleware based on IP support the *monitor component*, which simply maintains a static correspondence table between the IANA-registered permanent ports and their associated SDP. Hence, SDP detection only depends on which port raw data arrived. Therefore, the cost of SDP detection is reduced to a minimum.

2.2 SDP Interoperability

SDP detection is just a first step towards SDP interoperability. The main issue is still unresolved: the incoming raw data flow, which comes to the *monitor component*, needs to be correctly interpreted to deliver the service descriptions to the application components. To effectively support SDP interoperability, we reuse event-based parsing concepts.

Upon the arrival of raw data at monitored ports, the *monitor component* detects the SDP that is used (Figure 2, Step❶), and forwards the input data to the appropriate *parser* (Step❷), 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 *composers* that are locally deployed (Step❸). Finally, the composer delivers a SDP message understood by the target application (Step❹). The communication between the *parser* and the *composer* does not depend on any syntactic detail of any protocol. They communicate at a semantic level through the use of events. Indeed, a fixed set of common events has been identified for all SDPs (see §2.3). And, a larger, specific set of events is defined for each SDP. For example, a subset of events generated by a UPnP parser are successfully understood by a SLP *composer,* whereas specific UPnP events, due to UPnP functionalities that SLP does not provide, are simply discarded from the SLP *composer,* as they are unknown.

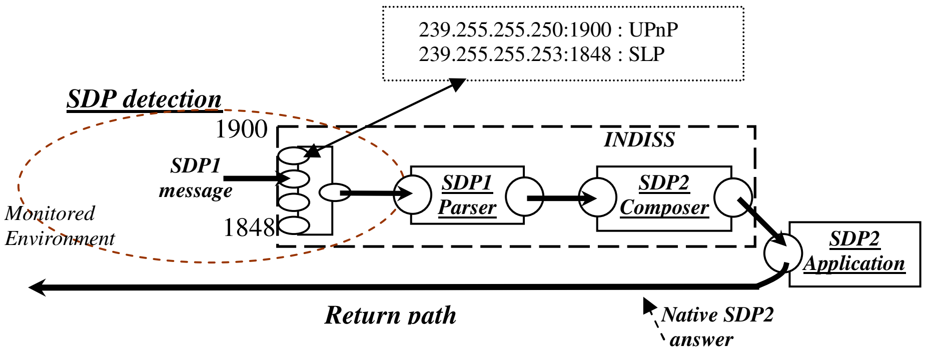


Fig. 2. SDP detection & interoperability mechanisms

Event streams are totally hidden to components outside INDISS, as they are assembled into SDP-specific messages through *composers*. Consequently, interoperability is guaranteed to existing applications tied to a specific SDP without requiring any change to applications. Similarly, future applications do not need to be developed with a specific middleware API to benefit from SDP interoperability. In general, application components continue to use their own native service discovery protocol; interoperability is achieved through a transparent integration of INDISS. It is further important to note that the system may be deployed on either the service provider or client application side. It may even be distributed among both parties or deployed on some intermediate (e.g., gateway) networked node (see §4.2).

Parsers and composers are dedicated to specific SDP protocols. Then, to support more than one SDP, several parsers and composers must be embedded into the system. Embedded parsers and composers are dynamically instantiated.

SDP interoperability comes from the composition of parsers and composers dedicated to different SDPs. As depicted in Figure 3, an incoming SDP1 message is successfully translated into an SDP2 message that is then forwarded to an SDP2-related application. According to several SDP specifications, an incoming message is often followed by a reply message. In this context, two cases may be considered: (i) the reply is directly sent by the native SDP (Figure 2, Step ❺), which requires the receiver to translate the message into a message of the hosted SDP, (ii) the reply is translated into a message of the destination's SDP (Figure 3). The former solution leads to the sharing of the interoperability tasks among all participating nodes. However, this requires all the nodes to embed INDISS. As a result, nodes that do not integrate the necessary interoperability mechanisms are likely to be isolated. Therefore, this specific configuration must be considered as a special case but cannot be assumed nor enforced in general. Instead, we consider that a node embedding INDISS is able to take care of the complete interoperability process, i.e., both receiving and sending messages from/to non-native SDPs. Thus, interoperability among nodes is achieved without requiring all the participant nodes to embed INDISS. SDP interoperability is achieved if the proposed interoperability system is embedded in at least one of the following nodes: client, server or even gateway.

