Middleware for Ubiquitous Service-oriented Spaces on the Web

Geert Vanderhulst, Kris Luyten, Karin Coninx
Hasselt University and transnationale Universiteit Limburg
Expertise Centre for Digital Media – Institute for BroadBand Technology
Wetenschapspark 2, 3590 Diepenbeek, Belgium
{geert.vanderhulst,kris.luyten,karin.coninx}@uhasselt.be

Abstract

Web services are todays preferred vehicle for creating service-oriented architectures (SOA). Due to the boom of personal networked devices, the Web also found its way to the mobile space. This gives rise to device hosted mobile services that deliver functionality over a wireless network. There is potential to unite these services in a collaborative service space and exploit them as building blocks for distributed applications. This paper concentrates on a light-weight middleware layer to facilitate the integration, accessibility and orchestration of Web-based services in heterogeneous environments. Leveraging Web technology, this layer allows to interconnect serverless, peer-to-peer services deployed across federations of devices. We present its value in a ubiquitous application.

1 Introduction

With the advent of affordable (portable) Internet-enabled devices, new opportunities arise to allow devices – and in particular the services that run on these devices – to collaborate. When integrated in a ubiquitous environment, collaborative interactive services can help to accomplish the vision of ambient intelligence (AmI), i.e. enable the environment to become aware of the user(s) interacting with it [3].

A mobile Internet can play a pivotal role in the creation of such ambient service-rich environments. The Web has already evolved to a natural habitat for information providing services and online businesses and lately, ubiquitous services with small footprint are built using Web technology as well [4, 5].

There are, however, some difficulties to overcome in order to use and deploy interoperable services on heterogeneous devices. The variety of devices and their capabilities manifests itself in different ways: wired versus wireless connections, always versus temporary connected, limited versus abundant processing power, etc. This enforces service implementations to adapt to a certain quality of service that can be delivered by the devices and e.g. outweigh performance in favor of reliablity. Another issue is due to the explicit notion of a client and server in most (Web-oriented) distributed systems. Intelligent services and agents both consume and produce information and act as peers that require a many-to-many communication medium to interact with each other.

The goal of the work postulated in this paper is to facilitate the integration of interactive services in ubiquitous environments. We want to support smooth collaboration with both co-located and remote services that is transparant for the user. For this purpose, we created a portable middleware layer that supports the routing and exchange of arbitrary messages in a ubiquitous environment. This layer is designed as a generic Web-based messaging framework that supports a high degree of flexibility for developing and deploying Web-accessible applications on heterogeneous devices.

The rest of the paper is organized as follows. Section 2 provides a short background to existing work. Next, section 3 discusses the major requirements to enable interactive service spaces. In section 4, we motivate the need for light-weight mobile Web services and section 5 presents the Web to Peer (W2P) middleware framework to support these services. Section 6 presents a case study in which a W2P implemention is used as service space glue. Finally, section 7 draws conclusions and states some future work.

2 Related work

In [3], Kirste presents self-organizable device ensembles (appliances) to enable smart user-aware environments. It is assumed that all devices employ event pipelines with the environment. When elevated with

personal computing devices (PDAs, smartphones, ...), we argue an ambient environment will impose on similar P2P communication channels between heterogeneous ensembles of devices and their services.

The WSAMI middleware [4] proposes a solution for the dynamic composition and deployment of WS on both stationary and mobile devices. A Java-based prototype implementation is available which consists of a Naming&Discovery (ND) component and a WSAMI core broker component with integrated compact SOAP container (CSOAP). In contrast with WSAMI, our approach breaks with the tradition to embed Web services (WS) inside a (SOAP) container backed by a server. W2P middleware rather offers a light-weight scalable library that can be used to turn code (Java, C++, ...) into a Web-accessible service with minimal effort.

