# Service Discovery Protocol Interoperability in the Mobile Environment

Yérom-David Bromberg, Valérie Issarny

# Service Discovery Protocol Interoperability in the Mobile Environment[1]

Yérom-David Bromberg, Valérie Issarny

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

**Abstract.** The emergence of portable computers and wireless technologies has introduced new challenges for middleware. Mobility brings new requirements and is becoming a key characteristic. Mobile devices may move around different areas and have to interact with different types of networks, services and may be exposed to new communication paradigms. Thus, mobile distributed systems need to dynamically detect and adapt their interaction protocols to interoperate with services available in the environment. As a result, middleware for mobile devices must overcome two heterogeneity issues to provide interoperability in the mobile environment, i.e, heterogeneity of discovery protocols and of interaction protocols between services. Whereas adaptation techniques from reflective middleware are suitable for the latter, it is more problematic for the former if both issues are addressed concurrently. Specifically, reflective mechanisms consume too many resources like bandwidth, memory and CPU, which are limited on the mobile devices. This paper first highlights why current solutions to interoperability fail to realize service discovery protocol interoperability with both high performance and low resource consumption. Second, this paper addresses this open issue by using software architecture concepts enhanced with event-based parsing techniques to provide efficient, lightweight and flexible mechanisms to bring full service discovery interoperability to any existing mobile platform.

## 1 Introduction

In the mobile computing domain, middleware holds a predominant role. Communication relationships amongst application components involve the use of protocols, making applications tightly coupled to middleware. Additionally, to overcome wireless networks constraints, like limited bandwidth, poor network quality of service and either voluntary or forced frequent disconnection, several communication models have arisen. Thus, as it exists many styles of communication and consequently many styles of middleware, we have to deal with middleware heterogeneity [1]. Significantly, an application implemented upon a specific middleware cannot interoperate with services developed upon another. Similarly, we cannot predict at design time the requirements needed at run-time since the execution environment is not known. How-

[1] *In Proceedings of the International Workshop Software Engineering and Middleware (SEM). September 2004.*

ever, no matter which underlying communication protocols are present, mobile nodes must both discover and interact with the services available in their vicinity. More precisely, service discovery protocols enable mobile nodes to find and use networked services without any previous knowledge of their specific location. Several Service Discovery Protocols (SDP), like Jini [4], SLP [2], UPnP [5] and Salutation [15], are now available. And, with the advent of both mobility and wireless networking, SDPs are taking on a major role, 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 mobile devices must overcome two heterogeneity issues to provide interoperability in the mobile environment, i.e.:

- Heterogeneity of service discovery protocols, and
- Heterogeneity of interaction protocols between services.

In addition, both SDPs and interaction protocols are not protected from evolution across time. Indeed, an application may neither interact correctly nor be compatible with services if they use different versions of the same protocol [12]. Interoperability is also difficult between devices made by different manufacturers as they can implement differently a standardized protocol. Protocol evolution increases communication failure probability between two mobile devices.

As outlined above, interoperability among entities of a spontaneous ad hoc network, which is formed by the random arrival of mobile devices for short periods of time, is becoming a real issue to overcome. A portable computer must be aware of its dynamic environment that evolves over time, and further adapt its communication paradigms according to the environment. Thus, mobile distributed systems must provide efficient mechanisms to detect and interpret protocols currently used, which are not known in advance. Furthermore, detection and interpretation must be achieved without increasing consumption of resources that are limited on the mobile devices. This paper introduces base mechanisms for achieving interoperability among heterogeneous SDPs, which consider the above mobility requirements. We reuse concepts from software architecture enriched with event-based parsing techniques to drastically improve SDP interoperability, enabling mobile applications to be efficiently aware of their environment. The originality of our approach comes from the trade offs achieved among efficiency, interoperability and flexibility. Our solution may further be applied to any existing middleware platform.

In the following, we first examine how reflective middleware manages interoperability among heterogeneous SDPs, highlighting the current drawbacks that need to be addressed to provide efficient SDP interoperability (§2). This leads us to investigate a solution grounded in the software architecture domain to overcome the limitation of reflective middleware (§3). Then, we present the design of our proposal to bring both efficient and flexible SDP interoperability (§4). Finally, we conclude by a summary of our contribution (§5).

