Enabling Ambient Intelligence via the Web

Valerie Issarny, Daniele Sacchetti, Ferda Tartanoglu, Francoise Sailhan

Enabling Ambient Intelligence via the Web*

Valérie Issarny, Daniele Sacchetti, Ferda Tartanoglu, Françoise Sailhan
INRIA, UR Rocquencourt, Domaine de Voluceau, 78153 Le Chesnay, France
URL : http://www-rocq.inria.fr/arles/

Abstract

Enabling the ambient intelligence vision means that consumers will be provided with universal and immediate access to available content and services, together with way of effectively exploiting them. Concentrating on the software system development aspect, this means that the actual implementation of any ambient intelligence application requested by a user can only be resolved at runtime according to the user's specific situation. This paper introduces a base declarative language and associated core middleware, which supports the abstract specification of ambient intelligence applications together with their dynamic composition according to the environment. The proposed solution builds on the Web Services Architecture, whose pervasiveness enables both service availability in most environments, and specification of applications supporting automated retrieval and composition.

1 Introduction

The vision of *ambient intelligence* (also termed *pervasive computing*) relies on provisioning *ubiquitous computing* (i.e., useful, pleasant and unobtrusive presence of computing devices everywhere), *ubiquitous networking* (i.e., access to network and computing facilities everywhere), and *intelligent aware interfaces* (i.e., perception of the system as intelligent by people who naturally interact with the system that automatically adapts to their preferences). While available technologies are significant enablers of the ambient intelligence vision, there is still a number of issues to address before its full realization, requiring advances in most areas relating to the computer science field (e.g., hardware, networking, human-computer interaction, development support). This paper concentrates on one such issue that is supporting the development of ambient intelligence applications, through the introduction of a base declarative language for specifying ambient intelligence applications and associated core middleware infrastructure for the any-time, any-where access to applications. The key feature of our solution relates to enabling the dynamic composition, possibly distributed, of requested services (i.e., functions provided

*This work has been partially funded by the Ozone IST project; http://www.extra.research.philips.com/euprojects/ozone/.

by the computing system) according to the user's situation, while guaranteeing quality of service to users in terms of at least performance and security properties. Our solution further builds on the Web that is pervasive enough for ensuring availability of services in most situations. We more specifically base our work on the Web Services Architecture, which comprises: the XML-based WSDL[1] and WSCL[2] declarative languages for Web Services specification, the SOAP[3] protocol for the exchange of XML documents, and the UDDI[4] registry for dynamically locating and advertising Web Services. The next section introduces the WSAMI language for the specification of Web Services, which enables their dynamic composition according to the user's situation, while keeping associated runtime overhead low. As presented in Section 3, actual dynamic composition of services relies on a minimal core middleware infrastructure, i.e., SOAP enriched with a naming&discovery service, which must be run on any terminal that is willing to take part in ambient intelligence applications. Finally, Section 4 summarizes our contribution and discusses our current and future work.

2 WSAMI

The XML-based WSAMI language allows the specification of Web Services so that they can be dynamically composed according to the environment in which services are requested. The XML specification of a service then decomposes into: the service's abstract interface (§ 2.1) and the non-functional properties associated with the service (§ 2.2). In addition, the notion of *customizer* [7] is exploited for customizing connectors with respect to enforcing non-functional properties, leading to the associated specification of middleware services (§ 2.3). For illustration, we consider the example of a collaborative schedule service that composes a Booking service (e.g., for tennis courts) with Agenda services so as to reach common agreement on booking according to the Booking service's availability and to the availability and preferences of participating users.

2.1 Abstract Interfaces

Using WSDL[5], a Web Service specification embeds: (i) the service's abstract interface that describes the messages exchanged with the service, and (ii) the concrete binding information that contains specific protocol-dependent details including the network endpoint address of the service. In our context, the first part is the basic for retrieving services, while the latter is associated with service instances that are dynamically retrieved by the middleware naming&discovery service according to the environment. Retrieval of a service is based on the matching of the abstract interface associated with the requested service, with the abstract interfaces of reachable instances. However, interaction with a service should also comply with the protocol that is assumed, i.e., the conversation

[1] http://www.w3.org/TR/wsdl
[2] http://www.w3.org/TR/wscl10
[3] http://www.w3.org/2000/xp/Group
[4] http://www.uddi.org
[5] http://www.w3.org/TR/wsdl

that must be realized. Such a specification is enabled by WSCL[6]. Hence, as far as the specification of the service's functional behavior is concerned, the definition of WSAMI is direct from WSDL and WSCL. As an illustration, the following WSAMI element defines the abstract interface of the collaborative schedule service:

```
<Abstract name=''CollaborativeSchedule''>
  <Interface hrefSchema=''http://example.com/schedule/ScheduleRequest.wsdl''/>
  <Conversation hrefSchema=''http://example.com/schedule/ScheduleInteraction.wscl''/>
</Abstract>
```