Several technologies have been proposed for creating device-level SOAs, most notably Jini [1] and UPnP [2]. In these systems (mobile) devices are discovered together with the services they offer. For example, a DVD player can advertise itself as a UPnP device on the local network (with its functions bundled as a UPnP service). Opposed to these frameworks, the W2P framework relies on a central registry to advertise and query services independent of their physical location and subnet (local or wide area). Recent research depicts an emerge of WS technology to the embedded device market as well [5].

[7] presents a Message-Oriented Middleware (MOM) system (called WSMQ) to enhance the value of WS. Some of the design concepts outlined in this work (e.g. WS message queues) are also found in our implementation, yet refined for a mobile context of use.

3 Interactive service spaces

In a ubiquitous computing environment, services are often distributed among computing resources that are available in the environment. A service space virtually unites the services present in the environment: it represents the environment as a software layer of service objects. We distinguish between a *local service space* where services are location-based w.r.t. the user's venue and a *remote services space* that unites services published on a remote network.

As an example, figure 1 depicts an environment with three services that run on different resources: a music service, a VOIP service and the AmI service, that observes the (independent) services present in the environment in order to let them operate as if they were a logical whole. The AmI service reacts, for instance, on an incoming VOIP call with a request to the music service to pause the music during the call. Furthermore, the user can interact with the environment using a distributed user interface. When the user starts an application on her PDA, a presentation is displayed to interact with individual services or with her surroundings in general. Both the services and UI immediately react on state changes. When provided with context information (e.g. location of the user, preferences, etc), the AmI service can make more profound decisions to guide the behavior of the system, yet this is beyond the scope of this paper.

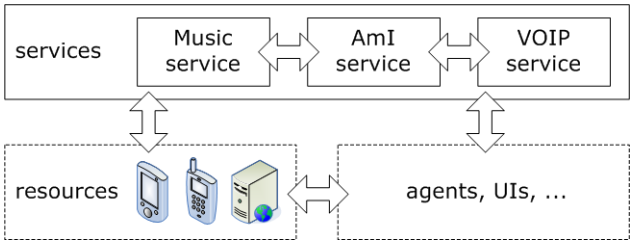

Figure 1. An environment and its services.

The main requirements to enable interactive service-rich environments as presented in this scenario can be summarized as follows:

1. *Smoothless integration of services in a space:* Services are implemented in different ways, for instance as stand-alone processes, deployed inside a runtime environment or embodied in appliances and applications. To inject a service in the space – independent of its implementation – a uniform *publish mechanism* is required. It is also necessary to annotate the service with a machine-processable description of its features and/or API so that other software components know how to interact with it. Semantic annotations pave the way to intelligent service lookup (e.g. find a service based on its purpose) and dynamic service interfacing (i.e. discover how to interact with a service at runtime).

2. *Uniform access to services:* Being able to access a service is a requisite for collaborating with it. When the backend of the service space acts as a registry service, entities can lookup services in the space at runtime. Besides, a *discovery mechanism* can enable entities to live discover distributed services, though discovery solutions are merely based on multicast and limited to the local subnet.

3. *Flexible collaboration with services:* In an interactive system, software entities exchange information, monitor and query data, invoke actions, etc. As such, services can share their features and collaborate with each other. However, tight-coupling

of services should be avoided because it undermines the robustness of a system. Especially in a dynamic environment where services may join and leave the service space all the time, it is hard to enforce complex dependencies. An *eventing mechanism* can avoid this coupling by providing constructs to notify software components of certain events that occur in a service. This permits to design the behavior of an interactive computing system on a higher level: achieving a desired behavior is mainly a manner of associating the right events with the right actions. For instance, events yield the decoupling of the VOIP and music service in figure 1 whilst preserving the option to relate both services through an independent software component (i.e. the AmI service).

4 Mobile Web-based services

A Web service (WS) is deployed in a Web server runtime environment (e.g. a servlet container such as Tomcat[1]) which is responsible for dispatching messages to the service. The runtime environment provides a message processing facility (provider agent) that can receive messages from a requestor agent. These agent components may support one or more network transport protocols, such as HTTP or SMTP. Messages are commonly encoded in XML and binded to a concrete transport protocol (usually SOAP) as defined in a WS's description (e.g. WSDL document). Service providers can advertise the services they offer via the Universal Description, Discovery and Integration (UDDI) protocol, allowing service consumers to discover services that suit their requirements.

