TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science, Institute for System Architecture, Chair for Computer Networks

DIPLOMA THESIS

Design and Implementation of a Web Gateway for Mobile Collaboration Services

submitted by

Nikolas Jansen

born August 6, 1982 in Bocholt

Professor: Prof. Dr. rer. nat. habil. Dr. h. c. A. Schill
Supervisor: Dr. Ing. Daniel Schuster

Submitted July 14, 2011

Technische Universität Dresden

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik, Institut für Systemarchitektur, Professur Rechnernetze

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name, Vorname:	Jansen, Nikolas	
Studiengang:	Medieninformatik	Matr.-Nr.: 2928402
Thema:	**Entwicklung eines Web-Gateways für mobile Kollaborationsdienste**	

ZIELSTELLUNG

Mit der Verbreitung leistungsstarker Smartphones entstehen neue Möglichkeiten für Kollaborationsdienste, die direkte Interaktion zwischen Personen auf vielfältige Weise unterstützen. Im Projekt Mobilis sind dazu eine Reihe von Anwendungen wie MobilisXHunt, MobilisBuddy oder MobilisTrader entstanden, die kollaborative Funktionen auf Android-Geräten innerhalb einer auf XMPP basierten Client-Server-Architektur bereitstellen.
Ziel der Arbeit ist es, neben dem Zugriff aus speziell für die Android-Plattform entwickelten Apps auch Web-basierten Zugriff auf die Dienste der Mobilis-Plattform von beliebigen Endgeräten zu gestatten. Dazu soll zunächst der Fall eines Standard-Web-Zugriffs (z.B. mit einem Netbook) untersucht werden und im Rahmen der Möglichkeiten der Arbeit auch auf Webzugriff mit eingeschränkten Darstellungsmöglichkeiten (z.B. auf dem iPhone) eingegangen werden. Es soll vor allem untersucht werden, wie die Echtzeitkommunikation zwischen Client und Web-Server (z.B. mittels Ajax oder Comet) und zwischen Web-Server und Mobilis-Server (XMPP) geeignet realisiert werden kann. Es soll ein generisches Framework für die Einbindung von Web-Clients in Mobilis-Anwendungen entwickelt und anhand von mindestens zwei existierenden Mobilis-Diensten validiert werden.

SCHWERPUNKTE

- Grundlagen: Web-Interaktionsframeworks (Ajax, Comet, …), XMPP, eCollaboration auf mobilen Geräten, Web-basiertes eCollaboration
- Konzeption eines Web-Gateway-Frameworks
- Implementierung des Frameworks
- Implementierung der Web-Clients für zwei selbst gewählte Mobilis-Anwendungen
- Evaluation des Frameworks

Betreuer:	Dr.-Ing. Daniel Schuster
Betreuender Hochschullehrer:	Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill
Beginn am:	15.01.2011
Einzureichen am:	14.07.2011

Unterschrift des betreuenden Hochschullehrers

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Ich erkläre, dass ich die vorliegende Arbeit eigenständig und ausschließlich unter Verwendung der im Quellenverzeichnis aufgeführten Literatur- und sonstigen Informationsquellen angefertigt habe.

Dresden, July 14, 2011

Abstract

The Mobilis Project introduces a service-based middleware platform to support development of mobile collaboration applications. Currently, the client applications for this are restricted to the Android platform severely limiting the reach of customers.

In this work we discuss the web as an additional platform for client applications in order to unlock more mobile platforms and devices for the *Mobilis* platform. The real-time characteristics of a collaboration application impose serious challenges when introducing the web browser and its limitations. The protocols that power the web are not designed for the necessary bi-directional real-time message exchange needed for the collaboration experience.

We investigate several technologies in regard to their capacity for facing these challenges. We identify the most appropriate solution and build a web gateway for mobile collaboration services on top of it. Our effort is dedicated to supporting web application development to produce web-based clients for the *Mobilis Project* that are in no way inferior to Android-based clients.

Contents

1. Introduction

This chapter illustrates the area of interest, motivates the topic, introduces the contribution of this thesis and describes the structure of the work.

1.1. The Overall Picture

A few years ago, most mobile devices were very limited in their capabilities. Early generations of smartphones were either entirely email focused or lacked sophisticated touch screen capabilities. Mobile web browsers were restricted to displaying simple text, links and occasionally images. The next generation of smartphones, pioneered by Apple, Inc[1] with the iPhone, introduced great changes for the mobile world: WiFi and GPS support, 3G and Long Term Evolution (LTE), powerful processing capabilities, desktop-like web browsers and streaming video. With the introduction of powerful mobile devices, consumer expectations for performance and user interfaces (UI) drastically changed. The exponential growth of mobile technology has seen private and public organizations become increasingly more savvy of the importance of providing content and services to consumers on mobile devices. The two fastest growing applications on mobile devices are location-based services and mobile social networks [PG11].

The term Mobile Social Software describes applications for mobile devices which allow and support social interaction between users. Real-time information delivery is fast emerging as one of the most important elements of the social online experience. As one of the first companies to adopt it, Research In Motion (RIM)[2] is quite famous for their BlackBerry Enterprise Server which is dedicated to pushing emails to client devices. A lot of different techniques and protocols are available, but most of them are closed and proprietary. The *eXtensible Messaging and Presence Protocol* (XMPP) brings real-time communication and rich presence mechanisms into the mobile world in an open and extensible way.

Mobile collaboration, as a form of Mobile Social Software, relies heavily on real-time messaging capabilities. It augments social networks with real-time information, such as location or collaborative messages. A comprehensive environment is provided in which users can interact in a target-oriented fashion to reach common goals.

[1] http://www.apple.com/
[2] http://www.rim.com/

The *Mobilis Project* introduces a service oriented platform to support developers of mobile collaboration applications. According to a survey performed by Springer et al. [SSB+08], more than 80% of the top 50 proposals of Google's Android Developers Challenge used at least one collaboration feature while 30% utilize at least three collaboration features. The *Mobilis* platform provides a set of collaboration features as reusable services for efficient application development. As of now, only client applications native to the Android Platform[3] are supported in the *Mobilis Project*.

Many mobile services suffer restrictions to one specific platform. The biggest challenge for mobile application development is the fragmentation across devices, mobile operating systems and their native application ecosystems. Building a different version of a mobile application for each platform is expensive if written in each native language. Required skills include C and Objective-C for iOS development, Java for Android and BlackBerry development, C and C++ for Symbian development and .NET for Windows Phone development.

The Tablet market suffers the same fragmentation among platforms. Four recently released tablet devices are powered by four different operation systems.

Tablet	Apple iPad	Samsung Galaxy Tab	RIM PlayBook	HP TouchPad
Platform	**iOS**	**Android**	**Tablet OS**	**webOS**

Table 1.1.: Fragmentation of Tablet Platforms

Despite the huge diversity of runtimes, the predominate focus in todays mobile world is to develop native applications that run on a particular platform. The future of mobile platforms is unclear in view of the fact that none of the competitors have emerged as a dominant force. With this trend in mind, platform independent mobile development seems very attractive. Web-based mobile application development provides the desired independence and enjoys increasing popularity these days.

With the newest web standards and powerful Javascript engines built-in, mobile web browsers provide a platform for feature rich applications. HTML5 [Hic11] brings richer semantics, 3D graphics and effects, native access to video and audio, better device integration, the ability to store data and run applications offline and enhanced scalability via utilization of multiple CPUs. The *World Wide Web Consortium* (W3C) Device API Working Group[4] aims to standardize the support for device specific *Application Programming Interface*s (APIs) that provide functionality so far reserved for native code, such as access to the local file system. Some mobile browsers already support access to: Geolocation[RCBH10], camera, microphone and even screen orientation and device motion.

Web technologies are just starting to bring native device functionalities to the

[3] http://www.android.com/
[4] http://www.w3.org/2009/dap/

browser, and the web has proven to be a serious alternative as a runtime for mobile social software. It even has some advantages not related to the development process or platform fragmentation. The web makes it a lot easier to distribute and monetize mobile applications. Web focused companies like Google, Inc[5] benefit from the reach their products have. Google+[Goo11b] and Google Docs[Goo11a] run on any device with a modern browser. iOS applications run only on Apple-made devices.

In April 2011, 39.1% of U.S. mobile subscribers used the browser on their mobile device, while downloaded applications were used by just 37.8% [com11]. That means even now the web is more popular than native applications. Providing a mobile web application to connect to your social network or your mobile collaboration services is a smart business decision.

Bottom line is: web application development provides a platform independent runtime for sophisticated client applications which are in no way inferior to native applications.

The capabilities of modern web technologies in combination with the numerous advantages just mentioned form mobile web applications into a promising route to take in order to unlock more platforms and extend the reach. The objective of this work is to design and implement a development framework for mobile collaboration web applications.

1.2. Contribution

The previous section provided the motivation for the decision to extend the *Mobilis Project* from Android exclusive client to platform independent web-based clients. In this section we describe the contribution of this work to support that decision. Web development faces the same challenges as desktop software development. Frameworks support the development process by providing libraries that implement commonly used functionality.

The contribution of this work is a web gateway framework for *Mobilis* applications. We investigate various web technologies to built a system that accommodates the real-time characteristic of collaboration applications. In addition to the communication aspect, we provide libraries for interacting with the services of the *Mobilis* platform. We enable developers to easily built web-based clients by reusing the collaboration services in the Mobilis Service Environment.

[5] www.google.com

1.3. Outline

The remainder of this work is organized as follows. Chapter 2 presents fundamental knowledge about real-time messaging and web technologies. We also conduct an analysis of the state-of-the-art in web-based collaboration systems. Chapter 3 illustrates the results of a comprehensive requirement analysis including functional and non-functional requirements. In Chapter 4 we present the conceptual design of a framework providing a web gateway for mobile collaboration services. Chapter 5 covers selected details of the implementation of concept. Both concept and implementation are evaluated in Chapter 6, while Chapter 7 sums up the results of this work and provides an outlook to web-based real-time collaboration.

2. Fundamentals

This chapter provides fundamental knowledge necessary to follow the conceptual design and implementation presented in this work. We begin in Section 2.1 with an introduction of the *eXtensible Messaging and Presence Protocol*. In Section 2.2 we discuss various web interaction models and technologies to achieve asynchronous web communication. Section 2.3 provides an overview of the Mobilis Service Environment, an extensible service platform for mobile collaboration applications. Section 2.4 wraps this chapter up with an in depth analysis of the state-of-the-art in research and industry.