The WSAMI element `Service` that is associated with any service instance further specifies the WSAMI abstract interface of the service (`Abstract` element) together with concrete binding information (`Concrete` element). Still taking the example of the collaborative schedule service, we get:

```
<Service name=''CollaborativeSportSchedule''
  <Abstract hrefSchema=''http://example.com/schedule/CollaborativeSchedule.wsami''/>
  <Concrete hrefSchema=''http://example.com/schedule/ConcreteSchedule.wsdl''/>
</Service>
```

Dynamic composition of services then relies on a specification matching relationship defined over WSAMI abstract interfaces. Given the request for a service identified by its WSAMI abstract interface, a service instance whose abstract interface (i.e., value of the `Abstract` element) *matches* the one of the requested service, is sought in the environment. Two abstract interfaces are then said to match if their respective documents are syntactically equal. This allows us to keep to a minimum the processing cost associated with checking specification matching: two abstract interfaces match if the related WSAMI documents have the same URI. This further suggests the development of Web Services for ambient intelligence, through the reuse of declarative specifications of services that are made available over the Web. This is consistent with the approach that is put forward for the development of Web Services in general, through the provision of *universal registries* as, e.g., enabled by UDDI.

2.2 Non-functional Properties

A key feature of WSAMI lies in the specification of non-functional properties associated with services, so as to enforce Quality of Service (QoS) in terms of at least security, and performance regarding resource consumption, i.e., the two mandatory criteria for the consumer acceptance of ambient intelligence systems, independent of the users and service providers. The former is enforced through authentication and the establishment of a secure communication channel. The latter is enforced through content filtering. A number of other quality of service criteria are relevant in the ambient intelligence context. However, their handling is left upon the responsibility of the Web Service developers, through either explicit Web Service composition (e.g., exploiting a caching service), or a proprietary middleware for content delivery (e.g., delivery of continuous media) where

[6] http://www.w3.org/TR/wscl10

WSAMI may still be exploited for initiating the service (e.g., negotiating quality of service). QoS requirements decompose into the non-functional properties that must be provided by the service itself (i.e., built in the service implementation) and the ones that must be enforced at the connector level. The latter specifies customization of the middleware for interaction with the service, i.e., middleware services that need be integrated.

The WSAMI specification of abstract interfaces then extends with the definition of the `ServiceQoS` and `ConnectorQoS` elements that set the non-functional properties associated with the service and connector, respectively. Properties are given in terms of the corresponding QoS criteria using the `QoSCriterion` element. Still considering the collaborative sport schedule service, we get the following enriched specification for the abstract interface associated with the embedded Agenda service:

```
<Abstract name=''Agenda''
      xmlns:qos=''http://www-rocq.inria.fr/arles/wsami/qos.xsd''>
  <Interface hrefSchema=''http://example.com/schedule/AgendaRequest.wsdl''/>
  <Conversation hrefSchema=''http://example.com/schedule/AgendaInteraction.wscl''/>
  <ServiceQoS>
      <QoSCriterion name=''qos:transactionalService''/>
      <QoSCriterion name=''qos:security''/>
  </ServiceQoS>
  <ConnectorQoS>
      <QoSCriterion name=''qos:security''/>
  </ConnectorQoS>
</Abstract>
```

QoS criteria are further defined through the following schema, considering support for transaction and security:

```
<xsd:schema xsd:id=''qos''
      xmlns=''http://www.w3.org/2001/XMLSchema'' ...>
  <xsd:element name=''transactionalService'' type=''xsd:anyType''>
  <xsd:element name=''security'' type=''xsd:anyType''>
</xsd:schema>
```

The WSAMI matching relationship over abstract interfaces then enforces matching behavior of services with respect to both functional properties and non-functional properties stated in the abstract interface. If QoS criteria should be enforced over the connector, as specified using the `QoSConnector` element, the connector must be customized, which may lead to quite complex interactions among services and can not be realized automatically in general. However, such automated customization can be supported for QoS properties that are enforced through middleware services adhering to the pipe&filter architectural style. Such a requirement is in particular met by QoS requirements relating to security and performance, as further discussed in the next subsection. The handling of QoS criteria that do not meet the above requirement should be dealt with by the Web Service developers who should make explicit the interactions with associated middleware services in the Web Service implementation.