Because WS are independent of development technology, they can basically be accessed from any Internet-enabled device. The request/response mechanism typical to Web-based communication allows firewall-friendly execution of remote procedure calls (RPC). Yet, recent WS specifications[2] (WS-Addressing, WS-Transaction, ...) indicate an emerging trend towards asynchronous peer-to-peer messaging. This type of interaction is very suitable for loosely coupled distributed environments. Considering the ubiquitous nature of the Web and its availability on a wide range of devices, WS are a powerful tool for building interactive service spaces. Nevertheless, several issues still prevent WS from becoming the preferred middleware in dynamic ubiquitous environments:

- The implementation of current WS toolkits is often constrained to a subset of WS specifications. Besides, WS toolkits mutually diverge in features, architecture and API which is a result of the fact that protocols are standardized, not their implementations [8]. This makes it hard to leverage knowledge from one toolkit to the other and implement WS in a consistent way across platforms.

- WS highly depend on a runtime environment. The runtime allows the service developer to focus on the core functionality of the service instead of on communication protocols etc, but it also restricts the "mobility" of a service. The footprint and server-oriented nature of WS runtimes makes it rather awkward to publish (part of) a user application as a WS or deploy a WS on an embedded device such as a PDA.

- Most WS-specifications specifically target service-to-service communication. Hence, these specifications ignore non-services settled at the client side such as software agents and UI components. For example, the proposed WS-Eventing specification allows WS to subscribe to event notification messages from other WS but requires entities to be WS-addressable (i.e. run in a Web server).

To cope with these issues, we introduce a lightweight variant of traditional WS: *mobile Web services* (MWS). MWS share the core concepts of WS – they are platform-independent, language-neutral, loosely coupled and they leverage Web protocols – but they do not run inside a containing runtime environment. Instead, they rely on a Portable Middleware library (PM) for interacting with other MWS as shown in figure 2. The PM can be used to implement a MWS as well as to communicate with one. It provides MWS with incoming and outgoing message channels. Messages sent over these channels pass through a gateway (the W2PS) that routes the messages to their destination. This is discussed in detail in section 5. Conventional WS initiate a connection with a remote Web server and then exchange messages with a target WS over this connection.

MWS are targeted to run anywhere, ranging from mobile platforms to dedicated server platforms. They can be encapsulated in user applications (e.g. as a plugin) or run as stand-alone applications. As a consequence MWS have to deal with the issues and limitations inherent to the environment they are deployed in. The PM tries to mask these by providing consistent software interfaces for various platforms.

[1] `http://tomcat.apache.org/`

[2] `http://www.ibm.com/developerworks/webservices/standards/`

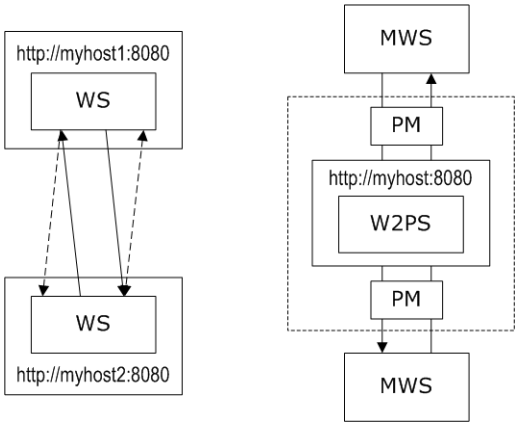

Figure 2. WS versus MWS interaction

5 Web To Peer middleware

The W2P (Web To Peer) middleware framework targets the creation of interactive service spaces. It is built around a centralized WS runtime environment in which a dedicated WS is deployed: the W2P service (W2PS). The W2PS is layered on top of the core of the framework and is accessible via a built-in proxy in the PM (figure 2). The PM tries to maintain a persistent or timed two-way connection with the W2PS so that both components can contact each other. If a connection is lost e.g. due to weak WiFi reception, the PM will automatically try to reconnect. Likewise, an entity is unregistered from the space when it remains offline for a predefined period of time.