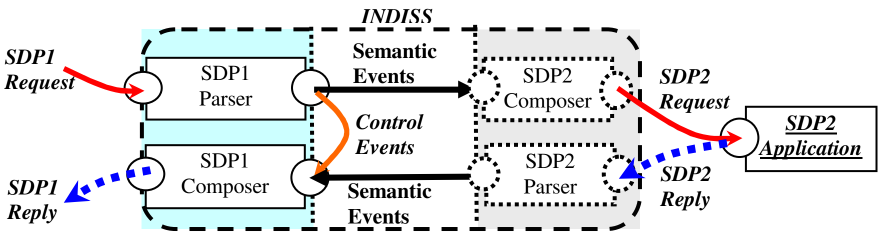


Fig. 3. Coupling of parser and composer

From the above, it follows that within INDISS, a parser is coupled with a composer that does the reverse translation process, in a way similar to the marshalling/unmarshalling functions of middleware stubs. Furthermore, depending on the SDP specification, the parser and composer may have to share one bi-directional session. Such a coupling occurs when, e.g., once the parser has received a request message, the composer has to send some acknowledgement or control message to simply maintain or validate a communication session with the requester. In general, SDP functions like service request, service registration or service advertisements, are complex distributed processes that require coordination between the actors of the specific service discovery function. It follows that the translation of SDP functions that is realised by INDISS is actually achieved in terms of translation of processes and not simply of exchanged messages, further requiring coordination between the parser and composer. This is realized by embedding the parser and composer within a *unit* that runs coordination processes associated with the functions of the supported 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 behaviour of the unit may easily be specified using finite state machines, as detailed in the next section.

2.3 Event-Based Interoperability

A unit implements event-based interoperability for a specific SDP by: (i) translating to and from semantic events associated with service discovery, messages of the specific SDP, and (ii) implementing coordination processes over the events according to the behaviour of the SDP functions.

The overall coordination process implemented by the SDP unit is specified using a Finite State Machine (FSM). A SDP state machine is a graph of states connected by transitions. A SDP state machine is a Deterministic Finite Automaton (DFA) and is, as usual, defined as a 5-tuple (Q, ∑, C,T, q0,F), where Q is a finite set of states, ∑ is the alphabet defining the set of input events (or triggers) the automaton operates on, C is a finite set of conditions, T: Q x ∑ x C → Q is the transition function, q0 ∈ Q is the starting state and F ⊂ Q is a set of accepting states. *States* keep track of the progress of the SDP coordination process. *Transitions* are labelled with events, conditions and actions.

Table 1. Mandatory events

Event set	Event type
SDP Control Events	SDP_C_START SDP_C_STOP SDP_C_PARSER_SWITCH SDP_C_SOCKET_SWITCH
SDP Network Events	SDP_NET_UNICAST SDP_NET_MULTICAST SDP_NET_SOURCE_ADDR SDP_NET_DEST_ADDR SDP_NET_TYPE
Service Events	SDP_SERVICE_REQUEST SDP_SERVICE_RESPONSE SDP_SERVICE_ALIVE SDP_SERVICE_BYEBYE SDP_SERVICE_TYPE SDP_SERVICE_ATTR
SDP Request Events	SDP_REQ_LANG
SDP Response Events	SDP_RES_OK SDP_RES_ERR SDP_RES_TTL, SDP_RES_SERV_URL

The occurrence of an event may cause a transition if the event matches both the event and the condition of the transition. When a transition is engaged, several actions may be executed, relating to translation of events to/from message data, coordination, and configuration management (see Section 3). A SDP DFA is dedicated to one protocol to account for the protocol's specifics and consequently realize some optimisation. Events are basic elements and consist of two parts: *event type* and *data*. Whatever their types, events are always considered as triggers for the unit components to react and eventually activate some coordination rule. We define the minimal/mandatory set of events that is common to all SDPs and sets of specialized events that are specific to SDPs. The set of mandatory events ∑ is defined as the union of a number of subsets (see Table 1):

$\sum_m$= *"SDP Control Events"* $\cup$ *"SDP Network Events"* $\cup$ *"SDP Service Events"* $\cup$ *"SDP Request Events"* $\cup$ *"SDP Response Events".*

The set "*SDP Control Events"* contains events that may be generated by components embedded in INDISS (See section 3) to notify their listeners of their internal states. For instance, it enables either the *unit* to control the coordination of its registered components (i.e., parsers, composers) or any other components, registered as a listener, eventually from an upper layer like the application layer, to trace, in real time, SDP internal mechanisms. This is a useful feature, not only for debugging purposes, but also for a dynamic representation of the run-time interoperability architecture. The set "*SDP Network Events"* is related to network properties and, for instance, defines events to determine if the SDP messages are either unicast or multicast, to indicate the SDP used and to specify the source or target address. Then, "*SDP Service Events"* enriches the above set with necessary events to describe the functions that are common to the different SDPs: service search request, service search response, service advertisements and the type of the service searched. Then, "*SDP Request Events"* and "*SDP Response Events"* contain events respectively dedicated to the description of SDP requests with richer descriptions, and to specific events to express possible common SDP answers (e.g., positive or negative acknowledgement, URL of the searched service etc).