2.3 Automated Connector Customization

A basic way to improve performance in a distributed environment with varying bandwidth and potential resource-constrained devices is to reduce the complexity of the content that is sent over the network. This may be achieved using specific proxy nodes that filter the content for mobile terminals [3], possibly leading to introduce a new system of protocols and document types for interaction with the wireless client as in the WAP (Wireless Access Protocol). An alternative approach is to use a pair of content customizers at both ends, as presented in [7]. Our approach is based on the latter, which may further be exploited for dealing with security requirements but also other QoS criteria such as reliability as presented in the aforementioned reference. A Customizer decomposes into a local and a remote service, i.e., the latter must be available in the environment of the client while the latter must be in the environment of the server. Then, any message exchanged between the client and the server goes through the customizer. As an example of customizer specification, the element that is defined below introduces a customizer enforcing security, through the use of symmetric cryptography for the encryption of messages that are exchanged between the client and the server. The management of public keys associated with authentication is further integrated within the middleware naming&discovery service. Note that it may not be possible to authenticate a service with which secure communication is required, if communication must occur with a certifier and the connectivity does not allow so. In this case, the request for the Web Service fails.

```
<Customizer name=''EnforceSecurity''
      xmlns:qos=''http://www-rocq.inria.fr/arles/wsami/qos.xsd''>
  <QoSCriterion name=''qos:security''/>
  <Local hrefSchema=''http://example.com/QoS/SymCrypto.wsami''/>
  <Remote hrefSchema=''http://example.com/QoS/SymCrypto.wsami''/>
</Customizer>
```

In the above definition, the `QoSCriterion` part specifies the specific QoS criteria that are enforced by the embedding customizer, and the `Local` and `Remote` parts specify the abstract interfaces of the customizer services, which must be run *close* to the client, and to the server, respectively. While in the general definition of customizers, the distance between the customizer services and their associated end-point is left quite open due to the focus on the Internet, they are here enforced to be co-located with their associated end-point due to our concern of effectively supporting mobile nodes.

The benefit of using customizers is dependent upon the environment in which the service is requested. While enforcing security is almost always mandatory, minimizing resource consumption using, e.g., filtering is dependent upon available network bandwidth and connectivity. We rely here on the specific implementation of customizers to adapt to resource availability. A more flexible solution could be undertaken by allowing to specify constraints over resource availability for both the use of customizers and QoS requirements. However, this would lead to more complex computation upon composition but also requires more cooperation from the local environment, which contradicts our goal of

introducing a minimal core middleware. In addition, as raised earlier, complex connector customization can still be realized through explicit composition of middleware services.

3 WSAMI Middleware

The core middleware associated with Web Services lies in the provision of SOAP containers that are able to deploy Web Services, and to manage RPCs from SOAP clients and dispatch them to services. There already exist various implementations of the above core middleware[7], which in our context may be deployed on possibly resource-constrained mobile terminals. However, we do not consider that the deployment of Web Services on mobile platforms is a major issue given ongoing work in the area. We are thus more specifically concentrating on the design and implementation of the middleware naming&discovery service (simply referred to as ND service in the following), which supports the dynamic composition of services according to the user's situation, given the services' WSAMI specification. We are in particular interested in exploiting both ad hoc and infrastructure-based wireless networks for enhanced connectivity and hence enhanced service availability. The ND support for dynamically locating requested services lies in: (i) the management of repositories of services' abstract interfaces and instances (§ 3.1), and (ii) locating instances of services that are reachable both in the local and in the wide area (§ 3.2). In addition, the ND service handles connector customization together with authentication of service instances (§ 3.3).

3.1 Service Repository

The ND services manages two repositories, which respectively allow retrieving information about services from their abstract interfaces, and about services instances that are locally supported. The former repository is actually a local cache whose content evolves according to the history of user requests, i.e., the size of the repository is set according to available storage and entries are removed according the cache's replacement policy, which currently adheres to the Least Recently Used (LRU) policy. The design of a dedicated replacement policy is part of our future work based on experimental results, regarding in particular user profiles in terms of service requests. Advanced automatic prefetching techniques with respect to the user's profile might further be exploited, although not part of our current design. Our concern of minimizing resource consumption on mobile terminals together with the fact that our approach relies on *universal repositories* of Web Services specification (including WSAMI specifications) lead us to not store nor process XML documents on the terminals for service discovery. Instead, only related URIs are exploited, which is enabled by the specific design choices for WSAMI.