In the W2P framework, all entities are considered as equal peers on the network. Entities rather differ in the role they fulfil: a provider role (e.g. service) and/or a consumer role (e.g. user agent).

5.1 Addressing

When a software entity registers with the W2PS, a naming service provides it with a unique name w.r.t. the service space. This is either a reserved name or an enumerated name (e.g. `guest0`, `guest1`, ...) by which entities can address each other. A reserved name is statically appointed to one particular software entity, often used for software entities that are known to participate in the service space. An enumerated name allows new software entities to join a service space without interfering with existing software entities. In practice this has shown to be a simple but effective approach: most distributed applications have a predefined set of services that make up the core of an application and live during the application's lifetime. Other entities simply join the service space for interacting with particular services and are not directly addressed by others (apart from the W2PS).

The W2PS also allows to address a group of entities at once. Entities can join or leave groups which are created on the fly. A group will typically contain similar entities (e.g. MWS with matching descriptions) or entities that have something in common (e.g. interested in the same type of events). For instance, a number of similar entities with assigned names `guest0`, `guest1`, ... can subscribe themselves to a `guest` group. Entities can then address a message to this group and use the group name (`/guest`) as a multicast address to forward the message to each member of the group. Groups are hierarchically organized similar to a unix filesystem tree: `/` is the root group, `/guest` and `/music` are child groups of `/` and `/guest/fun` points to a child group of `/guest`. When a message is addressed to a group, it is sent to all entities in the group and to all entities in its ancestor groups. Nested groups are particularly useful for eventing, as outlined in section 5.3.

5.2 Messaging

Entities communicate with each other by exchanging messages. A W2P message carries a number of headers with meta-data (e.g. sender, recipients, parameters, ...) and a payload with the message's content data and possibly a number of attachments. The payload can contain any kind of information: plain text, XML data, binary image files, etc. Figure 3 lists the skeleton of a W2P message when transmitted using the HTTP protocol.

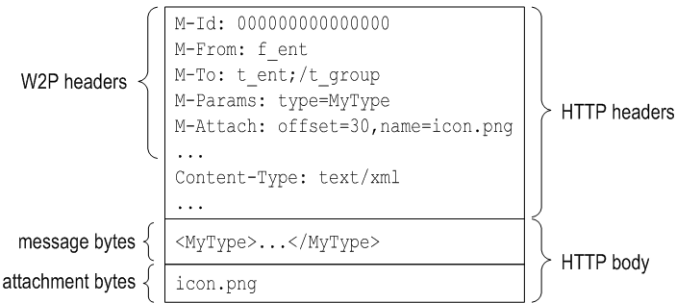

Figure 3. W2P message

To properly deal with messages, the W2P framework depends on two layers: a transport layer is responsible for the transfer of messages over the network and a data binding layer is applied to efficiently build and process raw messages (see figure 4). Besides, the PM and W2PS are equipped with incoming and outgoing message queues.

The W2PS has a shared message queue for incoming

messages and a per-entity queue for outgoing messages. The queues implemented in the PM ressemble an entity's incoming and outgoing mailbox. An entity can simply put a message in its outgoing queue and short time later it will arive in the recipient's incoming queue. Under the hood, the message is sent to the W2PS (by default over HTTP) as soon as a network connection is available. Next, the W2PS routes the message to the recipient's outgoing queue (at the W2PS). When the recipient is online, the message is transferred to its incoming queue. Otherwise the message is kept at the W2PS and the entity can pick it up later.

The W2P framework supports both one-way asynchronous messaging and two-way (request/reply) synchronous messaging. Entities can also engage in "persistent" conversations by continuously replying each other's messages. The chain of messages in such conversation is linked together by their message identifiers.