All SDP parsers must at least generate the mandatory events. Conversely, all SDP composers must also understand them. The mandatory events result from the greatest common denominator of the different SDP functionalities. Nevertheless, a given SDP parser may generate additional events related to its advanced functionalities. Similarly, a SDP composer may manage these additional events. However, SDP composers are free to handle or ignore them. For instance, SLP does not manage UPnP advanced functionalities. Consequently, the SLP composer ignores UPnP-specific events generated by the UPnP parser. On the other hand, a JINI-related composer may support some of the UPnP-specific events. In fact, events added to the mandatory ones enable the richest SDPs to interact using their advanced features without being misunderstood by the poorest. The behaviour of the latter is unchanged as they discard unknown events and consider only the mandatory events. Moreover, INDISS is extensible and integration of future SDPs is rather direct. In particular, the possible introduction of new events to increase the quality of the translation process will not trigger a whole cascade of changes of SDP components. This is a direct consequence of building INDISS upon the event-based architectural style. We introduce three open, *extension sets* for the definition of additional events: "*Registration Events"*, "*Discovery Events"* and "*Advertisement Events".* For instance, specific SDP messages involved in the registration of services are translated to events belonging to the *"Registering Events"* set, which enriches both "*SDP Requests Events"* and "*SDP Responses Events".* The same applies for the *"Discovery Events"* set. On the other hand, *"Advertisement Events"* enriches only "*SDP Responses Events*" since an advertisement is a one-way message to spread service location.

States of the DFA (or coordination process) of a unit are activated according to triggers that define the event types that can cause transitions between states. Transitions imply that the unit executes some actions or coordination rules among its components (i.e., composer, parser). According to the unit's current state, incoming events are filtered and may be dispatched to different listeners (i.e., composer, parser or other units) until new incoming triggers cause a transition to a new state and so on. Reply messages generated through the composer may rely on data associated with events generated previously by its associated parser. Thus, events data from previous states are recorded using state variables. Conditions are written as Boolean expressions over incoming and/or recorded data and may test their properties, whereas actions are a sequence of operations that a unit can perform to: dispatch events to components, record events, or reconfigure the composition of its embedded components (e.g., changing dynamically the current parser or composer). Actions that may be performed by a unit are specific to the SDP that it manages. However, all units have to support mandatory actions.

2.4 Example

We illustrate our solution using a scenario where a SLP client is searching, e.q., a clock service. The clock service is based on UPnP and interoperability is enabled through the transparent use of INDISS (See Figure 3 with SDP1=SLP and SDP2=UPnP). Our aim, in this scenario, is to outline the different steps involved in the interoperability process and more particularly, to describe how messages are successfully transformed to events and vice-versa, during a search session initiated by a SLP client, to discover a service based on UPnP. However, for brevity, we describe only the most meaningful events that occur during this scenario.

First, the client broadcasts a SLP search request to discover its environment in order to find a clock service. As presented in Sections 2.1 and 2.2, INDISS catches the request as a raw data stream and forwards it to the parser of the SLP unit that generates a stream of events, which is dispatched to the composer of the UPnP unit as depicted in Figure 4, step❶. The event stream always starts with a *SDP_C_START* event and ends with a *SDP_C_STOP* event to specify the events belonging to a same message. On the other hand, the *SDP_NET_MULTICAST*, *SDP_SERVICE_REQUEST*, *SDP_SERVICE_TYPE* events are used to generate a corresponding UPnP search request. *SDP_REQ_VERSION*, *SDP_REQ_SCOPE*, *SDP_REQ_PREDICATE* and *SDP_REQ_ID* are events specific to SLP and are thus discarded by the UPnP unit's composer. The *SDP_NET_SOURCE_ADDR* is directly forwarded to the SLP composer embedded into the SLP unit to prepare the reply. The routing of events and related actions are specified by the DFA of the units as presented in §2.3.

Once the UPnP service has received the UPnP search request from INDISS, it responds to it with a corresponding UPNP search answer (Figure 4, step ❷), which is then parsed by the UPnP unit. An event stream is generated and dispatched to the SLP unit's composer. However, thanks to its DFA, the UPnP unit detects that it does not get enough events from the UPnP service. The *SDP_RES_SERV_URL* event, which indicates the URL of the searched service, has never been generated. Therefore, the UPnP unit needs to recursively generate additional requests to the remote service until it receives the expected event. To achieve this task, the UPnP-specific events generated by the UPnP unit are consumed internally by the composer to generate other UPnP requests. For instance, the *SDP_DEVICE_URL_DESC* event gives the URL of the description of the remote service that contains the URL of the remote service endpoint. Therefore, once the composer of the UPnP unit receives this event, it generates a corresponding request to get the description. As previously, the next answer from the service is parsed (Figure 4, step ❸) but the reply contains a XML body that the current UPnP parser, which is dedicated to the SSDP protocol, does not understand. Therefore, the current parser generates a *SDP_C_PARSER_SWITCH* event to ask its unit to switch to a XML parser to continue the parsing to get finally the expected *SDP_RES_SERV_URL* event. The XML description is converted to several *SDP_RES_ ATTR* events. As soon as the composer of the SLP unit has received all of them (as indicated by *SDP_C_STOP*), a SLP answer is generated (the *SDP_RES_ ATTR* are translated to traditional SLP attributes) and received by the SLP client.