Each element of the *local repository associated with WSAMI abstract interfaces* (i.e., documents that define `Abstract` elements) allows getting the following information about

[7] See for instance the list at http://www.xmethods.net/ve2/ViewImplementations.po

the corresponding service: (i) the URI of the document defining the corresponding abstract interface, and (ii) the list (possibly empty) of *known* matching service instances where each element of the list is a pair that gives the URI of the corresponding service, and the actual binding information. Known service instances further decompose into: (i) the list of known instances available on the Internet, and (ii) the list of instances discovered in the local environment, through the ND service support for their discovery as discussed in the next subsection.

Elements of the *local repository associated with local service instances* are identified through the URIs of the corresponding `Service` documents. Each element allows getting the following information about the service instance: the respective URIs of the documents defining the service's abstract interface, instance, concrete interface, and the URI of the document defining the QoS criteria associated with the service connector -if any- (i.e., `ConnectorQoS` part of `Abstract`). The repository may then be requested for an entry given the associated service URI, but also for entries matching a provided abstract interface URI.

3.2 Locating Service Instances

The above repositories provide the base functionalities for handling requests for any service, given the service's abstract interface. An instance is first searched locally through a request to the repository of local instances. If an instance is available, request for the service's execution may proceed. Otherwise, a remote instance needs to be retrieved. Nodes that are contacted then depend on the underlying wireless network, including whether it is infrastructure-based or ad hoc. We are in particular interested in the exploitation of WLANs supporting the ad hoc mode, since communication does not incur any financial cost for the consumer. Using a WLAN such as IEEE 802.11b, the network may be run in either the ad hoc or infrastructure-based mode, where the latter requires availability of a base station in the local communication range. We further assume that the network may be switched from one mode to another, depending on the availability of a base station and whether the user accepts to be charged for communication. Then, instances may be retrieved either via the Internet if the network is in the infrastructure-based mode or via nodes in the local communication range in either mode. The retrieval of instances that are available in the local environment relies on an underlying service location protocol [1], where we are more specifically experimenting using the SLP standard. Using such a support, nodes running the ND service may be discovered in the local environment, leading to get necessary binding information as well as to identify whether the nodes are power-plugged or not. Then, given the request for a service that does not provide a specific instance, a matching instance is retrieved according to the following process:

- If the network is in the infrastructure mode, the following sequential steps are performed. The service is first requested to ND services in the local area that are run on power-plugged nodes. Otherwise, it is sought whether an instance available on the Internet is known, as given by the local repository of abstract interfaces,

which will be the one selected. If an instance is still not retrieved, the service request is sent to the ND services run on the wireless nodes in the local area. Ultimately, the service request is sent *via* the Internet, to the universal repository that is provided by the user upon ND initialization.

- If the network is in the ad hoc mode, the service is requested to nodes in the local area, interacting first with power-plugged nodes. This may ultimately lead to notify service unavailability to the user. Note that increased availability could be achieved by exploiting ad hoc routing protocols as, e.g., addressed in [6] for the specific case of cooperative caching.

In the case where multiple instances are retrieved, one is chosen randomly, although performance could still be improved by taking into account the resources available on the hosting nodes.

It may be the case that a given service instance gets replicated over various devices (e.g., personal Agenda service that may be replicated over the user's mobile devices while the persistent copy is on the user's home system). This raises the issue of which instance to access when there is a replica accessible in the local communication range. There is no single optimal solution to this issue since this depends on various factors, e.g., available connectivity (i.e., available base station or not), resources that are available on the terminal(s) in the local environment hosting the replica(s), coherency of the replica(s) with respect to the persistent copy that is managed by the service itself in a way transparent to the user. Our primary design choice is to minimize resource consumption on mobile terminals whenever possible, leading in particular to favor the selection of services run on power-plugged nodes in our retrieval process. In addition, every service is associated with an additional attribute that sets whether its instance should be retrieved in priority from its home server or any terminal hosting a replica (i.e., the `Service` element is extended with the attribute `<home =''true | false''/>`, and if `home` is set to false, any replica that is reachable will be accessed first). The retrieval process that is given above is modified accordingly.

3.3 Connector Customization

The above retrieval process should further be enriched to cope with quality of service requirements, which decompose into: (i) authenticating the nodes with which interactions take place so as to enforce communication with trusted nodes, and (ii) customizing the connector.

