Middleware for Communication and Deployment of Time Independent Mobile Web Services

Fahad Aijaz, Seyed Mohammad Adeli, Bernhard Walke
RWTH Aachen University, Faculty 6
Communication Networks
Kopernikusstr. 16, 52074 Aachen
{fah, ade}@comnets.rwth-aachen.de

Abstract

With the increasing storage capacity, processing power and battery capabilities, mobile devices are now able to providing services instead of just being service consumers. This paper introduces a middleware for time independent Mobile Web Services (Mob-WS) that avoids the overheads of long durational synchronous communication. Details of communication architecture and interaction among the middleware components is presented and discussed. The Bluetooth binding for SOAP has been developed and briefly presented.

1 Introduction

Mobile phones in today's era are equipped with more processing power, storage capacity and battery performance. This continuous growth in mobile technology has enabled hosting services on mobile devices that are termed as Mob-WS.

In this paper, a concept and architecture of a middleware for communication and deployment of asynchronous Mob-WS is introduced to perform long-lived operations. The proposed middleware supports variety of transport protocols. Here, we present details about the Bluetooth implementation. The work is based on the existing Mob-WS framework first presented in [2].

2 SOAP Binding for Bluetooth

Bluetooth is a wireless communication protocol mainly used for short range wireless connectivity between Bluetooth-capable devices. Since Bluetooth technology is a universal standard, it guarantees a high level of compatibility. In this work, SOAP messages over Bluetooth are used for service invocation, as well as monitoring and controlling of an existing service instance.

2.1 Bluetooth Discovery

Since wireless devices are mobile they need a mechnism that allows them to find accessible devices and gain access to their capabilities. Sun Wireless Toolkii 2.5 includes the optional package JSR 82 which contains the J2ME Bluetooth API. The `DiscoveryAgent` class and `DiscoveryListener` interface, which the `Observer` implements, provide the necessary service for discovery. The `Observer` of the client node creates an instance of `BTDiscovery` to look for any Bluetooth devices such as phones, PDAs or computers. It is essential to perform common Bluetooth operations like device discovery, in seperate threads in order to avoid potential deadlocks, since they may take several seconds to complete. After retrieving the UUID of the specified protocol (in our middleware L2CAP) from `BTProtocol`, it begins with the discovery of Bluetooth devices. The `DiscoveryAgent` is used to search for all accessible devices in the environment. The class `RemoteDevice` represents a device within the range of reach and provides methods to retrieve information about the device including it's Bluetooth address, name and available services. The method `deviceDiscovered()` is called everytime a new device is found, and `inquiryComplete()` when the discovery is completed. At the end, all discovered devices are sent back as a result to the `Observer`. It is also possible for the `Observer` to select a device from a list of pre-known devices, which the local device contacts frequently. In this case the device discovery would no longer be used.

2.2 Bluetooth Transport

Once a remote device is discovered on the client node, the `Observer` sends a SOAP request using `BluetoothTransport` class. This class represents a communication interface on client-side and provides method to facilitate SOAP calls over Bluetooth using the J2ME Generic Connection Framework. In order to establish a connection via Bluetooth the client makes use of the btspp protocol, which consists of a host name, retrieved at the device discovery, and a channel. The Bluetooth Serial Port Profile (btspp://) is a wireless interface specification for Bluetooth-based communication which emulates a serial cable and provides a simply implemented wireless replacement. Using the `StreamConnection`, it creates a new connection to a specific device. In order to start the transmission it opens an `OutputStream` and writes the SOAP message which is just a simple text.

2.2.1 SOAP Transmission

The `BluetoothServerThread` is the class responsible for Bluetooth connections on server-side in our prototype. It makes the server discoverable to other Bluetooth devices and waits for an incoming SOAP request from a client. The `LocalDevice` class represents the server and provides method to configure the local device. After the connection is established successfully, it is being forwarded to the `RequestHandler` to process the request.

Applications can use a single instance of BluetoothTransport for all SOAP calls to one, or multiple target endpoints.

3 Communication Architecture

The concept of Mob-WS involves two basic roles, termed as Web Service Consumer (WS-C) and Web Service Provider (WS-P). Within the context of this work, the focus is on Peer-to-Peer (P2P) interaction, where both the WS-C and WS-P are mobile nodes.

The asynchronous Mob-WS middleware provides a foundation to develop and deploy asynchronous long-lived services, not asynchronous messaging. This is because asynchronous messaging does not necessarily implies that the deployed services are asynchronous in nature. Since asynchronous services start and continue to perform their tasks over a longer duration of time, therefore, the need to be able to monitor and control such services becomes essential. This section presents the architectural details for inter-component communication within the proposed middleware, the service control and monitoring mechanisms and the service invocation process.

For a general overview of the core architecture of the asynchronous Mob-WS middleware, the reader is refered to study [1]. The architecture of the overall middleware consists of several components interlinked togather. Within this manuscript, the focus will be on discussing the asynchronous aspects of the architecture.

