

# Architecture for Consuming Long-Lived Mobile Web Services over Multiple Transport Protocols

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

**Abstract**—The concept of Mobile Web Services (Mob-WS) is catching pace within research communities due to rapid classification of mobile devices enabling pervasive environments. A Web Service based Middleware for mobile applications is a promising platform to accelerate the application development for mobile systems.

This paper introduces a middleware for time independent 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. Due to the transport neutral nature of SOAP, the Bluetooth initial short-ranged transport protocol binding is briefly presented. Finally, some preliminary processing latencies are shown followed by conclusion.

## I. INTRODUCTION

Mobile phones in today's era are not just small devices that provide the means of communication, rather, they are equipped with more processing power, storage capacity and battery performance. The continuous growth in mobile technology has introduced variety of research domains that strongly focus on developing innovative mobile applications and services that are human centric and closer to the user's personal needs. In such a dynamic and rapidly growing ubiquitous mobile environment, hand held devices are not only service consumers over the Internet, wireless network or within the operators' network, but are also capable of hosting and providing services to their peers. Such services, when hosted on mobile devices are often termed as Mob-WS.

In this paper, a concept and architecture of a middleware for communication of asynchronous Mob-WS is introduced to perform long-lived operations. The work is based on the existing Mob-WS framework first presented in [1]. Since synchronous Mob-WS are not a feasible choice for long durational tasks, therefore a need for asynchronously accessible Mob-WS becomes vital. These Mob-WS demand mechanisms for control and monitoring at runtime, since the requirements of service consumer may change dynamically. The proposed middleware supports the communication and deployment of such controllable and monitor able asynchronous Mob-WS by reducing the development costs. Some preliminary performance evaluations in terms of elementary processing latencies are presented.

## II. BLUETOOTH

Bluetooth is a wireless communication protocol mainly used for short range wireless connectivity between Bluetooth-capable devices. The most important advantage of Bluetooth is the fact, that it is a cable replacement technology and does not need any physical connection. Due to usage of low-power signals, Bluetooth works with a low power consumption, which makes it a popular protocol for interaction between mobile devices. Bluetooth signals are omni-directional and a direct line of site is not required for transmission. Since Bluetooth technology is a universal standard, it guarantees a high level of compatibility. There are three different protocols which provide methods for transmitting data: RFCOMM (Stream data), L2CAP (Packet Data) and OBEX (Object Data). The protocol used in this work is Logical Link Control and Adaptation Protocol (L2CAP).

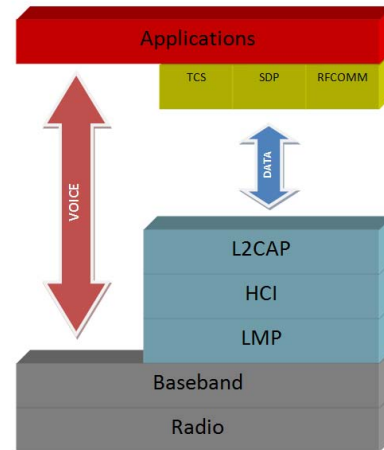


Fig. 1. Bluetooth Protocol Stack Diagram

## III. SOAP BINDING FOR BLUETOOTH

SOAP messages over Bluetooth are used for service invocation, as well as monitoring and controlling of an existing service instance.

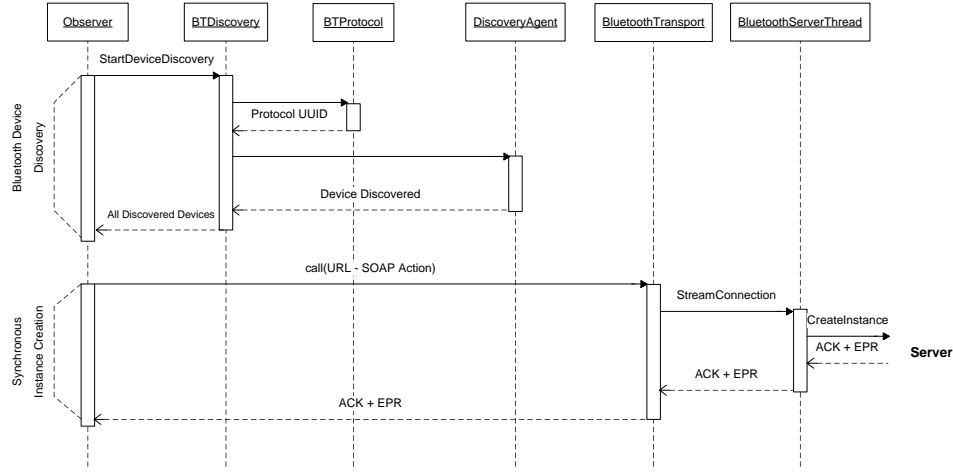


Fig. 2. Bluetooth Device Discovery Flow

### A. Bluetooth Discovery

Since wireless devices are mobile they need a mechanism that allows them to find accessible devices and gain access to their capabilities. Sun Wireless Toolkit 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, PDA's or computers. It is essential to perform common Bluetooth operations like device discovery, in separate 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 its 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.