The service instances that are selected need be authenticated for the sake of security[8]. In our context, authentication subdivides into: (i) authenticating an instance with respect to its published interface (i.e., the instance does realize the abstract interface it is supposed to), (ii) authenticating a given instance. The former applies to instances that are

[8]Note that we do not consider the case of user authentication since associated support is built in the service.

selected at runtime given an abstract interface, the latter applies to instances that are specified in the service request. We rely on certificates for both types of authentication, and every node contacts the certifiers on a regular basis to obtain up-to-date public keys. In the worst case, the public key on the terminal may have changed since its connectivity enabled interaction with the certifier. In this case, authentication may not be possible, leading to notify temporary service unavailability to the user.

In the case where a remote instance is accessed, connector customization is necessary if quality of service criteria are stated in the `QoSConnector` element of the service's abstract interface. However, customization needs only to be realized on the client-side since, by definition of the service's abstract interface, the service instance already accounts for the connector customization. Then, what needs to be achieved is customization on the client-side, which should match the one already achieved on the server-side. Every service instance thus keeps track of the specific customizers that are used, and in particular of the URIs of the related local customizers (i.e., the information is kept in the repository of local service instances). Then, hit messages in reply to service requests sent by the ND service embeds these URIs as part of the information about eligible service instances. Upon receipt of a hit message for a requested service, the requester checks for local availability of the local customizers embedded in the message, and the service will be selected only if the customizers are available. Increased availability of services may be achieved by ultimately downloading the code of the local customizer service, with respect to specific platforms. We will experiment with such a facility in our middleware prototype, which will be based on Java, but this is not part of the core middleware specification, since it is platform-specific.

4 Conclusion

The vision of ambient intelligence is among today's most challenging topics for information technology. Realizing the vision means that consumers will be provided with universal and immediate access to available content and services, together with ways of effectively exploiting them, which raises a number of issues relevant to most areas of computer science. This paper has concentrated on one such issue that is supporting the development of ambient intelligence software systems.

Our solution primarily lies in the development of ambient intelligence applications in terms of the composition of services that are defined through their abstract interfaces. The ambient intelligence requirement of enabling anytime, anywhere access to applications from any terminal further leads to bind with related services instances at runtime, according to the environment in which the service is requested and in particular service instances that may be reached. Such a facility then requires a software technology that is pervasive enough for being able to rely on both consistent specification and availability of services in most environments, so as to actually support anytime, anywhere discovery of service instances from abstract interfaces. This has led us to base our solution on the

Web, and more specifically on the Web Services architecture. Our solution then lies in the XML-based WSAMI declarative language for the specification of Web Services taking part in the realization of ambient intelligence applications, together with associated core SOAP-based middleware. The language allows for dynamically retrieving instances of services matching the realization of a requested application. Actual composition of services at runtime relies on the core middleware, which amounts to supporting SOAP and to a naming&discovery service for the dynamic retrieval of service instances, both in the local and the wide area, according to the network connectivity and associated cost.

Supporting the development of ambient intelligence or pervasive computing systems has given rise to extensive research over the last couple of years, which has led to introduce a number of complex middleware services that place high demand on the underlying platform and hence limit deployment in most environments (e.g., [2, 4, 5]). Our contribution lies in the definition of a minimal middleware infrastructure for the actual dynamic composition of services, i.e., a naming&discovery service in addition to SOAP, which allows for its wide deployment but also incurs minimal overhead in terms of resource consumption and response time. We are currently implementing a first prototype, using an existing Web Services platform. We will then assess our solution through the development of ambient intelligence demonstrators. We are further investigating the exploitation of user profiles for the naming&discovery service, which will in particular enable the integration of dedicated caching and prefetching techniques for enhancing response time.

References

[1] 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.

[2] M. Esler, J. Hightower, T. Anderson, and G. Borriello. Next century challenges: Date-centric networking for invisible computing. In *Proceedings of MOBICOM'99*, 1999.

[3] A. Fox, S. D. Gribble, and Y. Chawathe. Adapting to network and client variation using active proxies: Lessons and perspectives. *Special Issue of IEEE Personal Communications on Adaptation*, 1998.

[4] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. Project Aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2), 2002.

[5] D. Milojicic, A. Messaer, P. Bernadat, I. Greeberg, O. Spinczyk, D. Beuche, and W. Shroder-Preikschat. ψ - pervasive services infrastructure. In *Proceedings of TES'01*, 2001. LNCS 2193.

[6] F. Sailhan and V. Issarny. Cooperative caching in ad hoc network. In *Proceedings of the 4th International Conference on Mobile Data Management (MDM)*, 2003. to appear.

[7] J. Steinberg and J. Pasquale. A Web middleware architecture for dynamic customization of content for wireless clients. In *Proceedings of the WWW'02 Conference*, 2002.