2.1. Extensible Messaging and Presence Protocol

The *eXtensible Messaging and Presence Protocol* (XMPP) is, at its most basic level, a protocol for moving small, structured pieces of data between two network endpoints in near real time. This technology has been used to develop large-scale real-time instant messaging systems, collaboration spaces, and voice and video conferencing systems. The *Request for Comments* (RFC) 3920 [SA04a] and RFC 3921 [SA04b] define the core of XMPP as an implementation of the Model for Presence and Instant Messaging (IM) specified by the *Internet Engineering Task Force* (IETF)[1] in RFC 2778 [DRS00]. On top of the core specification, the *XMPP Standards Foundation* (XSF) elaborated a set of over 200 *XMPP Extension Protocol*s (XEPs). In Section 2.1.3 we cover the XEPs relevant to this work.

Jeremy Miller invented XMPP technologies under the name *Jabber* in 1998 as an interoperable open source instant messaging protocol. The developer community grew fast and support for multiple platforms and languages was achieved shortly. By 2001 the Jabber Software Foundation was formed to document the core protocol and define extensions to the core in order to keep *Jabber* open source. The Jabber Software Foundation was renamed in 2007 to the *XMPP Standards Foundation*. The XSF[2] decided to seek a wider review of the core protocols by formalizing them within the IETF. After two years of intensive work, the IETF published the core XMPP specifications in RFC 3920 [SA04a] and RFC 3921 [SA04b]. XMPP is today

[1] http://www.ietf.org/
[2] http://www.xmpp.org/

an open standard that defines the protocols and data formats that power real-time interactions over the Internet.

2.1.1. Architecture

XMPP uses a decentralized client-server architecture with multiple interconnected servers. Although the specification does not tie the protocol to any specific network architecture, in most real-life implementations the client establishes a long-lived *Transport Control Protocol* (TCP) connection to a server. Servers also communicate with each other over TCP connections. One design goal of XMPP is the interoperability with other messaging protocols that comply with the IETF IM Model. Figure 2.1 shows the basic architecture of XMPP network. Much like email, the client only communicates with its dedicated server and the federated network of interconnected servers delivers the information to the recipient with server-to-server communication. The server of the sending client directly forwards the information to the target server in a one-hop way unlike email where multiple hops between different servers for a single message transfer are common.

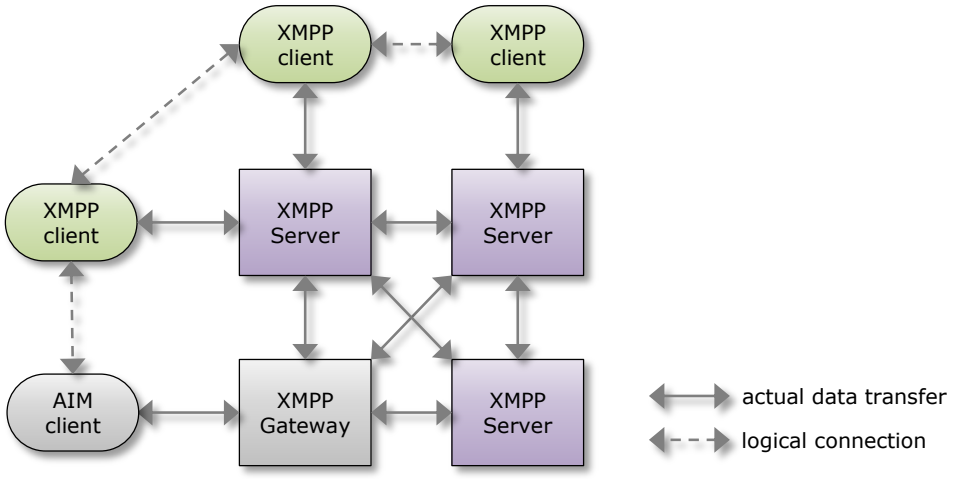

Figure 2.1.: XMPP Architecture

Addressing Scheme

The addressing scheme of XMPP is similar to the email addressing scheme. Every entity in the network has a *Jabber Identifier* or JID. A JID is composed of three ordered elements, the node identifier, the domain identifier and the resource identifier. The syntax of a JID is:

```
[ node "@" ] domain [ "/" resource ]
```

XMPP utilizes the *Domain Name System* (DNS) in order to provide a familiar address structure and avoid raw *Internet Protocol* (IP) addresses. The domain identifier is a resolvable DNS name of the entity and is mandatory while the node and the resource identifier are optional. In the common case of using the JID as an Instant Inbox Address[DRS00] a user has to register with an XMPP service such as `jabber.org`. The user gets a so-called *bare JID* which contains the node identifier and the domain identifier. Every user may use this *bare JID* to open up multiple connections to the server, with each connection distinguished by attaching a resource identifier to the *bare JID* (forming the *full JID*). In the IM use case, this mechanism allows the user to log-in with multiple chat clients or from multiple locations simultaneously. `Juliet@jabber.org/home` is an example of a *full JID*.

Security

The XMPP core specification describes extensive security concepts. *Transport Layer Security* (TLS) [DR08] and *Simple Authentication and Security Layer* (SASL) [MZ06] are employed to achieve confidentiality, data integrity and mutual authentication. XMPP provides a TLS profile to encrypt client-to-server communication. During connection negotiation the XMPP server communicates the instituted security policy to the client with either a `<required/>` or `<optional/>` tag followed by the usual TLS handshake. After the handshake successfully completes, the client restarts the XMPP session, only now encrypting everything it sends using the key received from the server during the TLS handshake. In order to provide mutual authentication, SASL enables application developers and service administrators to support many different authentication mechanisms, such as PLAIN, DIGEST-MD5, SCRAM, EXTERNAL, GSSAPI and ANONYMOUS. Server-to-server communications basically follows the same pattern to achieve confidentiality and integrity. In this case the EXTERNAL authentication mode is used to ensure inter-domain federation. Due to mandatory security policies during server-to-server communication, it is not possible in XMPP for a server to forge client identities.

2.1.2. Streaming XML

The base format for exchanging information in XMPP is XML. Two fundamental concepts facilitate asynchronous bidirectional exchange of structured pieces of data, XML streams and XML stanzas. Communication is accomplished by streaming XML, which means exchanging XML stanzas over XML streams. In order to stream XML, two entities need to establish a long-lived connection (e.g TCP) and negotiate two XML streams (one for each direction of communication). Negotiation in this context means to follow security protocols for authentication and encryption.

From a document-centric point of view, an XML stream is an XML document that

is built continuously throughout the communication session. The root element of an XML stream is the XML `<stream>` tag. XML stanzas are well-formed pieces of XML and represent the basic unit of communication, much like packet or messages in other network protocols. XML stanzas can contain a payload represented as nested substructures. In essence, an XML stream functions as an envelope for all XML stanzas exchanged during a communication session.

Three kinds of XML stanzas are defined in the XMPP core specification which we characterize in the following sections: Message, Presence and Info/Query (IQ).

Message Stanza

The message stanza is represented by placing the `<message/>` tag as a first order child in the XML stream and is regarded as the basic push method of XMPP for sending information from one entity to another. In the familiar IM scenario the message stanza is used to transfer chat data, encapsulating the text message as the payload. Other applications are event delivery, alerts and notifications. Message stanzas come in different types (e.g `normal`, `chat` or `error`) This determine how the stanzas will be processing by the receiving entity. In addition to the `type` attribute, the message stanza contains a `from` and `to` address and an `id` attribute for tracking purposes. The core XMPP specifications define very basic payload elements, such as `<body/>` or `<subject/>`. Many XEPs utilize the capabilities of the `<message/>` stanza and define special purpose payload elements. Listing 2.1 shows a basic chat message stanza.

```
<message from="juliet@verona.lit/balcony"
         to="romeo@verona.it"
         type="chat"
         id="msg1">
  <subject>The Balcony Scene</subject>
  <body>Wherefore art thou, Romeo?</body>
</message>
```

Listing 2.1: Example for a message stanza

Presence Stanza

The presence stanza (`<presence/>`) is the basic broadcast or publish-subscribe mechanism. It is used to control and report the availability of an entity, such as *online* or *do not disturb*. In addition, `<presence/>` is used to establish and terminate presence subscriptions to other entities. The subscription itself is maintained through entries in the roster, which represents the presence-enabled contact list of a particular entity. A presence stanza with no `type` attribute indicate that the entity is available. If present, the `type` attribute either means lack of availability or indicates the different

stages of the subscription, such as `subscribed` or `unsubscribed`. Listing 2.2 illustrates three different presence stanzas. The first two stanzas broadcast availability status, whereas the third stanza indicates that the user `romeo@book.it` wants to subscribe to availability broadcasts published by `juliet@book.lit/home`.

```
<presence>
  <show>away</show>
  <status>At home</status>
</presence>

<presence type="unavailable"/>

<presence from="romeo@verona.lit" to="juliet@verona.lit/balcony" type="
    subscribe"/>
```

Listing 2.2: Examples for presence stanzas

Presence subscriptions are not automatically bidirectional. In terms of the IETF Model for Presence and Instant Messaging [DRS00], the user `romeo@book.it` is the *Presentity* and the user `juliet@book.lit/home` is the *Subscriber* form of a *Watcher* and the exchange of presence stanzas functions as the *Presence Protocol*.

Info/Query Stanza

Info/Query, or IQ, is a request-response mechanism similar to the *Hypertext Transfer Protocol* (HTTP) GET and POST/PUT semantics. With IQ stanzas an entity is able to make a request of, and receive a response from, another entity. Info/Query interactions are tracked by the requesting entity with the mandatory `id` attribute as a reference. Common patterns (Figure 2.2) of structured data exchange such as get/result or set/result (error responses are also possible) are followed. This leaves four values for the mandatory `type` attribute. *Get*, *set*, *response* and *error*.

IQ stanzas contain one single piece of payload represented as a well-formed XML snippet for the child element of the `<iq/>` tag. This request/response mechanism is useful in any case where result data or simple acknowledgment is required. In combination with the message stanza, the IQ stanza form the building blocks for most XEPs. The `xmlns` attribute of the payload child element determines which XEP is in use for a particular IQ stanza.