3.1 Communication Subsystem

In order to build a system of asynchronous Mob-WS three different types of endpoints are defined to cater three distinct roles that forms a subsystem within the middleware architecture. For detailed description of each, a study of [1] is recommended.

3.2 Inter-Component Control Flow

Having explained the basic notion of the three roles of asynchronous Mob-WS middleware in [1], a detailed control flow of communication among them is now presented in this section. These details focus only on asynchronous interaction and are elaborated in reference to figure 1, and its parts (A), (B), (C) and (D).

3.2.1 Service Invocation - (A)

Consider two mobile nodes with the asnychronous Mob-WS middleware deployed on them. Each node will have the three basic communication components described in 3.1. In order to consume asynchronous Mob-WS, the `Observer` of the client node (labeled (A) in figure 1) is required to send a SOAP request to its peer with the service specific information and its own Endpoint Reference (EPR) that should be used by the host node to communication with the client asynchronously. Once the desired SOAP envelope is constructed, it is then send over the network to the host peer. The protocol listeners (later refered as listeners) of the host peer constantly waits for incoming Mob-WS requests from the peer's `Observer` component. The client node could send this request over HTTP, User Datagram Protocol (UDP) or Bluetooth etc. since the Mob-WS invocation is independent of the transport protocol. Once the SOAP envelope arrives at the host node, it is delegated to the `RequestHandler` and the listeners start waiting for new incomming requests. The aim of `RequestHandler` is to determine the service type, that is, asynchronous or synchronous. Else, the `RequestHandler` passes the message to the `ASAPHandler` which could be seen as a gateway to the asynchronous Mob-WS consumption.

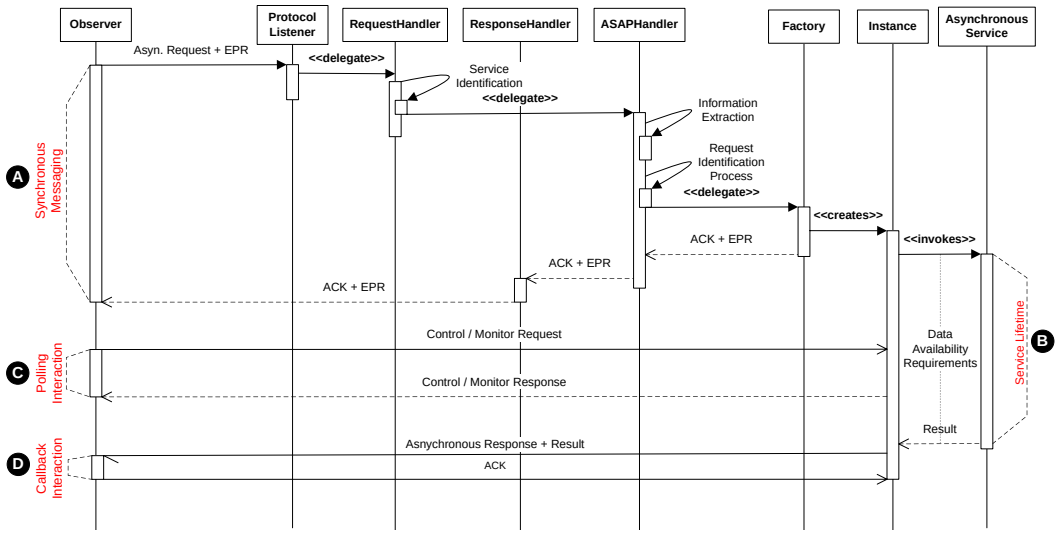

Figure 1. Asynchronous Communication Control Flow

At this point, the middleware has to identify the purpose of request since, it could either be directed to the `Factory` or `Instance` component. This is identified by extracting the information embedded within the SOAP envelope. Assuming the request for service consumtion, the middleware should first create an `Instance`, which in turn invokes the desired service. As stated earlier in 3.1, the `Factory` acts as a manager and is responsible for `Instance` creation, therefore, the request is delegated to the `Factory`'s `Instance` creation procedure. This is to note that `Instance` creation does not imply immediate service invocation, rather the `Instance` could be set to invoke the service at some later time which is decribed in SOAP envelope received from the client node. In this paper, we assume immediate service invocation to keep things as simple as possible.

Once the `Instance` is created with its unique EPR, it instantly invokes the asynchronous Mob-WS, whereas the `Factory` sends the Acknowledgment (ACK) to `ASAPHandler` along with the EPR of the `Instance` which is sent back to the client's `Observer` by the `ResponseHandler` over the preferred transport protocol.

This is vital to note that the entire communication between the peers decribed above is synchronous, but it invoked an asynchronous Mob-WS. Once the service is invoked, it keeps on performing its tasks for as longer duration as desired. The client is notified synchronously about the status of the service in order to avoid its blocked state whereas the service continues to perform its tasks at the server.