Step	Request	Generated Events	Composed request
❶	SLP Search	SDP_C_START …. **SDP_NET_MULTICAST** SDP_NET_SOURCE_ADDR **SDP_SERVICE_REQUEST** SDP_REQ_VERSION SDP_REQ_SCOPE SDP_REQ_PREDICATE SDP_REQ_ID **SDP_ SERVICE_TYPE:** SDP_C_STOP	*From the previous events, the UPnP unit multicasts a UPnP search request to discover UPnP services in its vicinity:* M-SEARCH * HTTP/1.1 SERVER: 239.255.255.250:1900 ST: urn:schemas-upnp org:device:clock MAN: ssdp:discover MX: 0

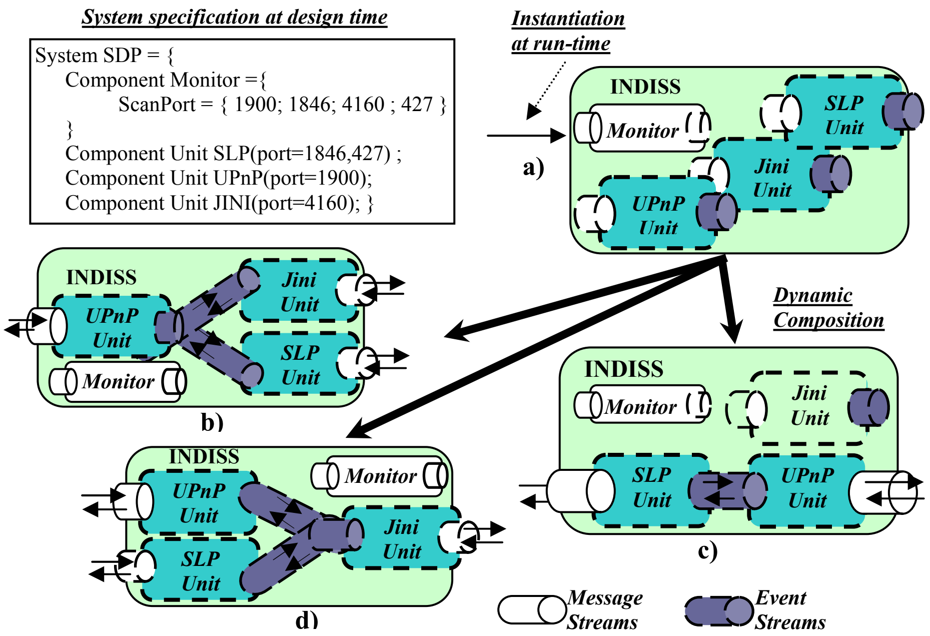
Step	Reply Parsing	Generated Events	Composed request
❷	**HTTP/1.1 200 OK** **CONTENT-TYPE:** text/html; **SERVER**: UPnP/1.0 CyberLink/1.3.2 **CONTENT-LENGTH**: 0 ……………… **ST:** upnp:clock **USN:** uuid: ClockDevice::upnp:clock **LOCATION:** http://128.93.8.112:4004/description.xml	SDP_C_START SDP_NET_TYPE SDP_SERVICE_TYPE **SDP_DEVICE_URL_DESC** ……	*As the UPnP unit did not get the location of the remote service it must generate additional UPnP requests:* GET /description.xml HTTP/1.1

Step	Reply	Generated Events	Composed reply
❸	*Service answer to the GET request:* **HTPP Reply**	*Events generated from the HTPP reply:* SDP_C_PARSER_SWITCH SDP_RES_ATTR SDP_RES_ATTR ….. **SDP_RES_SERV_URL** SDP_C_STOP	**SrvRply**: **sevice:clock:soap://128.93.8.112:4005/ service/timer/control** ;**major**:"1";**minor**:"0";**friendlyName**:"CyberGarage Clock Device"; **modelDescription**:"CyberGarage"; **manufacturerURL**:"http://www.cybergarage.org"; **modelDescription**:"CyberUPnP Clock Device"; **modeName**:"Clock";**modelNumber**:"1.0"; **modelURL**:"http://www.cybergarage.org";

Fig. 4. SLP-UPnP interoperability in action

3 Context-Aware, Self-adaptive Interoperability

INDISS is based on a specialization of the event-based architectural style. Advantages of using an event-based architecture are: increasing the degree of decoupling among components and of interoperability, and providing a dynamic and extensible architecture. Since interactions among components are based on events, components operate without being aware of the existence of other components and consequently parsers, composers and units may change dynamically at runtime without altering the system (see Figure 5). INDISS is consequently defined as a set of event-based components. We distinguish between these components that are inside the system, and other components that are outside INDISS and are therefore considered as application components.



```
System SDP = {
    Component Monitor ={
          ScanPort = { 1900; 1846; 4160 ; 427 }
    }
    Component Unit SLP(port=1846,427) ;
    Component Unit UPnP(port=1900);
    Component Unit JINI(port=4160); }
```

Fig. 5. Evolution of INDISS configuration

The INDISS internal architecture has to evolve across time due to two main reasons. First, as devices joining the network, whether mobile or stationary, evolve over time, the current SDP that is used and/or the SDPs with which interoperability is required may change accordingly. Second, some SDPs are actually based on a combination of protocols. For instance, UPnP uses alternatively SSDP, HTTP, and SOAP. To support these two types of changes, we need to define rigorous composition rules to describe the specific architecture of a given instance of INDISS. Configuration of a INDISS instance is initially defined in terms of supported SDPs and the corresponding units that need be instantiated. As illustrated in Figure 5.a, specification of the system configuration does not describe when and how to compose units. Indeed, unit composition is achieved dynamically according to both the context and the hosted application components. The context is discovered with the help of the monitor component, as presented in Section 2.1. At run-time, embedded units of different types are instantiated and dynamically composed depending on the environment and the applications used. Thus, several configurations may occur (e.g., see Figure 5.b, c, d).