## 2 Reflective Middleware to Cope with Middleware Heterogeneity

New techniques must be used to both offer lightweight mobile systems and support their adaptation according to the dynamics of the mobile environment. Classic mid-

dleware are not the most suitable for the mobile domain. Their design is based on fixed network and resources abundance. Moreover, network topologies and bandwidth are fixed over time. Hence, quality of service is predictable. Furthermore, with fixed network in mind, the common communication paradigm is synchronous and connections are permanent. However, many new middleware solutions, designed to cope with mobility aspects, have been introduced, as surveyed in [6]. From this pool of existing middleware, more or less adapted to the constraints of the mobile environment, reflective middleware seem to be flexible enough to fulfill mobility requirements, including providing interoperability among networked services.

A reflective system enables applications to reason and perform changes on their own behavior. Specifically, reflection provides both inspection and adaptation of systems at run-time. The former enables browsing the internal structure of the system, whereas the latter provides means to dynamically alter the system by changing the current state or by adding new features. Thus, the middleware embeds a minimal set of functionalities and is more adaptive to its environment by adding new behaviors when needed. This concept, applied to both service discovery and interaction protocols, allows accommodating mobility constraints. This is illustrated by the ReMMoC middleware [1], which is, at this time, the only one to overcome simultaneously SDPs and interaction protocols heterogeneity. The ReMMoC platform is composed of two component frameworks [1, 16]: (i) the *binding framework* that is dedicated to the management of different interaction paradigms, and (ii) the *service discovery framework* that is specialized in the discovery of the SDPs currently used in the local environment. The *binding framework* integrates as many components as interaction protocols supported by the platform. The *binding framework* can dynamically plug on demand, one at time or simultaneously, different components corresponding to the different interaction paradigms (e.g., publish/subscribe, RPC...). Correspondingly, the *service discovery framework* is composed of as many components as of SDPs recognized. For example, SLP and UPnP can be either plugged together or separately, depending of the context. Obviously, such plug in of components applies only to components that are specifically developed for the ReMMoC platform. It is further important to note that the client application is specific to the ReMMoC API but is independent from any protocol, the interested reader being referred to [13] for further details on the mapping of an API call to the current binding framework.

Although ReMMoC enables mobile devices to use simultaneously different SDPs and interaction protocols, this still requires the environment to be monitored to allow ReMMoC to detect over time the SDPs and interaction protocols that need be supported/integrated, due to the very dynamic nature of the mobile environment. Such a knowledge about the environment may be made available from a higher level, which would provide the *environment profile* updated by *context-based mechanisms* that are passed down to the system [1,3]. But, this increases the weight and the complexity of the overall mobile system. Alternatively, the system can either periodically check or continuously monitor the environment. However, a successful lookup depends on the pluggable discovery components that are embedded. The more there are components, better is the detection. But, the size of the middleware and the resources needed grow with the amount of embedded components. That is particularly not recommended for mobile devices. Furthermore, as long as the current SDP has not been found, the middleware has to reconfigure itself repeatedly with the available embedded components to perform a new environmental lookup until it finds the appropriate protocol. As a consequence, this leads both to an intensive use of the bandwidth already limited due

to the wireless context, and to a higher computational load. To save these scarce resources, a plug-in component, called *discoverdiscovery*, dedicated to SDP detection operations, has been added to the ReMMoC service discovery framework. In an initialization step, *mini-test-plug-ins*, implemented for each available SDP, are connected to *discoverdiscovery* to perform a test by both sending out a request and listening for responses. Once the detection is achieved, a configuration step begins by loading the corresponding complete SDP plug-ins.

The above *Mini-test-plug-ins* are lightweight and so consume fewer resources. Nevertheless, they increase the number of embedded plug-ins, do not decrease the use of the bandwidth and finally have to be specifically implemented. Last but not least, rather than embedding as many components as possible to provide the most interoperable middleware, it seems to be more efficient to design an optimized lightweight middleware, which *enables* loading from the ambient network new components on demand to supplement the already embedded ones [1,14]. But, still, it is necessary to discover, at least once, the appropriate protocols to interact with a service providing such a capability. This is rather unlikely to happen since we do not know the execution context (i.e., all potential available resources and services at a given time).