2.1.3. XMPP Extension Protocols

As mentioned earlier, data exchange in XMPP occurs in XML which gives the communication an extensible structure. It is easy to add new features to the protocol that are both backward and forward compatible. This extensibility is put to great use in

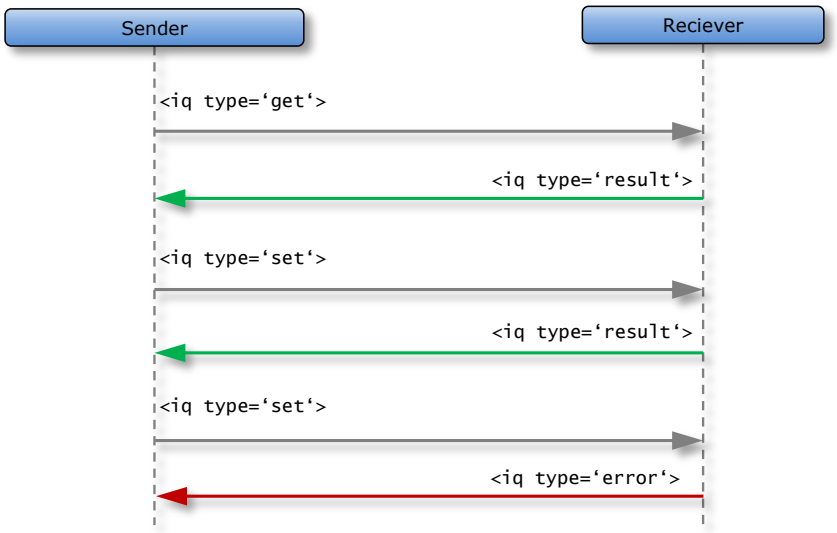

Figure 2.2.: IQ Interaction

the more than 200 XEPs registered with the XSF. XEPs are based on the aforementioned building blocks of XMPP communication, `<message/>`,`<presence/>` and `<iq/>` stanzas. XEPs work though extensive use of XML Namespaces as a way to scope XML stanza payloads. Extensions are matched on both the element name and the namespace.

In the following paragraph we describe the XEPs used in the *Mobilis Project* and also in the conceptual design for the web gateway for mobile collaboration services.

Data Forms (XEP-0004) The Data Forms extension [EHM+07] allows XMPP applications to define a form with fields of various types. In addition, a simple workflow mechanism on top of these forms allows applications to request, provide, submit and cancel forms. Data Forms are widely used by other XEPs as a generic data description format for dynamic form generation and data modeling.

Service Discovery (XEP-0030) The Service Discovery extension [HMESA08] offers mechanisms to discover information about entities in the XMPP network. Three distinct pieces of information are available for each entity: the basic identity, the features it offers and the protocols it supports. In order to discover information, the requesting entity sends an IQ stanza of type `get` containing an empty `<query/>` element qualified by the `http://jabber.org/protocol/disco#info` namespace to the JID of the target entity. The target entity then responses with a matching result IQ stanza containing the same `<query/>` element. All information about the target entity is included in the response stanza as child elements of the `<query/>` element.

Multi User Chat (XEP-0045) Unlike instant messaging, which is one-to-one communication, XMPP provides channels or rooms for multi-party interaction. This is similar to *Internet Relay Chat* (IRC). The XEP-0045 defines the namespace and necessary payload elements to provide *Multi User Chat* (MUC). The basic idea behind MUC is that participants can join a room and send messages that are delivered to all other participants. Rooms have their own JID. Participants send presence stanzas addressed to the room JID with the desired nickname as the resource element of the JID. After a participant joins a room, all other participants receive a notification about the new chat partner and his presence, which is followed by the room sending a notification back to the new user about all room occupants. All the aforementioned notifications are implemented as `<presence/>` stanzas. In order to exchange messages, participants send message stanzas addressed to the room, which are then reflected as a message stanza from the JID of the room with the original senders nickname attached as the resource. These reflected stanzas are sent to every participant in the room. The messages sent in a room are of type `groupchat`.

Publish Subscribe (XEP-0060) The XMPP publish subscribe extension provides a framework for arbitrary event notifications in real-time. This XEP eliminates the necessity for polling for updates on a regular basis. It implements the observer design pattern, where an entity publishes an update and all entities that have subscribed to receive updates are notified automatically in real-time. In this context real-time means that events are broadcasted to all authorized subscribers as soon as the event is triggered. The XEP-0060 defines this as a service that is able to receive and process publish requests, distribute event notifications and maintain subscriber lists. The central component in this publish-subscriber system is the *node*, to which the publisher sends their updates and from which the subscribers receive the event notifications.

2.2. Asynchronous Web Communication

As a second topic important to our work, we discuss in this section asynchronous web communication. We begin with a definition of interaction models in the World Wide Web to determine the features of asynchronous web communication. In order to understand why this form of interaction on the web is a challenge, we take a look at the architecture of the web. We then proceed to a detailed overview of approaches to achieve asynchronous web communication.

2.2.1. Web Interaction Models

Web interaction is the communication between the user, the client application (web browser) and the server application (web server). Two basic communication steps are considered. The first step occurs between the user and the client application, and the second step happens between the client application and the server application. Three basic interaction models exist: Synchronous, non-blocking and asynchronous.

Synchronous Interaction Model

This interaction model follows a strict request-response pattern, which means that the client sends a request for a resource and waits until the response is received. While the user waits for the response to arrive, all interactions with the client application are blocked, defining this interaction model as synchronous with respect to user actions. This model does not feature any mechanisms for the server to initiate communication with the client. The degree of dependency between request and response is very high. The server only sends data that is requested by the client. Server-side processing and user activity are completely coupled. Figure 2.3 illustrates synchronous interaction between the user, client and server application.

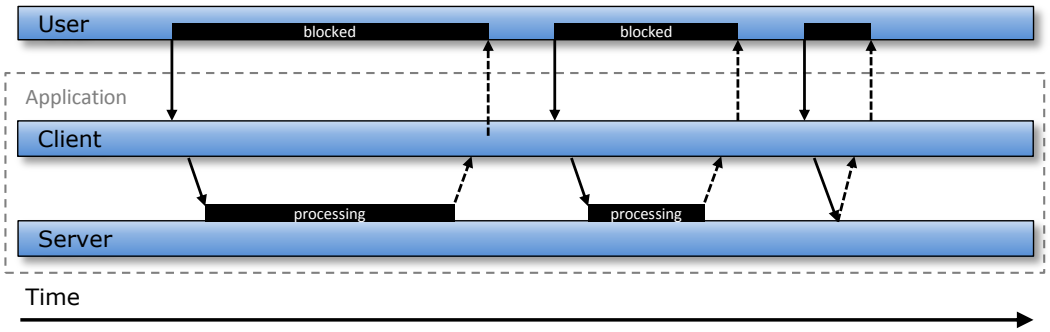

Figure 2.3.: Synchronous Interaction Model

Non-blocking Interaction Model

In the non-blocking interaction model an additional component within the client application is used to decouple the user action from data retrieval. A special object is used to send requests in the background without the user taking notice of this process. Once the client has received the response, the user interface is dynamically rendered incorporating the newly available data. This way, user action is not blocked while data is being retrieved. The user experiences a much more interactive and dynamic web application. Every piece of data sent by the server is still requested

by the client, indicating a high degree of dependency between request and response as illustrated in Figure 2.4.

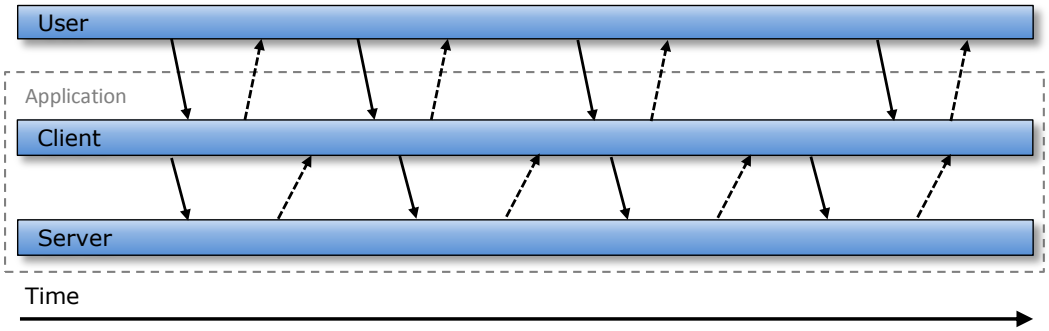

Figure 2.4.: Non-blocking Interaction Model

Asynchronous Interaction Model

Asynchronous interaction means that each communication participant can send data independent from one another. User actions are not blocked and completely decoupled from any data communication between client and server. This model enables the server application to send data to the client without the need for the client to request the data. Request and response are almost free of any dependency. However, an initial request from the client to the server to signal communication interest is still necessary. Figure 2.5 shows the asynchronous interaction model.

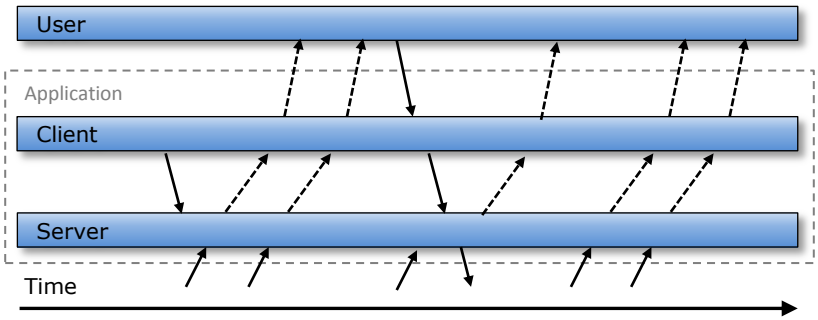

Figure 2.5.: Asynchronous Interaction Model

2.2.2. Architecture of the World Wide Web

The Internet is a TCP/IP based network using HTTP as the transportation protocol in its most popular interface, the World Wide Web. HTTP is an open, standardized

network protocol that enables clients to easily access files stored on servers. The main design goal of HTTP is to minimize latency and network communication on the one hand, and to maximize scalability and independence of the components on the other hand. This also provides the base for a popular architecture for distributed system design called REST.

In REST, all interactions for obtaining a resource representation are performed by synchronous request/response messages over HTTP. In this scheme, each interaction between the client and the server is independent of any other interactions. No permanent connection is established and the server maintains no state information about the client. In other words, the concepts of REST provide the scalable architecture for pull-based systems based on HTTP.