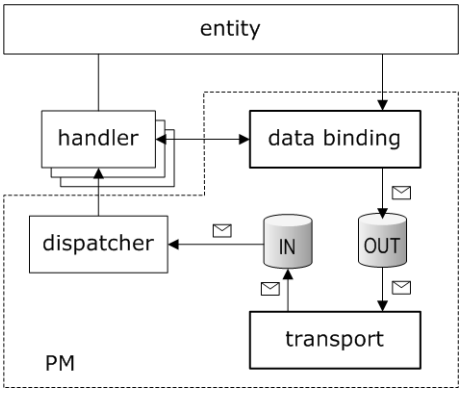

Figure 4. Messaging architecture

When W2P's rule-based dispatch mechanism is enabled, incoming messages are automatically dispatched to dedicated message *handlers* which is also illustrated in figure 4. An incoming message is matched against a set of dispatch rules that relate a number of message properties (e.g. sender, recipient(s), type, ...) with an handler. When a rule matches, the message is passed as an argument to the handler defined in the rule. The handler can appeal to a data binding layer to map the data in a message to native software objects. An XML databinding layer relieves the developer of validating and parsing raw XML data, querying DOM trees, converting extracted DOM data to concrete datatypes (error-prone), etc. When elevated with efficient parser techniques (such as presented in XML Screamer [6]), this layer can reduce the overhead inherent to data (de)serialization to a minimum and boost the overal performance of the application. Since data binding solutions are tightly-coupled with programming languages, the data binding layer is not completely integrated in the W2P framework. However, plenty of toolkits can be used in conjunction with W2P to automate the mapping between XML documents and native objects (e.g. JAXB[3], Liquid XML[4]).

5.3 Eventing

The propagation of events in the environment relies on addressing and messaging techniques when entities respect the following convention: an entity should direct its event messages to a group with path `/[entity name]/[event name]`. The VOIP service in figure 1 sends its incoming call event messages to `/voip/incomingcall`. When an entity is interested in an incoming call event, it should subscribe to this group and if an entity wants to receive all events of the VOIP service, it can subscribe to `/voip`.

6 Case study and evaluation

As a proof of concept, we used the W2P framework to design an interactive service space as outlined in section 3. The music service is built as a (C++) plugin for the XMMS[5] media player. The AmI service is simulated using Java technology and the VOIP service and UI for the music service run as C# code on a Pocket PC device. Due to the scalability of the W2P middleware, the diversity of platforms and development technology is not an issue. On the contrary, it promotes the design of a distributed system where technology matches an entity's local software environment.

Figure 5 illustrates the sequence of actions that take place to mute the music when a VOIP call comes in. It shows that entities first register with the W2PS, subscribe to the appropriate event groups and then can interact with each other. To the developer it appears as if native objects (IncomingCallEvent, PauseRequest) are "emailed" through the service space. These messages are transfered over the HTTP protocol and deserialized in the middleware. In the data binding layer, we used a simple XML data binding mechanism to map XML documents to native objects and vice versa. In summary, data is selected using XPATH expressions and converted to native data types (e.g. an integer, an array of floats, etc) according to information in the XML schema. The data binding is managed by the developer, though for complex projects this process can be automated as stated in section 5.2.

[3] `http://java.sun.com/webservices/jaxb/`
[4] `http://www.liquid-technologies.com/`
[5] `http://www.xmms.org/`