Summarizing, solutions to interoperability based on reflective techniques do not bring simultaneously interoperability and high performance. The SDP interoperability issue needs to be revisited to improve efficiency of SDP detection, interpretation and evolution. Furthermore, the ReMMoC reflective middleware does not provide a clean separation between components and protocols. In fact, pluggable components are tied to their respective protocols. For example, to maintain interoperability between several versions of the same SDP, a pluggable component is needed for each version. We need a fine-grained control over protocols. Our approach is thus to decouple components from protocols with the use of concepts inherited from software architecture enhanced with event-based parsing techniques.

## 3 Software Architecture to Decouple Components from Protocols

Software architecture concepts, like *components* and *connectors* to decouple applications from underlying protocols, offer an elegant means for modeling and reasoning about mobile systems [17]. Components abstract *computational elements* and bind with connectors that abstract *interaction protocols*, through interfaces, called *ports*, which correspond to communication gateways [10]. Similarly, connectors bind with components through connector interfaces named *roles* (*see Figure 1*).

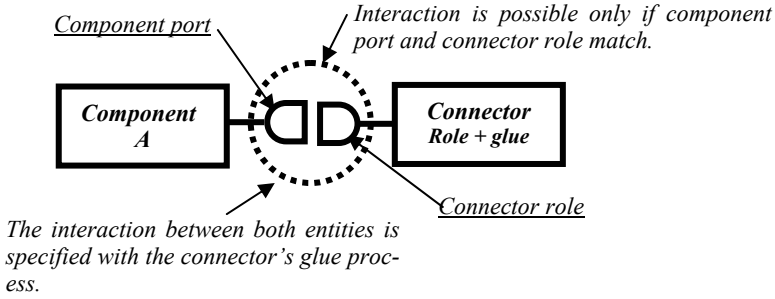

**Fig. 1.** Components decoupled from protocols

Regarding the issue of achieving protocol interoperability, this may be addressed through reasoning about the compatibility of port and role. This may be realized using, e.g., the *Wright* Architecture description language [11]. *Wright* defines *CSP*-like processes to model port and role behaviors. Then, compatibility between bound port and role is checked against, according to the CSP refinement relationship. However, the *Wright* approach does not bring enough flexibility with respect to dealing with the adaptation of port and role behavior so as to make them match when they share an identical aim, as, e.g., in the case of service discovery.