In HTTP, there are two different roles: server and client. In general, the client always initiates the conversation and the server replies. The server cannot transmit any data to the browser on its own, following the synchronous interaction model. HTTP messages are made of a header and a body. The body can remain empty. It usually contains the data to transmit over the network. The header contains metadata, such as encoding information but, in the case of a request, it also contains the important HTTP methods.

The initial architecture of the Internet allowed updates to object state to happen only on page reload. Thus, if two people worked on the same document, the first person could see the changes of the second person only after clicking the refresh button in the browser. Initial workarounds, such as polling, were unsophisticated but over the years, the situation has dramatically improved. The web has evolved from delivering web pages that are static documents, to pages that are enriched with dynamic content and now to highly interactive browser-based rich internet applications. This evolution of web content also required an evolution in the interaction capabilities of the web. Several solutions are used in practice that allow the server to propagate state changes asynchronously. Next we analyze the most important approaches.

2.2.3. Real-Time Web Techniques

Real-time web relies on the ability of a web server to provide information in a push-based style [HJ99], broadcasting state changes to the client asynchronously every time its state changes. In this model, a consumer subscribes to a channel and receives any information that is sent on the channel. A producer sends the data to the broadcaster with rules on how and where to distribute the data.

HTTP has most often been used as a request/response protocol, leading to clients polling for new data, or users hitting the refresh button in their browsers. Ajax provides a mild salve to the HTTP communication model by enabling clients to

poll for server-side events in a non-blocking fashion. By polling, server events can be queued and delivered to the browser at each pre-determined poll interval. This emulates server initiated communications and provides real-time message delivery within the bounds of the poll interval. Attempting to simulate server initiated communications with Ajax requires polling schemes, blindly checking for updates irrespective of state changes in the application. The result of this technique is poor resource utilization on both the client and server, since CPU-cycles and memory are needlessly allocated to prematurely or belatedly detect updates on the server. Consequently, depending on the rate in which events are published on the server, traditional Ajax applications must constantly strike a balance between shorter and longer polling internals in an effort to improve the accuracy of individual requests. Furthermore, high polling frequencies result in increased network traffic and server demands, while low polling frequencies result in missed updates and the delivery of stale information. In either case, some added latency is incurred in message delivery. Thus, true real-time communication is simply not possible with Ajax as our only tool. More appropriate solutions are necessary to bring real-time to the web.

Real-time web applications include instant messaging applications, stock market applications, online multiplayer games and collaboration spaces. The ability to send relevant information to the client in reaction to new events, enables highly interactive web applications.

Applications that are not built to run in a web browser transport data via different application layer protocols. XMPP is a very good example of a true real-time communication protocol. Unfortunately, web browsers provide no API for protocols, such as XMPP, which would empower the bi-directional communication channel of TCP sockets.

Over the years web engineers have developed multiple technologies to realize event-driven web applications, but the observation in [FZ98], that push systems are actually implemented using a periodic pull or its derivative, is still valid. Another way of realizing asynchronous web interaction is the use of non-standard (proprietary) plug-in-based technologies like Java Applets, which extend the web browser, using a *Java Virtual Machine* (JVM), Adobe Flash or Microsoft Silverlight. Users need to install specific plug-ins that act like an intermediate layer of code, taking over responsibility of server communication and rendering the web user interface.

Social network applications specially benefit from real-time event notification of state changes to clients. Actions that users take, opinions they share or places they reside are valuable information in a social network, and broadcasting them with very low latency certainly is useful.

The next section introduces various approaches that augment HTTP and Ajax to facilitate real-time web communication.

Comet

In 2006 Russel [Rus] coined the term Comet as an umbrella term for techniques that introduce an even greater departure from the synchronous communication model by enabling an asynchronous push-style of communication over HTTP. Comet defines several techniques that allow the server to send information to the browser without specific prompting from a client. All techniques re-purpose existing mechanisms that were not originally designed for asynchronous interaction between client and server. They simulate a simplex channel for transmitting data from the server to the browser. However, with the help of an additional HTTP connection, Comet can even facilitate bi-directional communications over two HTTP connections. In the following section we analyze three Comet techniques, two using long-lived HTTP connections to stream data down to the client and another mechanism called long-polling. All techniques allow the server to asynchronously dispatch a message to the client at any point.

XMLHttpRequest Streaming The *XMLHttpRequest* (XHR) was originally designed to implement non-blocking web applications by performing HTTP requests in the background. With the help of a *Multipurpose Internet Mail Extensions* (MIME) type specified as `multipart/x-mixed-replace`, this technique provides near real-time communication semantics. After an initial request from the browser the server indicates the response type as multipart and keeps the HTTP connection open. Each update that the server now wants to communicate to the client is pushed as a separate portion of the multipart response through the long-lived connection. The multipart content type was designed for large responses, such as images. In XHR streaming this content type is re-purposed to use it as an asynchronous downstream channel. Unfortunately not all browser engines accept multipart responses and keeping the connection open brings up major performance issues for the server.

Inline frame Streaming The *Hypertext Markup Language* (HTML) `<iframe/>` element allows nesting of HTML documents. Inline frame streaming uses a hidden Iframe with the corresponding HTML document requested via chunked transfer encoding. Chunked Encoding is a feature in the HTTP/1.1 [FGM+99] specification, allowing a server to start sending a response before knowing its total length. This feature was originally intended for transfer and incremental rendering of very large documents. It is put to use to incrementally deliver data through an Iframe element in a series of chunks. Iframe streaming pushes dynamically-generated content wrapped in a `<script/>` block to the client. Browsers incrementally render chunked encoded documents after each chunk is transfered. The technique basically consists of writing `<script/>` tags

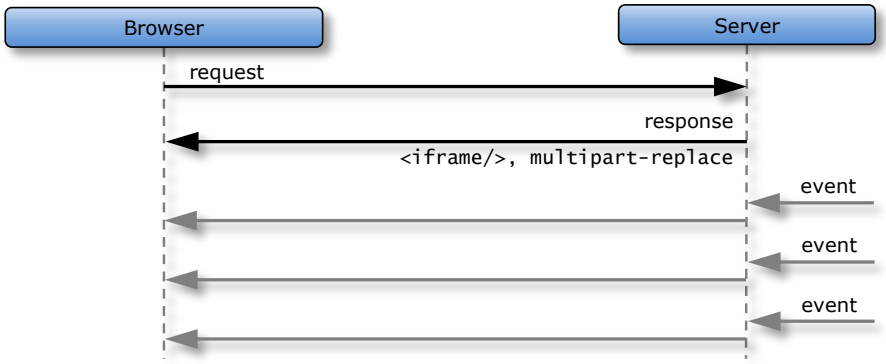

Figure 2.6.: Comet HTTP Streaming

that call a function on the parent page as information becomes available to push to the client. Not every browser behaves well with this method. In order to avoid excessive peak memory usage, the *Document Object Model* (DOM) nodes added to the Iframe are typically removed after they are rendered. Responses are easy to identify because their header contains `Transfer-Encoding chunked`. The advantage of this technique is that it provides almost negligible latency because it avoids HTTP and TCP/IP set-up and tear-down by reusing a single long-lived connection. The disadvantage of this method is that it will trigger a never-ending spinner or progress bar in most user agents, severely hurting the user experience.

Long-Polling This technique lets the connection between browser and server (HTTP request) sit idle either until the server has data to send or it times out. As soon as data or a timeout is received by the client, it sends out another request and waits for the next update. This technique dramatically decreases latency and network traffic compared to regular polling mechanisms, which means it efficiently disguises itself as a server-push technique. Figure 2.7 shows network interaction sequence for long-polling.

Comet enables event driven web application design. As soon as an event notification about state changes on the server reaches the client, JavaScript call-back functions manipulate the DOM of the web page to display the newly available data or process it in the background. However, the trouble with Comet is its lack of standard implementation due to the varying levels of support provided by browser vendors for XHR and Iframes, the two building blocks of Comet style communications. In addition, there is a significant amount of overhead, both in terms of networking and development, to manage two connections for communications. These costs can introduce latency that limit the accuracy of the real-time communications they

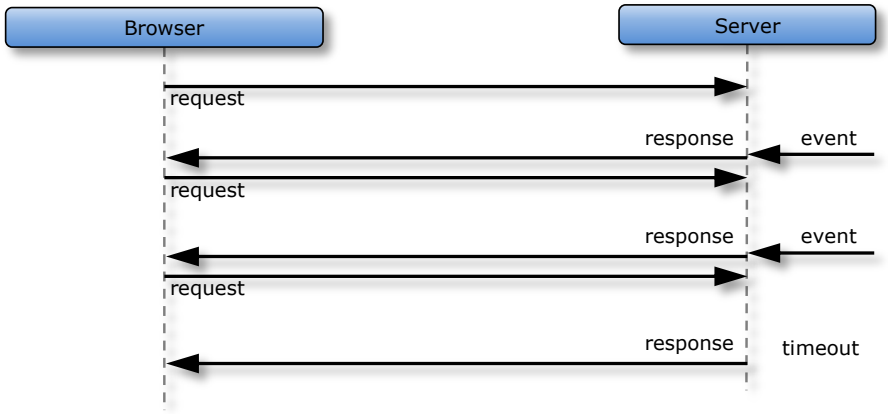

Figure 2.7.: Comet Long Polling

provide. Additionally, browsers are often configured to limit HTTP connections by domain. This further complicates the use of Comet as it often requires complex techniques such as managing multiple domains.

Several web application development frameworks implement these techniques to support server push. Popular examples are CometD[3] or Socket.IO[4]. These frameworks abstract from Comet techniques and expose interfaces to the developer that provide asynchronous communication channel semantics between server and client.