At the system level, SDP interoperability is achieved through the correct composition of some units. As depicted in Figure 5.c, the translation from SLP to UPnP discovery corresponds to the composition of a SLP unit with a UPnP unit. At this level, a unit is only considered as a computational element that transforms messages to events and vice versa. The unit's internal mechanisms are totally hidden. Referring to event-based architectures, components can be either event listeners or event generators or both. The same applies for units; they are both event generator and listener. Units are composed and communicate together through events, whereas they use messages to interact with components that are outside INDISS. Therefore, the use of events is internal to INDISS.

At the unit level, coordination and composition rules among embedded SDP components are specialized with respect to a given SDP, according to the unit's state-machine. The unit is then in charge of dispatching event notifications to its registered listeners. However, there are some variations applied to the traditional event-based style. First, the unit does not systematically forward incoming events to all subscribers. The unit filters events, and may additionally react to them through actions to modify its current configuration. Events delivery and executed actions are dependent upon the unit's state machine described earlier. A notable feature of our solution is that SDP interoperability components that are developed are not necessarily specific to a SDP. Customization of a unit with respect to a SDP results from the specific configuration and in particular the embedded FSM.

As a result, interoperability components may be reused in various units, even if not related to the same SDP. For instance, at the implementation level, HTTP or XML parsers developed for one SDP may be reused for another. Definition of a unit then relies upon specifying embedded components, as exemplified below for a UPnP unit:

```
Component Unit UPnP = {
        setFSM(fsm, UPNP);
        AddParser(component, SSDP);
        AddComposer(component, SSDP);
        ...}
```

The state machine's description is itself considered as a part of the system specification. Hence, a new operator is introduced to define state machines:

```
Component UPnP-FSM ={
        AddTuple(CurrentState,triggers,condition-guards,NewState,actions)
                                ...}
```

In the above tuple, *CurrentState* and *NewState* are labels to name different states, *triggers* are taken from the set of previously defined events, *condition-guards* are Boolean expression on events and *actions* are those provided by the unit's interface.

4 Prototype Implementation and Performance

We have implemented a first prototype of INDISS. Currently, it includes a UPnP unit and a SLP unit. Although our prototype is not yet optimised, it is robust enough to assess the performance of our approach in different use cases. The following discusses key elements of the prototype. We first outline its small size requirements compared to existing solutions (§4.1). We then discuss how it improves interoperability within the networked home according to the nodes on which it is deployed and the usage context (§4.2). Finally, we evaluate INDISS performance by comparing response times with native service discovery (§4.3).

4.1 Prototype Implementation

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 J2ME [12], which provides a Java virtual machine customized for devices with limited resources. However, INDISS is not constrained to be written in Java, and may be developed as well in C or in any other programming language closer to the embedded operating system, to get a smaller code-size foot print and better execution speeds. Nevertheless, in Java, we get already very encouraging results. We compare the size required by INDISS with common open-source library like *OpenSlp*[1] and *Cyberlink* for Java[2].

As depicted in Table 2, currently, the overall INDISS system consists of 39 Java classes, and 2910 lines of Non Commented Source Statements Classes (NCSS). The overall system size is 218 Kbytes. This includes 125Kbytes for the UPnP Unit and 49Kbytes for the SLP one. To be interoperable, nodes running UPnP (resp. SLP) applications need to host native UPnP (resp. SLP) library plus INDISS. This is to be contrasted with a 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.

Still in Table 2, without INDISS, the size requirements of a middleware that needs to be interoperable for hosting one simple service is 514Kbytes. Conversely, the size requirement for a middleware dedicated to UPnP (resp. SLP) equipped with INDISS is 598Kbytes (resp. 352Kbytes). Moreover, the size requirements increase proportionally with the number of hosted services. Therefore, according to the number of hosted services, the size requirements of an interoperable middleware without INDISS increases faster than the one equipped with INDISS simply because, for the former, each time we add a service we are multiplying its size by two (e.g., SLP service size + UPnP service size).