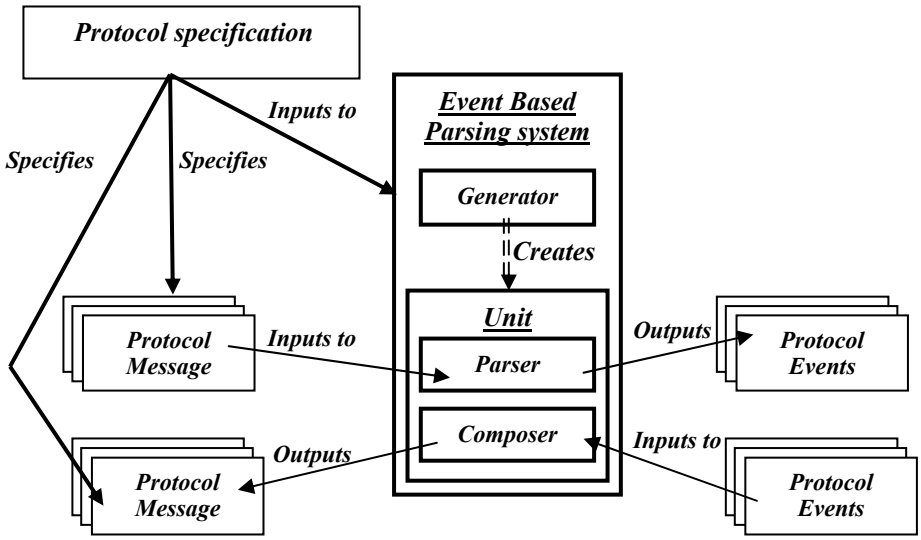


**Fig. 2.** Event based parsing system for achieving protocol interoperability

To overcome the aforementioned limitation, [12] reuses the architectural concepts of *component*, *connector, port* and *role*. However, port and role behaviors are modeled by handlers of unordered event streams rather than by abstract roles processes. The challenge is then to transform protocol messages into events, and interpret them according to a protocol specification. To achieve this, an event-based parsing system, composed of *generator*, *composer*, *unit*, *parser* and *proxy*, is used (*see Figure 2*). A protocol specification feeds a *generator* that generates a dedicated *parser* and *composer*. The former takes, as input, protocol messages that are decomposed as tokens and outputs the corresponding events. The latter does the invert process; it takes series of events and transforms them into protocol messages. *Parser* and *composer* form a *unit*, which is specific to one protocol. *Generators* are able to generate on the fly new units, as needed, for different specifications. As a result, whatever is the underlying protocol, messages from a component are always transformed into events through the

adequate *parser* and conversely, events sent towards a component are always transformed into protocol messages understood by this component through its adequate *composer*. Furthermore, events are sent from one component to another through a *proxy* whose role is to forward handled events to the *composer* of the remote component (*see Figure 3*). The latter can either discard some events if they are unknown or force the *generator* to produce a new *unit* more suitable to parsed events. Thus, any connector gets represented as a universal event communication bus, which is able to transport any event, independently of any protocol, as the protocol reconstruction process is let to each extremity. Thereby, event streams are hidden from components and so protocol interoperability is maintained.

Summarizing, event-based parsing is interesting in theory for its flexibility, and opens new perspectives to overcome protocols heterogeneity. However, it is still confined to theory: it has been applied only to protocol evolution issue, as it is simpler to test protocol interoperability between two similar protocols that differ with only small changes. Therefore, [12] addresses heterogeneity issues neither for SDPs nor for interaction protocols but brings interesting concepts. In the next section, we show how event-based parsing applied to software architecture enables efficient SDP detection and interoperability in the mobile environment.

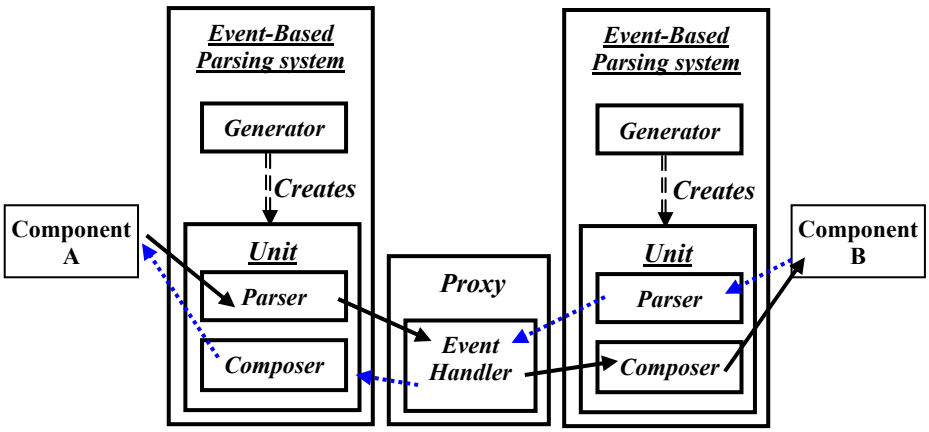


**Fig. 3.** Interaction between two components

## 4 Event-based Parsing for Discovery Protocol Interoperability

With the emergence of mobility and wireless technologies, SDP heterogeneity becomes a major issue. ReMMoC is currently the only middleware to provide a first approach to resolve this issue through the use of the pluggable component philosophy. However, as stated earlier, this solution incurs high resource consumption (i.e., bandwidth, memory and CPU). Our objective is to provide a much more powerful solution, dedicated to the ad hoc network context, which both induces low resource consumption and introduces a lightweight mechanism that may be adapted easily to any platform. To achieve this challenge, we reuse the component and connector abstrac-

tions, and event-based parsing techniques from software architecture. Moreover, as our aim is to provide interoperability to the greatest number of portable devices, we base our technology on IP. The following first briefly introduces conceptual similarities among SDPs (§4.1), and then details our solution, addressing SDP detection (§4.2) and interoperability (§4.3).