Bidirectional-streams Over Synchronous HTTP

Stream oriented communication protocols such as XMPP rely on long-lived TCP connections to enable an interactive session between entities. Unfortunately in many use cases such long-lived connections are not desirable or even possible. Web browsers communicate by definition exclusively over HTTP and are therefore constrained to the request/response pattern. The XEP-0124 *Bidirectional-streams Over Synchronous HTTP* (BOSH) [PSSAM10] defines a technology that emulates the behavior of a long-lived TCP connection using a sequenced series of request and responses that are exchanged over HTTP. Much like Comet, this technology re-purposes the HTTP protocol to facilitate asynchronous web interaction. BOSH utilizes a specialized HTTP server called the Connection Manager, which translates between the HTTP requests and responses and the stream oriented communication protocol, such as XMPP. Figure 2.8 illustrates the architecture of BOSH. In order to provide asynchronous communication between the client and the Connection Manager, BOSH implements a mechanism similar to long polling. The Connection Manager does not

[3] http://cometd.org/
[4] http://socket.io/

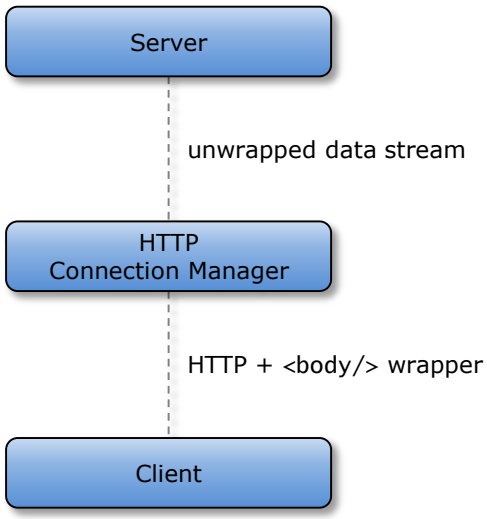

Figure 2.8.: BOSH architecture [PSSAM10]

respond to a request until it actually has data to send to the client. As soon as the client receives a HTTP response from the Connection Manager it initiates another request, thereby ensuring that the Connection Manager is always holding an open request to push data down to the client.

The main purpose of BOSH is to provide clients, which cannot maintain arbitrary persistent TCP connections with a server, the ability to use a bi-directional asynchronous communication channel. This makes BOSH a viable technology for real-time web applications. The heritage of BOSH lays in the XMPP community, but the payload of BOSH is not restricted to XML. Listing 2.3 shows an XMPP message XML stanza wrapped in a BOSH enriched <body/> element of a HTTP body with the corresponding HTTP header.

```
POST /webclient HTTP/1.1
Host: example.com
Content-Type: text/xml; charset=utf-8
Content-Length: 205

<body rid='1573741822' sid='58896478' xmlns='http://jabber.org/protocol/httpbind'>
  <message from='contact@example.com' to='user@example.com' xmlns='jabber:client'>
    <body>I said "Hi!"</body>
  </message>
</body>
```

Listing 2.3: Example for a message stanza

HTML5 WebSockets

The HTML5 WebSocket specification defines a single-socket full-duplex connection for pushing and pulling information between the browser and server. At present, WebSocket is the most advanced mechanism for facilitating full-duplex, real-time communications on the web. It takes advantage of the upgrade header of the HTTP/1.1 specification, which means its essentially a new protocol for communication. The IETF leads the efforts in standardizing the WebSocket protocol [Hic09],

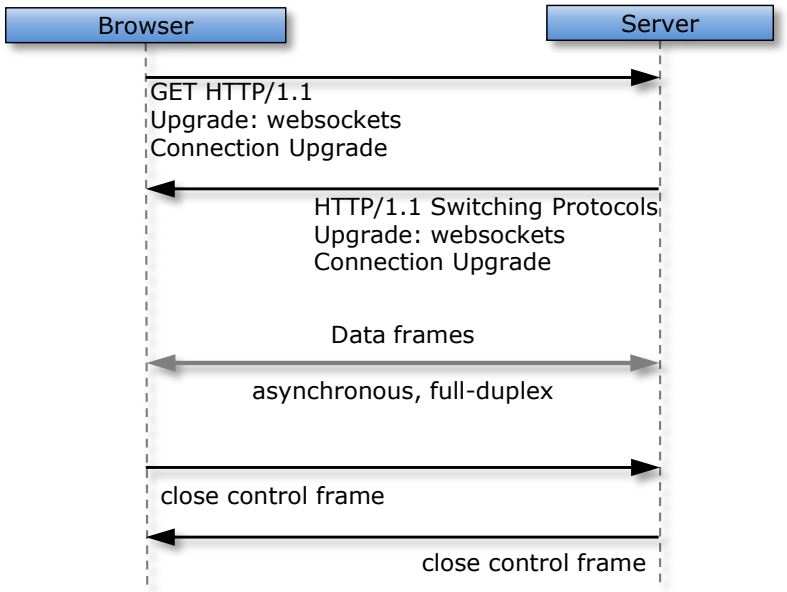

Figure 2.9.: Websocket connection life cycle

while the W3C hosts the WebSocket API specification [Hic10]. These two specification aim to expose raw socket communication to runtimes, such as JavaScript while respecting constraints of the web, such as security, firewalls and proxies. WebSockets also require some level of server-side support as there is an initial handshake over HTTP that is required in order to upgrade an existing HTTP connection to a TCP socket like connection. Figure 2.9 illustrates the life cycle of a HTTP connection upgrade to WebSocket connection

It is by far the most comprehensive solution for delivering real-time information over the Web. However, there is often the question of availability in regards to WebSockets. At the moment, browsers engines such as WebKit and Gecko support both WebSocket specifications to some degree. As of the date of this work, Microsoft Internet Explorer does not support WebSockets, therefore relying only on this new technology is infeasible.

Several web application frameworks propose fallback solutions for asynchronous

communication. In the case where the most advanced technology is not supported by either the client or the server, the framework falls back to more basic techniques such as Comet streaming or long-polling.

2.3. Mobilis Service Environment

The Mobilis Project is a joined effort between *Technische Universität Dresden* and *Pontifícia Universidade Católica do Rio de Janeiro* to build a service-oriented middleware platform for mobile real-time collaboration applications. Springer et al. [SSB+08] define a conceptual architecture for mobile collaboration systems. They propose several services for collaboration functionality placed on multiple layers. Schuster et al. [SKS+09] refine this conceptual architecture by realizing a service-based platform using XMPP as the underlying communication technology. In addition to the environment, the authors present a set of reusable collaboration services. The Mobilis Service Environment refined in [Lüb11] is going to serve as the main service provider for the concept and implementation in this work and will be introduced in the following section.

Due to the real-time requirements, all communication in the Mobilis Service Environment is realized via XMPP. The XMPP infrastructure acts as a black box delivering structured messages between *Mobilis* client applications and the *Mobilis Server*. All services in the environment are represented as XMPP entities with their respective JID and are approachable via standard XMPP communication mechanisms, therefore acting as regular XMPP clients. On the client application side, all collaboration participants are addressable via a JID (also regular XMPP clients), including the resource identifier, allowing for multiple simultaneous connections (devices and/or locations) per participant. Figure 2.10 illustrates the configuration of the Mobilis Service Environment, which provides a scalable infrastructure for client applications to connect to *Mobilis Services* to implement collaborative functionality.

Services exist on three layers. The topmost layer contains the *Coordinator Service* which uses the service discovery extension XEP-0030 [HMESA08] to find *App-specific services* and *Generic Mobilis Service* on the subjacent layers. An *App-specific Service* bundles functionality incorporated by mobile collaborative applications. All *App-specific Services* are registered with the *Coordinator Service* in order to be found during discovery. The lowest layer provides several collaboration services that implement the actual functionality of the platform and are ready to be reused by *App-specific Services* or directly by the *Mobilis* client application to form the application logic. This service-based implementation of collaborative functionality in combination with a standard XMPP server architecture allows for high flexibility, scalability and extensibility.

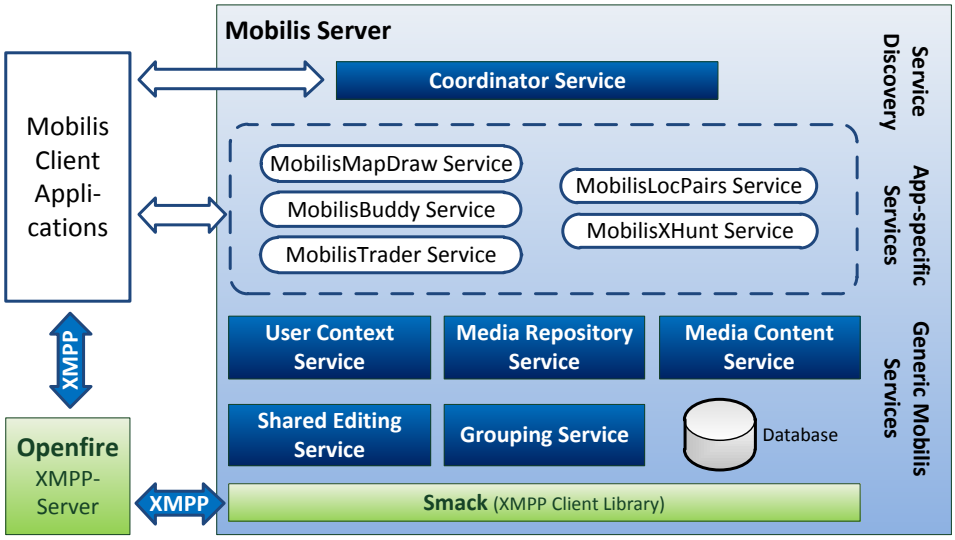

Figure 2.10.: Mobilis Service Environment [Lüb11]

2.3.1. Mobilis Services

Lübke [Lüb11] implements the aforementioned architecture using OpenFire as the XMPP server and *Java* as the development platform for the *MobilisServer*. In addition to the server implementation, Lübke also presents a framework for development of mobile social software for the *Android* platform. The Following collaboration services are incorporated into the Mobilis Service Environment.

User Context Service The *User Context Service* [Lüb11] maintains arbitrary information defining the user's context. Context information is of personal, physical or technical nature and is stored as key/value pairs in context trees. Each node in the context tree represents a specific piece of information such as the location of the user or specific characteristics of the user's device. Every node is addressable via the user's *Jabber ID* (JID) and the respective path, such as `device/display`. Updates to the context tree are distributed using the publish/subscribe semantic.

Grouping Service The *Grouping Service* basically implements the concept of *Location Based Group Formation* [LSS11]. A distinctive feature of this service is the consideration of time restriction in addition to location information of the user. Therefor the user's ability to join or even learn about a group depends on both values, location and time.

Media Repository and Content Service The *Repository Service* and the *Content Service* [Söl09] provide media sharing capabilities, especially for sharing pictures with attached meta information, such as location and time of shoot.

Users interact with the *Repository Service* to manage meta information while the *Content Service* is responsible for storing the actual media data in the background. Together they enable the user to up-/download, delete and browse media files.