### B. 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 dis-

covery, 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.

#### 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.

## IV. 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, as shown in 3.

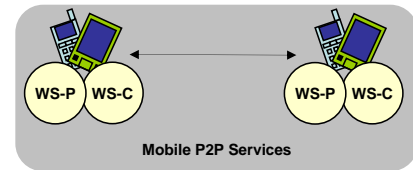


Fig. 3. P2P Mobile Web Services

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 referred to study [2], [3]. The architecture of the overall middleware consists of several components interlinked together. Within this manuscript, the focus will be on discussing the asynchronous aspects of the architecture. For details regarding other components and synchronous Mob-WS, another study of [1], [4], [5] is recommended.

#### A. 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 [2], [3]. For the reader's convenience, we summarize the entire communication architecture of the asynchronous middleware by presenting the following sequence of interaction among these components. For detailed description of each, a study of [2], [3] is recommended.

- 1) *Observer* sends a new service *Instance* creation message to *Factory* containing a context data along with its *Endpoint Reference (EPR)*.
- 2) *Factory* creates an *Instance*, having a corresponding unique *EPR*. The *EPR* of the *Observer* is also passed to this *Instance* for later notifications.
- 3) *Factory* sends a response message to the *Observer* that contains the *EPR* of the service *Instance* that has just been created.
- 4) When execution of the service ends, the *Instance* sends a notification message to the *Observer* containing the result data that is produced as a result of successful task completion.
- 5) The *Observer* sends an acknowledgment to the service *Instance* indicating that it has received the result data. Upon reception of the acknowledgment, the service instance expires.

#### B. Inter-Component Control Flow

Having explained the basic notion of the three roles of asynchronous Mob-WS middleware, a detailed control flow of communication among them is now presented in this section. These details are elaborated in reference to figure 4, and its parts (A), (B), (C) and (D).

1) *Service Invocation - (A)*: Consider two mobile nodes with the asynchronous Mob-WS middleware deployed on them. Each node will have the three basic communication components described in IV-A. In order to consume asynchronous Mob-WS, the *Observer* of the client node (labeled

(A) in figure 4) is required to send a SOAP request to its peer with the service specific information and its own *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 referred 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 incoming requests. The aim of *RequestHandler* is to determine the service type, that is, asynchronous or synchronous. In case the requested message is directed toward a synchronous service, the control flow continues as described in [4]. Else, the *RequestHandler* passes the message to the *ASAPHandler* which could be seen as a gateway to the asynchronous Mob-WS consumption.

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 consumption, the middleware should first create an *Instance*, which in turn invokes the desired service. As stated earlier in IV-A, 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 described 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 described 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.

2) *Asynchronous Service - (B)*: One important aspect of the middleware architecture is the dependancy between the *Instance* and the asynchronous Mob-WS (later referred as service). The service is not directly exposed to the invoking peer's *Observer*, rather it is the *Instance*. Since requirements of every service is dependant on its usage environment, 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

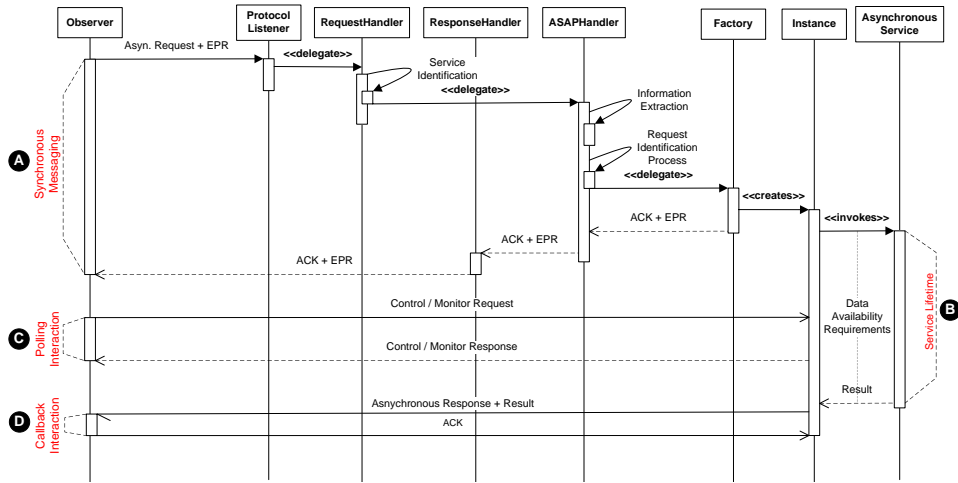


Fig. 4. Asynchronous Communication Control Flow

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 dependency is labeled as 'Data Availability Requirements' in figure 4.