### 4.1 Conceptual similarities among SDPs

The majority of SDPs support the concepts of *client*, *service* and *repository*. In order to find needed services, clients may perform two types of request: *unicast* or *multicast*. The former implies the use of a repository, equivalent to a centralized lookup service, which aggregates services information from services 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:

1. The passive discovery model, and
2. The active discovery model

When a repository exists in an environment, the main challenge for clients and services is to discover the location of the repository, which acts as a mandatory intermediary between clients and services [2]. 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 a repository multicast advertisements. 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 that is specific to the SDP 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 reply unicast responses directly to the requester only if they match the requested service. To summarize, most SDPs support both passive and active discovery with either optional or mandatory centralization points.

Note that although service repositories reduce both bandwidth consumption and time for service location, they are not adequate to the dynamic nature of the mobile domain. All the entities from an ad hoc network form spontaneously a purely peer-to-peer architecture, which does not rely on any centralization point. Thus, SDPs, like Jini [4], exclusively based on a lookup server, break the peer-to-peer model and hence, conceptually, it is not advised to use it. However, we introduce a solution to SDP interoperability that supports almost all types of SDPs. The only exception is for the Jini SDP that is tied to the Java language and hence makes it harder to achieve interoperability because it requires that all mobile devices embed a Java virtual machine. In addition, properties of other SDPs must be Java byte-code encoded to allow interoperability with Jini clients. Addressing such an issue is part of our future work so as to fully support SDPs interoperability.

The two next sections detail our solution to SDPs interoperability, which is compatible with both the passive and active discovery models. However, when the SDP provides both models, the passive discovery model should be preferred over the active discovery model. Indeed, with the latter, the requester's neighbors do not improve their environment knowledge from the requester's lookup because services, that the requester wishes to locate, send only unicast replies directly to the requester. So, the services' existence is not shared by all the entities of the peer-to-peer network. Thus, it is unfortunate to not take benefit from the bandwidth consumption caused by the clients' multicast lookups. In this context, services' multicast announcements provide a more considerable added value for the multicast group members. Secondly, in a highly dynamic network, mobile devices are expected to be part of the network for short periods of time. Thus, services' repetitive multicast announcements provide a more accurate view of their availability. Therefore, the passive discovery model saves more the scarce bandwidth resources than the active discovery model.

### 4.2 SDP detection

Basically, all SDPs use a multicast group address and a UDP/TCP port that must and have been assigned by the Internet Assigned Numbers Authority (IANA). Thus, assigned ports and multicast group addresses are reserved, without any ambiguity, to only one type of use. Typically, SDPs are detected through the use of their respective address and port. These two properties form unique pairs. The latter may be interpreted as a permanent SDP identification tag. Furthermore, it is important to notice that an entity may subscribe to several multicast groups and so may be simultaneously a member of different types of multicast groups. These only two characteristics are sufficient to provide simple but efficient environmental SDP detection. Due to the very dynamic nature of the ad hoc network, the environment is continuously monitored to detect changes as fast as possible. Moreover, we do not need to generate additional traffic. 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. As a result, the specific protocol of either the passive or active service discovery may be determined. To achieve this feature, a component, called *monitor component*, embeds two major behaviors (*see Figure 4*):

1. The ability to subscribe to several SDP multicast groups, irrespectively of their technologies; and
2. The ability to listen to all their respective ports.