Shared Editing Service Hering [Her09] designs a shared editing service facilitating synchronous collaborative work on documents among collocated or geographically distributed participants. One important design goal is the support for a variety of document types and use cases. Therefor the service is based on the Collaborative Editing Framework for XML [YI03] and adds special concurrency control for the mobile use case. Collaboration sessions are realized as XMPP MUCs. Local changes of each participant are broadcasted immediately as `groupchat` messages in order to keep a consistent version among all participants. Every participant is able to detect and solve update conflicts. The *Shared Editing Service* acts as a regular MUC participant maintaining a consistent local copy of the shared document to accommodate late-joining participants.

2.3.2. Mobilis Beans

An integral part of the *Mobilis Server* are the *Mobilis Beans*. This component holds a bunch of structures that implement service specific protocols which describe valid messages and rules for exchanging those messages for *Mobilis Services*. All XMPP messages that are exchanged are composed and parsed here. Service developers can re-use *Mobilis Beans* on both the server and client side. The greatest advantage of these components is the consistency of service specific protocol use, since *Mobilis Beans* provide clearly defined interfaces on both ends of the XMPP communication.

The goal of this work is to design and implement a web gateway for the Mobilis Service Environment supporting web client development for *Mobilis* applications.

2.4. State-of-the-Art

In this section, we take an in-depth look at state-of-the-art solutions and recent research efforts related to the objectives of this work.

2.4.1. Related Research Activities

The field of real-time eCollaboration has garnered a lot of attention throughout different research groups for more than a decade. There are early solutions for

eCollaboration systems [RG96], even for web-based [BHT97] and mobile collaboration [BLH01]. However, feature rich web-based collaboration software was not realized until the Web 2.0 phenomenon and its innovative use of technology changed the World Wide Web [MW10]. In combination with real-time web techniques the web browser constitutes a legitimate platform even for synchronous collaboration [GLG11]. In this section we will investigate recent related research activities.

Real-Time Collaborative Browsing Yue et al. [YCW09] propose a pure browser-based framework that elevates web browsing from a heavily isolated Internet activity to a real-time collaborative experience. *Real-time Collaborative Browsing* (RCB) allows participants to browse the same webpage in a collaborative session. RCB sets itself apart since it is neither server- nor proxy-based and it is not built on top of another collaboration platform. In order to enable a collaborative session, the hosting web browser incorporates the *RCB-Agent*, implemented as a Firefox browser extension. This component is approachable by all participants via the *Ajax-Snippet*, which is served by the agent. The participant poll in pre-determined intervals the *RCB-Agent* for updates on the host. All actions on the host are monitored by the agent and distributed to the participants. The main disadvantage of this solutions is the unidirectional broadcast of user action and the poll characteristic of the update mechanism. The proposed solution does not overcome the limitations of HTTP and is restricted to a poll-based synchronization model. Even though the interval is set to one second, collaborative messages are technically not distributed in real-time. Additionally the poll-based model introduces scalability issues into the system design.

Augmented Reality Gutierrez et al. propose in [GNS+11] *fAR-Play*, a platform to support the development of a variety of *Augmented/Alternate Reality Games* (AARG). The framework incorporates a mobile client application, a game status web site, the game engine and an authoring environment. The game engine implements the game logic and stores all relevant information in a database and exposes it over RESTfull web services. Mobile client applications communicate with the game engine over HTTP. Information about the state of the game are pulled in a non-blocking fashion via Ajax. The game engine provides besides a RESTlet Engine a web server that serves the mobile client application Ajax calls and the web site that holds the status of the game. *fAR-Play* provides excellent support for application development and a complex infrastructure to realize AARG. Additionally this architecture is based on pure web technologies which makes it accessible through HTTP on mobile devices and standard web browser. However, the main disadvantage is the lack of real-time notification mechanism of state changes.

Server push with instant messaging Pohja introduces in [Poh09] a server push system using XMPP technology to overcome the lack of real-time notification of state change that the previous works suffer from. The system uses an additional component to the regular web application architecture called push server. The push server is an XMPP server whose address is served to the client by the web server. On the client side the proposed system includes, besides the regular browser, an XMPP client and an update handler. The XMPP client subscribes after initial page request (which responds with XMPP server details) to a publish/subscribe node on the XMPP server and is notified in real-time of any state changes. The system relies on the pure push paradigm of XMPP and provides information delivery to the client with low latency. Unfortunately, this server push system requires an additional client component in the XMPP client, which is not compliant to web standards. Additionally, the use of an additional piece of proprietary software, beside the web browser, is not feasible for a web application architecture.

Collaborative web-base mapping of real-time sensor data Dagher et al. present in [DGI11] a framework that allows displaying real-time sensor data within a collaborative web-based environment. The framework augments a JEE-based publish/subscribe architecture to provide a social and collaborative online environment based on real-time data. The system relies on the Open Layer API in conjunction with an in-house platform called UC-IC to display real-time data from a large variety of sources. UC-IC offers a flexible environment for collaborative application development by providing the ability of recreate a familiar desktop environment and real-time server push capabilities. Real-time data is provided by a publish/subscribe semantic based on JBoss Messaging, a *Java Messaging Service* (JMS) server. UC-IC acts as a client proxy subscribing to real-time data and uses Comet techniques to push the data to the web browser. Although, Comet provides asynchronous transport mechanisms, the JMS based system itself updates data from the sensors with polling semantics. Therefore, data is technically not provided in real-time. An additional disadvantage is the use of different systems, along with different data formats (Sensor, JMS, Comet) which introduces unnecessary complexity and scalability issues.

R-U-In Banerjee et al. present in [BCD+09] *R-U-In?*, a real-time activity-oriented social networking system. *R-U-In?* integrates popular social networking portals providing real-time interests and presence management. The core components is a Java-based application on a IBM Websphere Application Server that implements the *User Interaction Manager* (UIM). The UIM provides adapters to aggregate real-time information from various sources, such as social networks, instant messaging services, and SMS. Each adapter communicates with the IBM Presence Server which provides the real-time information through a publish/subscribe mechanism. The

Interaction Manager on the other hand subscribes to data updates and forwards them through various channels, such as instant messaging, TWSS (3G devices) or HTTP. Unfortunately the HTTP channel is not push based and therefore does not provide real-time update of web clients. However, the aggregation of information from various sources to provide a single real-time data stream is promising.

2.4.2. Collaboration Tools

In addition to research activities, we investigate state-of-the-art collaborative tools. The presented solutions include open-source projects and commercial products. We pay special attention to a specific set of features and the respective degree of support in every solution. Features such as real-time capabilities, development support, mobile clients and implementation of various collaboration services are examined. Section 2.4.3 provides an overview of all mentioned collaboration tools and draws a conclusion.

Google Wave The Google Wave Protocol enables near real-time communication and collaboration in large-scale networks. Communication is based on hosted conversations, so-called Waves. All forms of collaboration, such as regular conversations or editing shared objects are contained in a wave and are hosted on a wave server. [LT09] introduces the Google Wave Protocol over XMPP that allows near real-time propagation of wave updates between Wave Providers. The heart of every collaboration session is the Wave which consists of XML documents. Waves are hosted by Wave Providers which are identified by their Internet Domain Name. Wave Users have Wave Addresses which consist of the username and a wave provider name. Each Wave has a list of participants and is hosted (maintain a copy) by all Wave Providers that have at least one registered user participating on that Wave. Due to sophisticated implementations of Operational Transformation [WML10] a Wave supports concurrent modification. Waves are modified through operations that are exchanged between the client and server. The Google Wave Protocol provides a mechanism to update the Wave Providers of all participants with very low latency. How a Wave Provider pushes the updates to the client is not defined in the specification. Google provides a web-based client application [Goo09] to participate in a collaborative session. In order to makes use of the real-time capabilities of the Google Wave Protocol a real-time web framework is used. Techniques like XHR- and Iframe streaming, long polling and WebSockets are possible. The Google Wave Protocol does not provide a framework to implement custom services or support for efficient client application development. The lack of mobile client applications is also noticeable.

Twiddla [Twi10] is a commercial real-time online collaboration tool. It is entirely browser based and uses Comet and WebSocket technologies for asynchronous communication. Provided services include instant messaging, media sharing, voice conferencing and shared editing. The system does not include support for application development or mobile clients.

Colaab [Ltd09] is a web-based real-time collaboration space incorporating instant messaging, voice conferencing and media sharing. The system requires Microsoft Silverlight for real-time semantics making it one of the plug-in based solutions mentioned in Section 2.2. Mobile clients are not supported and Colaab does not provide any extensibility or development framework.

Smyle: social collaboration application Drakontas introduces a social collaboration application called Smyle [Dra11]. The platform offers a variety of collaboration services, such as instant messaging, location sharing, shared editing and media sharing. Client applications include a web-based interface and native application for the Android platform. The system broadcasts all updates via the publish/subscribe semantic in real-time. Smyle uses XMPP as the underlying communication technology. For the web client BOSH is used to achieve asynchronous communication over HTTP. However, Smyle does not provide support for application development or extensibility and therefore limits the user to the fixed feature set the system offers.

Convore [Inc11] is a real-time communication tool that offers group chat and social network integration. The system is entirely browser based and does not require a third party plug-in. In addition to the web interface Convore also provides a mobile application for the iOS platform.

SAP Streamwork [SAP11] is a commercial collaborative decision making tool. The web-based system offers instant messaging and document sharing tools and is well integrated in existing enterprise applications offered by SAP . StreamWork also integrates with Google's OpenSocial specification and allows third-party developers to write or port applications to the environment through REST APIs.

Oracle WebCenter Real-Time Collaboration [Ora10] is a commercial product enabling users to connect and collaborate with others via instant messaging, web and voice conferencing. This solution is well integrated in other Enterprise 2.0 services available in Oracle WebCenter. Client application platforms range from desktop operating systems such as Windows and Mac OS and mobile platforms.

Open Cooperative Web Framework [The11] provides excellent support for development of cooperative web applications. The main focus of this framework is the concurrent real-time interaction among remote users and external data sources. Familiar techniques to enable real-time web, such as WebSockets and Comet are supported. Although this web framework facilitates the infrastructure and resources that are necessary for a successful collaborative web application, the main drawback is the lack of collaborative services.

PowerMeeting [Wan08] is a web-based synchronous groupware framework based on the CommonGround toolkit. The system relies entirely on web standards and facilitates Comet techniques to provide real-time broadcast of user interactions. The CommonGround toolkit employs a transactional replicated architecture providing user management, session and group management, replication and transaction management and persistence management. The PowerMeeting framework offers on top of this toolkit a skeleton for flexible, customizable and extensible development of groupware applications. Performance, scalability and rich user experience are main design goals for this framework. However PowerMeeting does not provide client applications optimized for mobile devices.