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 4 (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 technique due to a short-termed nature of its messages. This implies that such communication is synchronous in nature.

a) *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 4 (C). The monitoring SOAP request looks like as shown in figure 5.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  xmlns:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing">
  <SOAP-ENV:Header>
    <!-- WS-Addressing request headers -->
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <computePerf xmlns="urn:Services" id="o0"
      SOAP-ENC:root="1">
      <getPropertiesRq xsi:type="getPropertiesRq"/>
    </computePerf>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 5. Service Monitoring Request

b) *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 [6]; *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. Figure 6 shows a sample of such control message. The response SOAP message follows exactly the same structure except that the message name is changed to `changeStateRs`.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  xmlns:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing">
  <SOAP-ENV:Header>
    <!-- WS-Addressing request headers -->
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <computePerf xmlns="urn:Services" id="o0"
      SOAP-ENC:root="1">
      <changeStateRq xsi:type="changeStateRq">
        <State xsi:type="xsd:string">
          closed.completed
        </State>
      </changeStateRq>
    </computePerf>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 6. Service Control Request

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 4 (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.

## V. PROCESSING LATENCIES

Results as depicted in figure 7 show that the major processing cost takes place during the protocol and service type identification and response process. This is because the underlying protocol is identified either by parsing Web Service Addressing headers in SOAP message request (e.g. in case of UDP [5], [4]) or by checking for HTTP headers. In the current implementation, Bluetooth transport binding also exists, and therefore, a Bluetooth request is also identified by SOAP parsing that adds to the cost. In order to provide the flexibility of multiple transport protocols for service consumption, each request has to under go such protocol identification process since the SOAP message could change accordingly [5], [4], [1]. The service type means synchronous or asynchronous request. In case of synchronous requests, the processing becomes relatively simple when compared to asynchronous due to difference in SOAP message sizes, their structures and complexity of architectures.

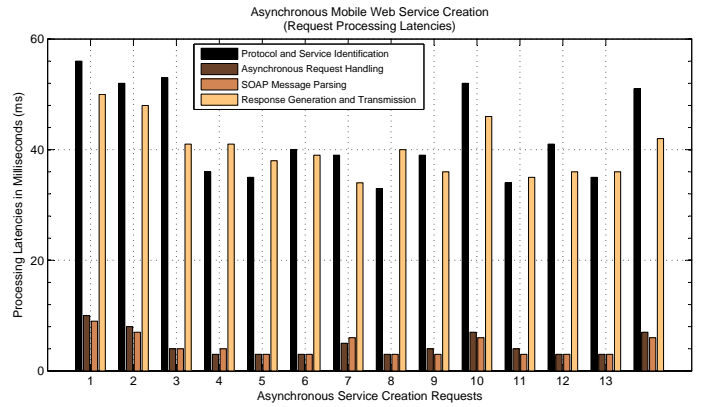


Fig. 7. Asynchronous Service Creation Processing Latencies

## VI. 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. Furthermore, some elementary performance measurements are shown. The presented middleware enables control and monitoring of long-lived asynchronous Mob-WS and therefore forms potential basis for innovative mobile applications and reduces development cost.

## VII. ACKNOWLEDGEMENT

This work is funded by the Ultra High-Speed Mobile Information and Communication (UMIC)<sup>1</sup> 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] L. Pham and G. Gehlen, "Realization and Performance Analysis of a SOAP Server for Mobile Devices," in *Proceedings of the 11th European Wireless Conference 2005*, vol. 2. Nicosia, Cyprus: VDE Verlag, Apr 2005, pp. 791–797. [Online]. Available: <http://www.comnets.rwth-aachen.de>
- [2] F. Aijaz, B. Hameed, and B. Walke, "Asynchronous mobile web services: Concept and architecture," in *Proceedings of the IEEE 8th International Conference on Computer and Information Technology*. Sydney, Australia: IEEE, July 2008, p. 6. [Online]. Available: <http://www.comnets.rwth-aachen.de>
- [3] —, "Towards peer-to-peer long lived mobile web services," in *Proceedings of the 4th International Conference on Innovations in Information Technology*. Dubai, UAE: IEEE, Nov 2007, p. 5. [Online]. Available: <http://www.comnets.rwth-aachen.de>
- [4] G. Gehlen, F. Aijaz, and B. Walke, "An enhanced udp soap-binding for a mobile web service based middleware," in *Proceedings of IST Mobile Summit 06*. Mykonos, Greece: ComNets, Faculty 6, RWTH Aachen University, Germany, Jun 2006, p. 8. [Online]. Available: <http://www.comnets.rwth-aachen.de>
- [5] W. B. Gehlen G, Aijaz F, "Mobile web service communication over udp," in *Proceedings of the 64th IEEE Vehicular Technology Conference*, IEEE. Montral, Canada: IEEE, Sep 2006, p. 1. [Online]. Available: <http://www.comnets.rwth-aachen.de>
- [6] J. F. et al., "Oasis asynchronous service access protocol (asap)." [Online]. Available: [http://www.oasis-open.org/committees/tc\\_home.php?wg\\_abbrev=asap](http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=asap)

<sup>1</sup>[www.umic-aachen.de](http://www.umic-aachen.de)