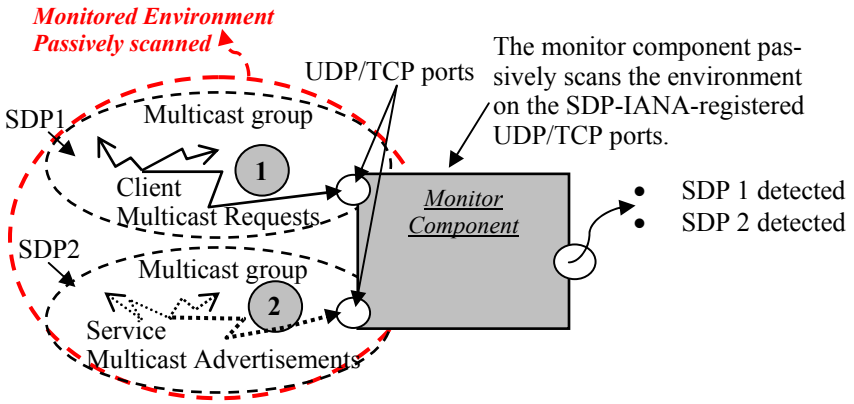

**Fig. 4.** Detection of active and passive SDPs through the monitor component.

Figure 4 depicts the mechanism used to detect active and passive SDPs in a repository-less context. The *monitor component,* located at either the client side or service side, joins both the SDP1 and SDP2 multicast groups and listens to the corresponding registered UDP/TCP ports. SDP1 and SDP2 are identified by their respective identification tag. However, SDP1 is based on an active discovery model. Hence, 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. Still, in *Figure 4*, SDP2 is based on a passive discovery model. So, services advertise themselves to the SDP2 multicast group to announce their existence to their vicinity. Once again, similarly to SDP1, as soon as data arrives 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 nor 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.

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

Our *monitor component* can be either integrated into the ReMMoC middleware or considered as one primary element from a larger software architecture that we describe in the next section. The current ReMMoC *discoverdiscovery* plug-in may in particular be replaced by our *monitor component*, which avoids both implementing mini-test-plug-in for each available SDP and their loading just to perform SDP detection. In this way, we save both scarce bandwidth consumption and computation resources. However, once the detection is achieved, further processing is left to the appropriate SDP plug-in. The ReMMoC SDP configuration step then stays unaltered.

### 4.3 SDP interoperability

From a software architecture viewpoint, SDP detection is just a first step towards SDP interoperability and represents a primary component. The main issue is still unresolved: the incoming raw data flow, which comes to the *monitor component*, needs to be correctly interpreted to deliver the services descriptions to the application components. To support such functionality, we reuse event-based parsing concepts (*see Figure 5*). As a result, upon the arrival of raw data at monitored ports (step 1), the *monitor component* detects the SDP that is used, and sends a corresponding event to the *generator* (step 2), that instantiates the appropriate *parser* (step 3) to successfully

transform the raw data flow into a series of events (step 4). The *parser* extract semantic concepts as events from syntactic details of the SDP detected. Then, the generated events are delivered to a *proxy* (step 5). In its turn, the *proxy* forwards handled events to the local components' *composers* (step 6). Contrary to [12], *parser* and *composer* are not coupled by type. As events bring the necessary abstraction from the SDP syntactic details, events from a *parser* specific to one SDP are understood by a *composer* dedicated to another SDP.