Jive Engage platform [Sof11] is a social business software platform. Jive integrates social networks and collaboration services to improve productivity in different departments of an enterprise. The main feature is communication with customers, employees and social networks integrated in regular workflows. It enables rich presence and connects with various collaboration software for employees to work together or engage customers.

OneSocialWeb [Gro11] is an effort to connect existing social networks and make them work together in a user centric way. The system is based on XMPP to streamline delivery of personally identifiable information from different social networks. Service specific data formats are translated into XMPP messages in order to be exchanged between networks. Direct user interaction from one network to another is also supported. However, information is not distributed in real-time.

2.4.3. Conclusion

Table 2.1 clearly illustrates that none of the discussed solutions offer the entire set of desired features, provide services for all collaboration tasks and support generic collaborative application development. Either solutions support application development but have no collaboration services available for reuse, or collaborative functionality is built in, but the systems are not extensible or provide support for devel-

	Framework	Mobile Collaboration	Real-time	Instant Messaging	Multi User Chat	User Context	Media Sharing	Group Formation	Location-based services	Shared Editing
Google Wave	–	–	✓	✓	✓	✓	✓	–	–	✓
Twiddla	–	–	✓	✓	✓	✓	✓	–	–	✓
Coolab	–	–	✓	✓	✓	✓	✓	–	–	–
Smyle	–	✓	✓	✓	–	✓	✓	✓	✓	✓
Convore	–	✓	✓	✓	✓	–	–	–	–	–
SAP Streamwork	–	✓	–	✓	✓	–	–	✓	–	✓
Oracle WebCenter	–	✓	–	✓	✓	–	–	✓	–	✓
OCWF	✓	–	✓	–	–	–	–	–	–	–
PowerMeeting	✓	–	✓	–	–	–	–	–	–	–
Jive Engage	✓	–	✓	✓	–	✓	✓	–	–	–
OneSocialWeb	✓	–	–	–	–	–	–	–	–	–
Mobilis Platform	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 2.1.: Features of web-based eCollaboration tools

opment. Additionally the lack of mobile collaboration integration is noticeable for the majority of solutions.

In this work, we present a web application development framework augmenting the *Mobilis* platform in order to support all aforementioned functionalities in one system. The next Chapter is dedicated to an extensive requirement analysis to build a system that eliminates all shortcomings of the state-of-the-art solutions.

2.5. Summary

In this chapter we introduce technologies essential to the work at hand. We present XMPP as a extensible communication platform that provides real-time message exchange in a federated server architecture. Beside addressing mechanisms and general security functionality, we mention the three message types (`<message/>`, `<presence/>`, `<iq/>`) and the respective usage. *XMPP Extension Protocols* are also covered.

In Section 2.2 we discuss asynchronous web communication. After defining the characteristics of asynchronous interaction, we introduce three different approaches to achieve real-time web. First, Comet is mentioned as an umbrella term for different server push techniques, such as XHR streaming, inline frame streaming and long-polling. Second, we present *Bidirectional-streams Over Synchronous HTTP* as a protocol to emulate stateful asynchronous connections over multiple individual HTTP request/response cycles. Last, we introduce WebSockets for true full-duplex socket communication between web server and web browser.

The *Mobilis Project* along with the *Mobilis Service Environment* is the focus of Section 2.3. We introduce the service oriented platform with special attention to the *Mobilis Services*. *Mobilis Beans* are mentioned briefly.

The last section provides an in-depth discussion of the state-of-the-art. First, we present related research activities with focus on real-time web collaboration systems. Second, we list state-of-the-art web-based collaboration tools. Both commercial and open-source solutions are covered. Section 2.3 and Table 2.1 summarize the discussion.

3. Requirement Analysis

In the previous chapter we provided a state-of-the-art analysis and drew the conclusion, no service based mobile collaboration platform incorporates a generic development framework for web-based client application. As discussed in the introduction, providing a web client to a mobile collaboration system has numerous advantages. The ratio between development cost and potential reach of customers is phenomenal. However, before we present a conceptual design for such a web gateway we perform in this chapter a comprehensive requirements analysis, including both functional and non-functional requirements.

3.1. Functional Requirements

This section discusses functional requirements to define what functionality the framework is supposed to provide. Each functionality is given an ID and a priority. These requirements are closely geared towards the services provided by the Mobilis Service Environment. The term *web client* in the following is used as a placeholder. It represents the actual framework which must provide the APIs for the listed functionalities to be implemented in web applications.

Users of collaboration applications need to communicate with each other. Web clients must provide essential instant messaging functionality (FUNC-1). All messages must be delivered in real-time to the recipient's instant inbox. Communication should be possible between users as well as between client and server. The addressing schema must be adopted from XMPP.

Information broadcasts from the *Mobilis* platform rely heavily on the publish/subscribe functionality of XMPP and the XEP-0060 (Publish Subscribe). Web clients must be able to perform basic actions that are necessary for a publish/subscribe system (FUNC-2). They need to be able to *subscribe* to, *unsubscribe* from and *publish* nodes. Information that is published by another user must be delivered to web clients in real-time as long as the user is on the subscriber list of the publishing node.

Presence information of participants is of utmost importance for collaboration services. Users connected through a web client must be able to propagate their own presence, subscribe to other user's presence and maintain a friends list (FUNC-3). Presence information includes *offline* and *online* plus a customizable status messages

ID	Description	Priority
FUNC-1	Instant Messaging (user-to-user, client-to-server)	high
FUNC-2	Publish Subscribe	high
FUNC-3	Presence Management	high
FUNC-4	Multi-User-Chat	high
FUNC-5	User Context Management	high
FUNC-6	Group Management	high
FUNC-7	File/Media Sharing	middle
FUNC-8	Shared Editing	low
FUNC-9	Service Discovery	high
FUNC-10	Social Network Integration	middle

Table 3.1.: Functional requirements

(e.g.*do-not-disturb*).

Web clients must be able to participate in MUCs (FUNC-4). Necessary actions include, join, leave and create a MUC. Characteristics of MUC are presented in Section 2.1.3.

Another important functionality that mobile social collaboration applications facilitate is user context management. In order to assure adaptability of services, web clients must be able to manage its user context (FUNC-5). Context information, such as location or the device used, must be available for the web client to communicate with the *Mobilis* platform.

Grouping mechanisms are an important part of many social networks. Also for collaboration between users, group functionality is very useful. Web clients must be able to connect to the grouping service and support all actions, such as create, join, leave and browse groups (FUNC-6). A web client has to incorporate its own context information with the grouping service.

File and especially media sharing is a popular application in mobile social software. Users must be able to browse the file repository, up-/download and annotate files(FUNC-7). Meta data such as time and location must be editable.

Web clients must also enable the user to discover available services of the Mobilis Service Environment(FUNC-9). Additionally, it must be possible to query for specific service and versions of services.

3.2. Non-functional Requirements

In order to provide a solid system design and high quality implementation, it is crucial to analyze non-functional requirements.

The framework must be generic in nature (NONF-1). That means, a developer must not be restricted in their choice of architecture for web client application. The framework must support the developer with all features listed in the previous section no matter which implementation concepts are followed.

ID	Description	Priority
NONF-1	Generic Framework	high
NONF-1.1	Extensibility	high
NONF-1.2	Reusability	high
NONF-1.3	Interchangeability	middle
NONF-2	A-Grade Browser Compatibility	high
NONF-3	A-Grade Mobile Browser Compatibility	high
NONF-4	Native Web Technologies (HTML, CSS, JavaScript)	high
NONF-5	Real-Time Bidirectional Communication	high
NONF-6	Security	high

Table 3.2.: Non-functional requirements

Additionally, the framework has to be extensible (NONF-1.1), reusable (NONF-1.2), and interchangeable (NONF-1.3). New functionality must be easy to add, especially client protocols for new services in the Mobilis Service Environment. Individual components of the framework must be designed for reuse by other components to provide richer functionality. Implementations of features should be interchangeable, therefore clean interfaces need to be defined.

Implementations that are provided by the framework for employment in web clients must be both desktop and mobile browser compatible (NONF-2, NONF-3). Browsers listed as *A-Grade* browsers by the *Graded Browser Support* chart maintained by *Yahoo!*[YAH11] must be supported.

The next important non-functional requirement is the exclusive use of native web technologies, such as HTML, CSS and JavaScript (NONF-4). Third party plug-ins, such as Flash and Silverlight must not be necessary to meet functional requirements.

Communication between the web client and the Mobilis Service Environment must be carried out in real-time. Messages have to be delivered with very low latency in both directions (NONF-5).

Security is the last non-functional requirement (NONF-6). Communication channels must be encrypted and entities must be authenticated using modern standards,

such as TLS and SASL.

3.3. Summary

If all of the functional and non-functional requirements are met, an effective development framework for web clients for a mobile collaboration platform can be created. Each of these requirements are addressed in the conceptual design.

4. Conceptual Design

In this chapter we present the conceptual design for a web gateway to mobile collaboration services. We rely heavily on the technologies and concepts introduced in Chapter 2. We start with a discussion about different real-time web techniques and their applicability in our system, especially with the endgoal in mind, to connect web applications with the XMPP-based *Mobilis* platform. We continue with providing an overview of the general architecture and introduce individual components in detail.

4.1. XMPP and the Web

Mobile social software, such as *MobilisGroups* [LSS11] depend on the real-time messaging capabilities of XMPP. The Android based *Mobilis* client applications employ *Mobilis XMPP on Android* (MXA) in conjunction with the Smack API[1] to establish a native XMPP connection over TCP in order to interact with the Mobilis Service Environment. The full power of TCP socket communication is exploited for bi-directional exchange of collaboration messages. Users are able to communicate, coordinate and collaborate in a social environment in real-time.

Due to the security and compatibility focused design of HTTP, the world of web application development is restricted to APIs and the sandbox that web browsers provide. Unfortunately none of the major web browsers available today provides an API for native communication over XMPP. This inadequacy is compounded by the fact that XMPP and HTTP are fundamentally different communication protocols.