Thus, the small size overhead introduced by INDISS with UPnP applications disappears with the number of hosted services. Last but not least, a middleware that needs to host different services, in terms of both functionalities and SDP used,

[1] http://www.openslp.org/

[2] http://www.cybergarage.org/net/upnp/java/

Table 2. Size requirements in KBytes for known libraries and INDISS

INDISS size requirements				
	Size (KB)	**Classes**	**NCSS**	**Overhead**
Core framework	44	15	789	-
UPnP Unit	125	18	1515	-
SLP Unit	49	6	606	-
Total	**218**	**39**	**2910**	-
SDP library size requirements				
OpenSlp Library	126	21	1361	-
Cyberlink UPnP	372	107	5887	-
Total	**498**	**128**	**7248**	-
Size requirements to provide interoperability with and without INDISS				
SLP &UPNP Library + SLP & UPnP clients	514	-	-	-
UPnP client & Library + INDISS	598	-	-	**14%**
SLP client & Library + INDISS	352	-	-	**-31.5%**

must have all the corresponding native libraries irrespectively of the use of INDISS. How ever, 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 you do not have to develop and deploy services for each existing SDP.

4.2 Interoperability Scenarios

One of our objectives is to provide service discovery interoperability to applications without altering them. Hence, applications are not aware of interoperability mechanisms and actually have the illusion that the remote applications that they discover (and/or discover them) use the same SDP. In this context, several use cases may be considered, according to both the nature of the SDPs that are used and the location of INDISS, which can be localized on the client, server, both or gateway.

Another of our other objectives is to save resources on resource-constrained devices and the bandwidth that is shared among devices in the network. It is thus important to examine the impact of INDISS on resource consumption. This may in particular vary according to the system's location (i.e., where it is deployed) and usage context. The usage context of the system depends on the SDP model used by the clients and services. Referring to Section 2, there exist two SDP models: passive and active. We need thus to distinguish cases where the client (resp. service provider) acts as listener and as a requester. Moreover, we obviously assume that either the client or service node hosts INDISS. As a result, for each possible scenario, two uses cases are possible, according to the location of INDISS.

Consider first that both clients and services are based on the passive discovery model (see Figure 6). In this context, clients are listeners and services are requesters. The most optimised location for INDISS is to be hosted on the client side. Thereby, clients are able to intercept all messages generated by the remote service whatever its

specific multicast group or message format (see left-top of Figure 6). In contrast, if as, INDISS is localized on the service side, it will never intercept messages from clients
INDISS is localized on the service side, it will never intercept messages from clients
by definition of the passive discovery model, clients are listeners and never generate messages. We get a blocked situation as depicted in the right top of Figure 6.

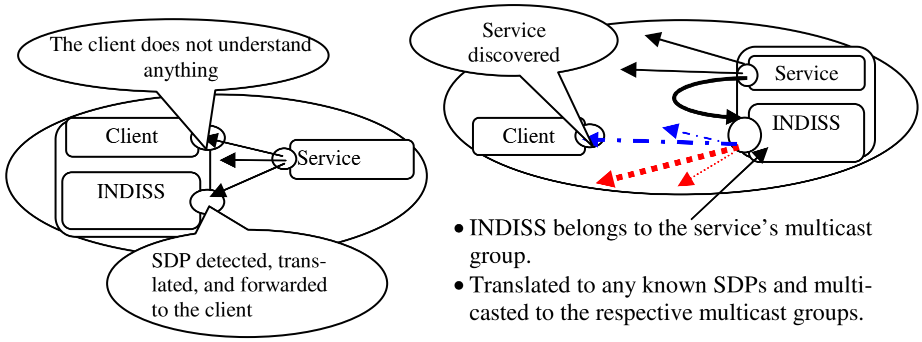

The client does not understand anything
Client
INDISS
Service
SDP detected, translated, and forwarded to the client
Service discovered
Client
Service
INDISS
• INDISS belongs to the service's multicast group.
• Translated to any known SDPs and multicasted to the respective multicast groups.

Fig. 6. SDP interoperability and passive service discovery

Consequently, we must define a *network traffic threshold* below which INDISS, hosted on the service host, must become active so as to intercept messages generated from the local services in order to translate them to any known SDPs according to the embedded units (see bottom of Figure 6). Although this specific use case illustrates the high flexibility of INDISS to adapt itself to the context, it has non-negligible impact on resource consumption. Indeed, dynamic reconfiguration of the system has a processing cost and service advertisements following the enactment of the active model increases bandwidth usage. However, interoperability is enforced without really saturating the bandwidth, as INDISS is switched to the active model only when the network traffic is low.

Consider now the case where both clients and services are based on the active discovery model, i.e., clients are requesters and services are listeners. In order to optimise the bandwidth usage and computational resources, the most suitable location for INDISS is to be on the service side. Otherwise, in a way similar to the previous scenario, ineffective SDP interoperability may arise when INDISS is located on the requester side. In general, when the clients and services are based on the same discovery model, the most convenient location for INDISS is on the listener side.