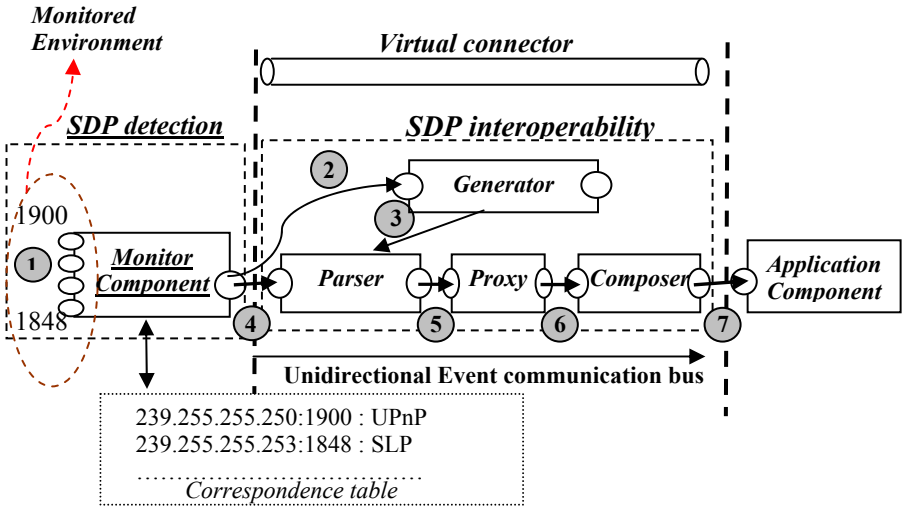


**Fig. 5.** SDP detection & interoperability mechanisms

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. In fact, a fixed set of common events has been defined for all SDPs. The set of common events is itself an event subset of a larger event set dedicated to each SDP. For example, a subset of events generated by a UPnP parser is successfully understood by an 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. Event streams are totally hidden from components as they are reconstructed through *composers* (*see Figure 5, step7*). *Monitor component* and local application components are therefore virtually connected through a *connector*, which acts as a universal event communication bus. Consequently, interoperability is guaranteed to existing applications tied to a specific SDP without being altered. Similarly, future applications do not need to be developed with a specific middleware API to get the SDP interoperability property. Furthermore, application components continue to use their own native service discovery protocol without using the virtual connector, which is unidirectional. Hence, there is no return path and the generator needs to instantiate neither a dedicated *parser* nor a dedicated *composer* to translate replies from the native SDP to the discovered SDP. This makes drastic computation resources economies. Moreover, it is important to note that our SDP interoperability may be applied to both service provider and client application. On the former side, requests, which are generated by clients using protocols other than the service provider's native SDP, are automatically detected thanks to the *monitor component* and transparently

translated through the virtual connector into new semantically equivalent requests but understood by the service provider. Then, the latter replies, according to its native protocol, to the client. The virtual connector acts like a "SDP translator". However, this conversion process is without losses as it is based on the greatest common denominator of the different SDP functionalities. For example, SLP does not manage UPnP eventing mechanism [5] and consequently related messages are simply discarded but this is not a loss as SLP does not support it anyway.
On the client side, the same mechanism occurs : received messages, generated by services using a different discovery protocol from the one used by the client are translated to new messages semantically equivalent but syntaxically different according to the client's native SDP.

## 5 Conclusion

Service discovery protocol heterogeneity is a key challenge in the mobile computing domain. If services are advertised with SDPs different than those supported by mobile clients, mobile clients are unable to discover their environment and are consequently isolated. Due to the highly dynamic nature of the mobile network, available networked resources changed very often. Therefore, this requires a very efficient mechanism to monitor the mobile environment without generating additional resource consumption. In this context, inspection and adaptation functionalities offered by reflective middleware are not adequate to support service discovery protocol interoperability, as they induce too high resource consumption. This paper has addressed this challenge, providing an efficient solution to achieving interoperability among heterogeneous service discovery protocols. Our solution is specifically designed for highly dynamic ad hoc networks, which requires both minimizing resource consumption, and introducing lightweight mechanisms that may be adapted easily to any platform. An implementation will soon be released to validate both its design and efficiency.

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, [7] proposes to use ReMMoC together with WSDL [8] 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 [7] suffers from a major constraint: service and client must agree on a unique WSDL description. But, once again, in a dynamic mobile 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, which must account for mobility constraints and in particular related resource limitation. This issue is addressed by the WSAMI middleware [9], 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 so as to complement our solution to SDP interoperability and thus support middleware interoperability, as required by today's mobile environment.

## Acknowledgements

This work has received the support at the European Commission through the IST program, as part of the UBISEC project (http://www.ubisec.org). The authors would like to thank Paul Grace for providing us with detail about ReMMoc. They are further grateful to anonymous reviewers for useful comments.

## References

1. 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.
2. 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.
3. L. Capra, G. Blair, C. Mascolo, W. Emmerich, P. Grace. Exploiting reflection in mobile computing middleware. In ACM Mobile Computing and Communications Review. May 2002.
4. Sun. Technical White Paper: Jini Architectural Overview. 1999.
5. Universal Plug and Play Forum. Universal Plug And Play Device Architecture. 2000.
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. 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.
8. W3C."Web Services Description Language (WSDL)", W3C Working Draft. 2003
9. 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, 2004. To appear.
10. 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.
11. R. Allen, D.Garlan. A formal basis for architectural connection. ACM Transactions on Software Engeneering and Methodology, July 1997.
12. 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
13. G. Coulson, G. Blair, M. Clarke and N. Parlavantzas. The design of a configurable and reconfigurable middleware platform. In Distributed Computing. April 2002.
14. X. Fu, W. Shi, A. Akkerman, and V. Karamceti. CANS: composable, adaptive network services infrastructure. In Proceedings of the USENIX Symposium on Internet Tecnologies and Systems ( USITS), 2001.

15. Salutation Consortium. White paper: Salutation Architecture. 1998.
16. C. Szyperski “Component Software: Beyond Object-Oriented Programming”. Addison Wesley, 1998.
17. V. Issarny, F. Tartanoglu, J. Liu, F. Sailhan. Software Architecture for mobile distributed computing. In Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA), Oslo, June 2004.