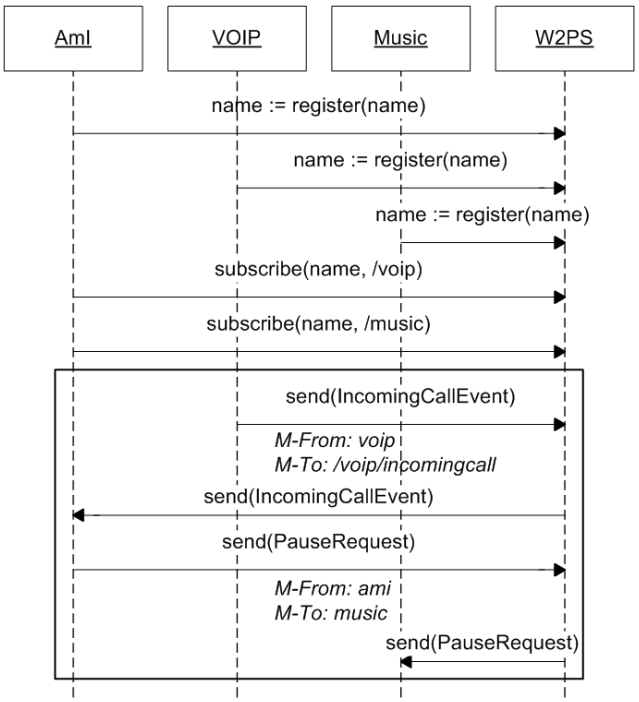

Figure 5. MWS handling of a VOIP mute call

7 Conclusions and future work

Mobile services can be carried on a personal device and executed on an "as-needed" basis, enabling tomorrow's innovative applications. Web services are very suited to publish widely accessible services on a network, but still impose some scalability issues, in particular on mobile platforms. The W2P framework presented in this paper provides a light-weight middleware solution to integrate services in a ubiquitous environment, while being completely independent of the underlying platform. We believe a merit of W2P technology is the ease of turning new or existant functionality into a Web-accessible service and this in a consistent way across platforms.

Further research should go into the description of W2P-based services. The WSDL specification[6] can serve for this purpose when extended with W2P bindings. Another possibility is to use ontologies to describe a service along with its relations with the environment, e.g. the device it is deployed on. Besides, an integration with SOAP-based WS specifications will enhance WS compatibility. This mainly requires extensions to the W2PS (and its runtime environment). For example, support for the WS-Addressing specification can be obtained by assigning W2PS-encoded URIs to entities and groups, e.g. `http://host:8080/w2ps#entity`.

The source code of the current W2P implementation (LGPL) can be found on `http://research.edm.uhasselt.be/w2p`.

[6] `http://www.w3.org/TR/wsdl`

Acknowledgments

Part of the research at EDM is funded by European Fund for Regional Development (ERDF), the Flemish Government and the Flemish Interdisciplinary institute for Broadband technology (IBBT, iConnect project).

References

[1] *Jini*. `http://java.sun.com/products/jini/`.

[2] *Universal Plug and Play (UPnP)*. `http://www.upnp.org/resources/`.

[3] José L. Encarnação and Thomas Kirste. Ambient Intelligence: Towards Smart Appliance Ensembles. In *From Integrated Publication and Information Systems to Virtual Information and Knowledge Environments*, pages 261–270, 2005.

[4] V. Issarny, D. Sacchetti, F. Tartanoglou, F. Sailhan, F. Chibout, R. Levy, N., and A. Talamona. Developing Ambient Intelligence Systems: A Solution Based on Web Services. *Journal of Automated Software Engineering*, 12:101–137, 2005.

[5] François Jammes, Antoine Mensch, and Harm Smit. Service-Oriented Device Communications using the Devices Profile for Web Services. In *Proc. of the 3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing*, pages 1–8, New York, NY, USA, 2005. ACM Press.

[6] Margaret G. Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets, and Martha Mercaldi. XML Screamer: an Integrated Approach to High Performance XML Parsing, Validation and Deserialization. In *Proc. of the 15th International Conference on World Wide Web*, pages 93–102, New York, NY, USA, 2006. ACM Press.

[7] Piyush Maheshwari, Hua Tang, and Roger Liang. Enhancing Web Services with Message-Oriented Middleware. In *Proc. of the IEEE International Conference on Web Services*, pages 524–531, San Diego, CA, USA, 2004. IEEE Computer Society.

[8] Steve Vinoski. Integration with Web Services. *IEEE Internet Computing*, 7(6):75–77, 2003.