3.2.2 Asynchronous Service - (B)

One important aspect of the middleware architecture is the dependancy between the `Instance` and the asynchronous Mob-WS (later refered as service). The service is not directly exposed to the invocating peer's `Observer`, rather it is the `Instance`. Since requirements of every service is dependant on its usage enviornment, therefore, it is left to the application developer how to make the result of the service available to its clients.

For instance, in deadline critical applications, the peer's `Observer` might be interested in having partial results while the service is still running, whereas, in normal cases, there is no need to expose the result data of the service, unless it has completely performed its duties. This dependancy is labeled as '*Data Availability Requirements*' in figure 1.

3.2.3 Control and Monitoring - (C)

For an asynchronous long-lived Mob-WS it is essential to be controllable and provide mechanisms for monitoring its state. This is because during the execution of such services, the service consumer might be interested in receiving status updates or the requirements of the client might change. For instance, in a sensor based environment where the real-time information is constantly changing, an asynchronous Mob-WS client might want a service to utilize the updated context information that it has just received from its environment and discard the one that was previously sent at the time of service invocation. In order to meet such dynamic change in requirements, a client should be able to send

control messages to the service `Instance` and on the other hand, the service `Instance` should be capable of updating itself at runtime.

The asynchronous Mob-WS middleware incorporates the control and monitoring mechanisms within its architecture that makes the deployed services capable of adapting to dynamic changes. Once the service is invoked and running, the client's `Observer` can send control and monitoring messages in form of SOAP as illustrated in figure 1 (C). These messages are directly sent to the service `Instance`, since the client's `Observer` already contain its EPR that was received during service invocation process.

This is vital to note that such dynamic service management makes use of the polling interaction tecnique due to a short-termed nature of its messages. This implies that such communication is synchronous in nature.

Service Monitoring: Consider an example in which the client's `Observer` wishes to receive the properties of the service that is currently in execution phase. Once the status information is received, it can then analyze and act accordingly. The client might wish to control the service based on its current state. Such monitoring requests can be send as SOAP envelopes from the peer `Observer` directly to the service `Instance`. This request message once received by the `ASAPHandler` is analyzed and passed on to the `Instance`. For simplicity, we show a direct link to the `Instance` in figure 1 (C).

Service Control: In order to control a service, consider a scenario in which a service client wishes to change the state of a running service. Such control message could involve activities from setting various properties of the service to pausing or terminating the service. In such case, a control request as a SOAP envelope from the `Observer` is received by the `ASAPHandler` that describes which state the service should change to. An asynchronous Mob-WS at any given time could be in the states; *open.notrunning, open.notrunning.suspended, open.running, closed.completed, closed.abnormalCompleted or closed.abnormalCompleted.terminated*. After changing the state of the service `Instance`, a notification is sent to the `Observer` that triggered the control message sequence.

Any change in state of the service triggers a callback StateChanged message to be sent to all the clients that have subscribed to this particular asynchronous Mob-WS. A change in state could be caused by some internal event such as occurrence of an exception during processing, or could be triggered externally by a client.

3.2.4 Service Response - (D)

Upon completion of a long-lived Mob-WS, the `Instance` is responsible for sending the result back to the peer `Observer`(s). Since the `Instance` incorporates the `Observer`'s EPR, therefore, a callback interaction mechanism is adapted for communicating the service response as depicted in figure 1 (D). This is done by invoking a method at the `Observer` via SOAP. The SOAP response carries with it, the entire result information that the service has produced during its runtime.

Once the `Observer` at the calling peer receives final response, it acknowledges the `Instance`, and hence the `Instance` expires.

4 Conclusion

In this paper, a communication architecture of an asynchronous Mob-WS middleware is introduced that discusses the idea of deploying long-lived complex asynchronous Mob-WS on mobile devices. In addition to Bluetooth binding for SOAP, the service interaction techniques are also presented and briefly compared. The presented middleware enables control and monitoring of long-lived asynchronous Mob-WS and also reduces development cost.

5 Acknowledgement

This work is funded by the Ultra High-Speed Mobile Information and Communication (UMIC)[1] research cluster at RWTH Aachen University (RWTH) under the German Excellence Initiative. The authors would like to thank the members of the project for their contributions.

References

[1] F. Aijaz, B. Hameed, and B. Walke. Asynchronous mobile web services: Concept and architecture. In *Proceedings of the IEEE 8th International Conference on Computer andInformation Technology*, page 6, Sydney, Australia, July 2008. IEEE.

[2] L. Pham and G. Gehlen. Realization and Performance Analysis of a SOAP Server for Mobile Devices. In *Proceedings of the 11th European Wireless Confernce 2005*, volume 2, pages 791–797, Nicosia, Cyprus, Apr 2005. VDE Verlag.

[1] www.umic-aachen.de