It may be the case that the clients and services are based on different discovery models. If the clients are based on the active model and services are based on the passive model, then both clients and services generate SDP messages. Interoperability is guaranteed without additional resources cost. Nevertheless, some subtleties arise. Hosting INDISS on the client side means that the client benefits from the advertisements of remote services. But, the client's requests will not reach remote services that are based on different SDPs if they are not interoperable (i.e., they do not host our interoperability system). On the contrary, if services embed INDISS and not the clients, requests from the latter will be taken into consideration from services, whereas clients will not be aware of services' advertisements originating from SDPs different than the one hosted on the clients. Although, in this case, interoperability is not as effective as expected, clients and services do interact. Furthermore, interoperability effectiveness may be improved if the bandwidth is under-utilized, thanks to INDISS reconfigurability

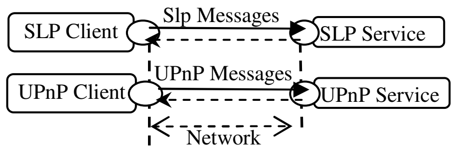
Conversely, when clients are based on the passive model and services are based on the active model, both clients and services are listeners. Once again, we are faced with the recurrent ineffective discovery interoperability. However, in this particular case, dynamic reconfiguration of INDISS does not resolve the clients' inability to discover services, since there is no node initiating SDP-related communication. There is no way to resolve this issue, considering our constraint to not alter the behaviour of SDPs, clients and services. On the other hand, this specific case is unlikely to happen. Nowadays, in practice, clients are always able to generate requests.

Summarizing, irrespective of the service discovery model used by clients and services, we are able to guarantee a minimum level of interoperability. Depending on the environment, the bandwidth usage may be increased to enable interoperability. The basic idea is to provide a quasi-full interoperability as long as the bandwidth-usage enables it. Then, interoperability degradation may occur according to the traffic. Furthermore, by design, INDISS is independent of its host. Thus, it is not mandatory for INDISS to be deployed on the client or service host. INDISS may be deployed on a dedicated networked node, depending on the specific network environment. Such a dedicated node may in particular translate messages generated in one environment from any SDP to messages handled by any other SDP, according to the traffic condition. Obviously, this specific configuration generates additional traffic and is only valid as long as there is enough bandwidth.

4.3 Experimental Results

We evaluate the performance of our interoperability mechanisms by investigating the response time of INDISS 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 answer from an INDISS-translated service. The impact of INDISS on performance varies according to its location, on either the client or 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 resources 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 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 UPnP client and service use Cyberlink for Java. The given measurements are in ms and are the median of 30 successful tests to avoid a mean skewed by a single high or low value.



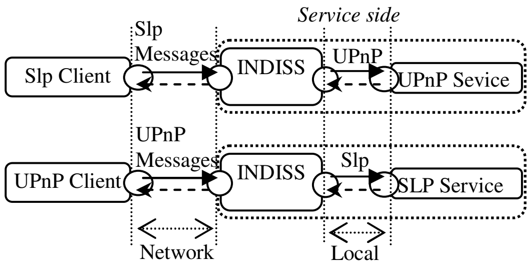
	SLP -> SLP	UPnP -> UPnP
Median value (ms)	0.7	40

Fig. 7. Native clients & services

In Figure 7, 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.

Consider now the case where INDISS is located on the service side to enable the latter to be interoperable with any client independently of its SDP (Figure 8). In the context where the client is SLP and the service is UPnP, the client gets an answer in 65 ms. 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, INDISS 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 INDISS has waited and parsed successively two UPnP responses increasing thus the SLP responsiveness latency. On the service side, it is clear that INDISS simulates a UPnP client and therefore we cannot interfere on the native time taken to get UPnP response from the service. In this context, the INDISS result is pretty good.

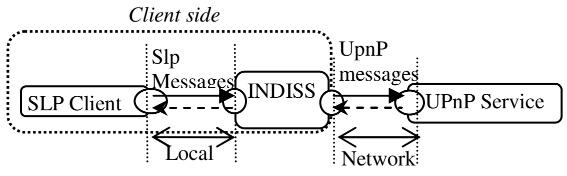
Still in Figure 8, when the client is UPnP and the service is SLP, the response time to get an answer is 40ms. In fact, it corresponds exactly to a search request generated



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

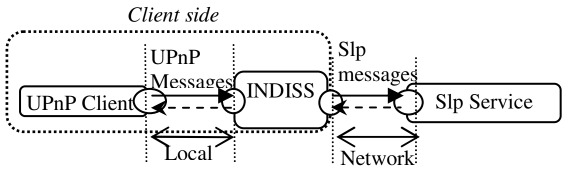
	UPnP->[UPnP-Slp]
Median value (ms)	**40**

Fig. 8. Performance with INDISS located on the service side



	[Slp-UPnP]->UPnP
Median value (ms)	80

a) SLP search request to a UPnP service



	[UPnP-Slp]->Slp
Median value (ms)	0.12

b) UPnP search request to a SLP service

Fig. 9. Performance with INDISS located on the client side

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.

When INDISS is located on the client side (Figure 9a), 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. It is obvious since, as previously, INDISS has translated the SLP request into two network UPnP requests to get the necessary information to generate locally the corresponding SLP response. Once again, INDISS result is encouraging. It is important to note that compared to the case depicted in Figure 8, the response time is higher than previously simply because the UPnP traffic goes across the network between INDISS and the UPnP service, increasing by 15 ms the response time. In the same context, the lack of speed inherent to the UPnP protocol is confirmed as a UPnP client gets a response from a SLP service in only 0.12ms (Figure 9b). 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.

From the above results, we have shown that INDISS is particularly efficient in providing interoperability in all possible context use.

5 Conclusion

INDISS overcomes the heterogeneity of service discovery in the networked home and decomposes into two mechanisms: SDP detection and SDP interoperability, allowing therefore any networked home system to discover and interpret all the services available in the home environment, independent of underlying middleware technologies.

Our solution is specifically designed for highly dynamic home networks, which requires both minimizing resource consumption, and introducing lightweight mechanisms that may be adapted easily to any platform. INDISS is composed of a set of event-based components and their composition/configuration is performed dynamically at run-time according to both the context and the host on which INDISS is deployed. As a result, service discovery interoperability is provided to applications without altering them: applications are not aware of the existence of INDISS, which adapts itself to the context. In particular, INDISS may be deployed on a client, a service or a gateway. As demonstrated by the first INDISS prototype, experiment results are encouraging. The response time of INDISS when enabling a client dedicated to one SDP to discover a service based on another SDP is close to request/response among related native clients/services.

Once services are discovered, applications further need to use the same interaction protocol to allow unanticipated connections and interactions with them. In this context, the ReMMoC reflective middleware introduces a quite efficient solution to interaction protocol interoperability. The plug-in architecture associated with reflection features allows mobile devices to adapt dynamically their interaction protocols (i.e., publish/subscribe, RPC etc.). Furthermore, [13] proposes to use ReMMoC together with WSDL [14] for providing an abstract definition of the remote component's functionalities. Client applications may then be developed against this abstract interface without worrying about service implementation's details. However, the solution discussed in [13] suffers from a major constraint: service and client must agree on a unique WSDL description. But, once again, in a dynamic network, the client does not know the execution context. Therefore, it is not guaranteed to find exactly the expected service. Client applications have to find the most appropriate service instance that matches the abstract requested service. In addition, this leads to the dynamic composition of services. This issue is addressed by the WSAMI middleware developed in the context of the Ozone project [15], which introduces enhanced WSDL specification for mobile services and a dedicated middleware to allow a service instance to be *automatically selected and composed upon a user request, according to the services that may be retrieved in the environment*. However, if WSAMI provides interoperability to Web services in the mobile environment, it is still a SOAP based middleware, and hence does not deal with interoperability among components using heterogeneous interaction protocols. We are currently investigating solutions to this issue to complement our solution to SDP interoperability and thus support middleware interoperability, as required by today's network environments [1].

Acknowledgements

This work has received the support at the European Commission through the IST program, as part of the AMIGO project (http://www.amigo-project.org). The authors would like to thank Daniele Sacchetti for helping us make performance measurements. The authors are further grateful to anonymous reviewers for useful comments.

References

[1] Amigo Consortium. Specification of the Amigo abstract middleware architecture. http://www.hitech-projects.com/euprojects/amigo/.

[2] Sun. Technical White Paper: Jini Architectural Overview. 1999.

[3] C. Bettstetter and C. Renner. A comparison of service discovery protocols and implementation of the service location protocol. In Proceedings of the 6th EUNICE Open European Summer School: Innovative Internet Applications, 2000.

[4] Universal Plug and Play Forum. Universal Plug And Play Device Architecture. 2000.

[5] Salutation Consortium. White paper: Salutation Architecture. 1998.

[6] C. Mascolo, L. Capra, W. Emmerich. Middleware for mobile computing (A survey). In Advanced Lectures in Networking. Editors E. Gregori, G. Anastasi, S. Basagni. Springer. LNCS 2497. 2002.

[7] G. Coulson, G. Blair, M. Clarke and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. In Distributed Computing. April 2002.

[8] P. Grace, G. Blair and S. Samuel. Middleware awareness in mobile computing. In Proceedings of the 1st international ICDCS Workshop on Mobile Computing Middleware, May 2003.

[9] Y.-D. Bromberg, V. Issarny. Service Discovery Protocols Interoperability in the Mobile Environment. In Proceedings of the International Workshop Software Engineering and Middleware (SEM). September 2004.

[10] N. Ryan and A. Wolf. Using event-based parsing to support dynamic protocol evolution. In Proceedings of the 26th International Conference on Software Engineering (ICSE'04).2004

[11] D. Garlan. Formal modeling and analysis of software architecture: Components, connectors, and events. In Third International School on Formal Methods for the Design of Computer, Communication and Software Systems. September 2003.

[12] The Micro Edition of the Java 2 Platform, http://java.sun.com/j2me/

[13] P. Grace, G. Blair and S. Samuel. A marriage of Web services and reflective middleware to solve the problem of mobile client interoperability. In Proceedings of Workshop on Middleware Interoperability of Enterprise Applications. September 2003.

[14] W3C."Web Services Description Language (WSDL)", W3C Working Draft. 2003

[15] V. Issarny, D. Sacchetti, F. Tartanoglu, F. Sailhan, R. Chibout, N. Levy, and A. Taloma. Developing ambient intelligence systems: A solution based on Web services. Journal of Automated Software Engineering, 2005.