The main discrepancy between the two protocols is the way clients and servers interact with each other. HTTP follows a strict synchronous request/response interaction pattern, while XMPP communicates over a long-lived (e.g. TCP) connection that runs through well defined states. Both XMPP client-to-server and server-to-server connections are long-lived and kept open as long as the session is needed. This stateful connection is contrary to the individual request/response cycles of HTTP. With HTTP, the client and server have clearly defined roles. Clients are the only instances that are able to initiate a request. Servers are not allowed to instigate communication. Because of this one-way paradigm, no always-on channel is

[1] http://www.igniterealtime.org/projects/smack/

	HTTP	**XMPP**
state	stateless	stateful
interaction	request/response	duplex
paradigm	polling	event-driven
native browser support	yes	no

Table 4.1.: Differences between communication protocols

available for the server to push data down to the client. In XMPP, it is a completely different story. Once an XMPP connection is established the entities may exchange an unbound number of messages back and forth in both directions asynchronously. XMPP based applications follow a rapid-fire, event-driven approach which can be confusing at first to developers who are more accustomed to traditional web development. However, it has a number of advantages with regard to collaboration systems. Table 4.1 summarizes the key differences between HTTP and XMPP.

Our design goal is to develop a web-based gateway that interacts with mobile social software, which is based on the *Mobilis* platform. Requirement NONF-4 ties us to native web technologies. However, requirement NONF-5 demands real-time bi-directional communication capabilities for the web client to ensure a social collaborative user experience. Social software is all about the content that is dynamically created by users. Collaboration software even demands that this content is broadcasted to all participants without any latency. Users should be able to consume information as soon as it is produced by another user. Event and state changes should be pushed asynchronously to the consumer in real-time.

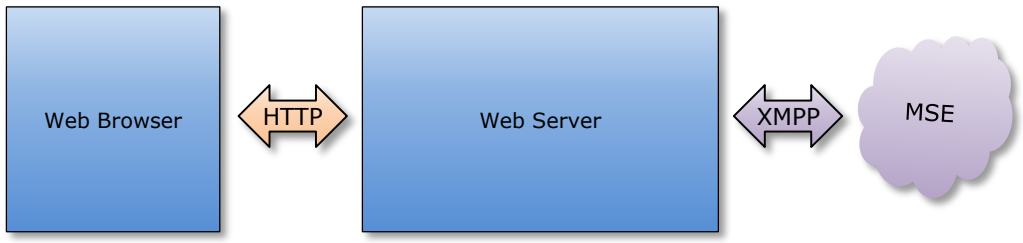

Figure 4.1.: Major components of the Architecture

We introduced in Section 2.2 several techniques that facilitate asynchronous communication over HTTP. In the following section we discuss the three aforementioned approaches and their applicability for our design. Subject to investigation are Comet techniques, WebSockets and BOSH. Figure 4.1 illustrates the necessary components

of our architecture and the communication protocols used. This basic diagram serves as the basis for discussion and will be augmented for each approach.

4.1.1. Comet

The main strength of the Comet technique is event and state change notification transport from the server to the client over HTTP with low latency. The browser does not need to employ any plug-ins and can work with standardized APIs. In addition to that, Comet techniques even facilitate bi-directional asynchronous communication with the use of a second HTTP connection. These features comprise a solution to overcome some of the differences between HTTP and XMPP. Interaction patterns with Comet resemble XMPP communication much more than traditional HTTP-based interaction. Messages which are sent from the XMPP network to the entity that is connected via the web client can be delivered over a long-polled HTTP connection or with one of the Comet streaming techniques. Both alternatives deliver messages in near real-time to the web browser.

The other difference we identified between XMPP and HTTP is state. Modern web applications are able to keep state with server-side session management and client-side cookie management. Therefore, independent request/response round trips via HTTP can be united to a stateful connection. This fact is important for our design and enables a bridge between HTTP and XMPP.

Figure 4.2 illustrates the architecture using Comet techniques.

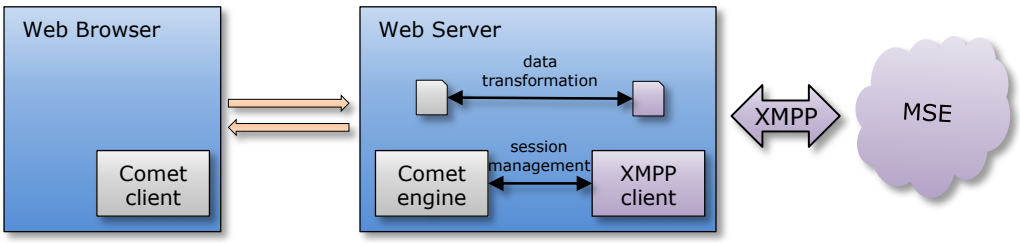

Figure 4.2.: Architecture using Comet techniques

A component that lives in the application server functions as a regular XMPP client, streaming stanzas via a long-lived TCP connection with the *Mobilis* platform. An integral part of this design is an adapter that transforms XMPP stanzas into a data format that can be pushed via Comet techniques to the client. Most Comet implementations rely on the Bayeux[RWDN07] protocol, which uses its own data format. However, the transformation code used to bridge the origin protocol (XMPP) to Bayeux (JSON-based) brings an unnecessary performance overhead into our system by forcing a message to be interpreted and process prior to being sent

over HTTP. For every user connected to the web application, the server needs to hold one separate XMPP client component to provide the semantics of a native XMPP connection. One major issue with this design is load balancing on the web server. The server must manage two HTTP connections (holding one open as the push channel and the other for upstream requests) and one XMPP connection per user. On top of that, it needs to manage the respective code transformation. Web servers usually create one thread per request. However, an HTTP based Comet solution will have an outstanding request waiting on the server to be used to send a response to the client the instant an asynchronous event occurs. This will break the thread per request model. Better technologies are needed on the server side in order to serve a growing number of client requests.

In addition to that, presence information for each client needs to be managed, since clients are not bound by any real stateful connection to the web server. This non-trivial architecture is able to bridge XMPP to the browser over HTTP, but comes with a significant price.

Additionally, adherence to the numerous XEPs pose another problem with this design. In order to implement the XEP-0096 (Si File Transfer) for example, we have to develop a complex adapter that transforms the XMPP managed to Comet-base self-managed file transfer.

Comet certainly is a technique we can base our framework on, but considering the disadvantages we proceed to find better and more viable approaches.

4.1.2. WebSockets

The WebSockets protocol is a bi-directional protocol that provides a simple message-based framing layer over raw sockets and allows for more robust and efficient communication between a browser and a server. After the initial handshake and the following upgrade of the HTTP connection, server and client are free to exchange data frames independent from each other. This full-duplex communication channel provides a suitable basis for bridging a web client to the XMPP network. The WebSocket API provides an `onMessage` event for JavaScript to handle incoming messages, which serves the even-driven characteristic of an XMPP-based application well. Once the WebSocket connection is established, client and server can basically exchange any data packets, either text or binary. Therefore, the messages received by the server (XMPP) can be the same message delivered to the browser, eliminating the complexity and performance concerns introduced by transformation code with the Comet techniques.

Additional benefits and advantages of communication via WebSockets are the elimination of the known Comet request/response mechanism overhead, and apart from the initial handshake, the entire HTTP overhead. This leads to shorter latencies

and higher bandwidth.

The main disadvantage, as of now, is the fact that WebSockets require some level of server-side support for the initial handshake. This is required in order to upgrade an existing HTTP connection to a raw TCP/IP connection. The lack of server support along side the lack of browser support for the WebSocket API makes this approach highly unattractive as a stand-alone solution.

The IETF internet-draft *An XMPP Sub-protocol for WebSocket* [Mof10] introduces a sub-protocol for XMPP to enable entities to connect natively to XMPP over WebSockets instead of regular TCP sockets. This protocol also defines XMPP connection management over WebSockets. Unfortunately, this internet draft is at the beginning stages of the standards track and not yet implemented in any client or server libraries. Without this sub-protocol, the system design based on WebSockets also lacks connection management like the Comet approach.

Figure 4.3 shows the necessary components of the architecture based on WebSockets.

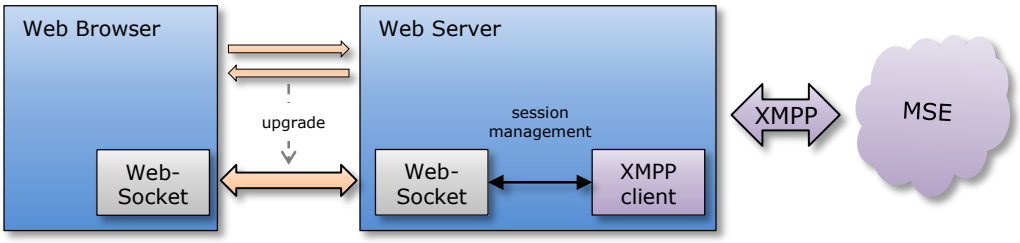

Figure 4.3.: Architecture with WebSockets

As discussed, this design approach is not feasible at the moment, but there is very high probability it will be the preferred basis for architectures bridging HTTP and XMPP in the future.

4.1.3. BOSH

BOSH is designed to provide the semantics of a long-lived connection via multiple independent HTTP request/response cycles. BOSH solves a different problem than Comet or WebSockets. Its focus does not lie on enabling the server push, but rather provide a stateful bi-directional communication channel over HTTP. Different types of long-lived connections can be tunneled through HTTP with BOSH. For our architecture, XEP-0206 (XMPP over BOSH) [PSA10] is interesting. This XEP provides an extension to BOSH that enables XMPP streams to be bound to HTTP over BOSH. The protocol basically specifies what XML messages in the `<body/>` wrapper can be

exchanged and what messages are appropriate replies to others in order to establish and maintain an XMPP connection.

The connection semantic that BOSH provides is very interesting for our system design. XMPP over BOSH allows the web client to communicate with the XMPP server through a native connection. The client connects through a standard *Uniform Resource Locater* (URL) over HTTP on port 80. This request is then proxied by the web server to a different port, which is operated by the XMPP *Connection Manager*. As a result, the web client can operate from behind a firewall using commonly supported web standards. The overhead of maintaining the long-polled HTTP connections of BOSH is mostly handled by the Connection Manager rather than the web server or the web application. We are able to establish a bi-directional stream over HTTP between web clients and the Connection Manager, while the Connection Manager works as a proxy to the *Mobilis* platform. The Connection Manager itself communicates as a normal XMPP entity by streaming XML with other entities. This way the web client is able to maintain a stateful connection to the Mobilis Service Environment. Connection management, security and real-time delivery of notification and state changes are all handled by BOSH and XMPP over BOSH. Especially security of XMPP with encryption via TLS and authentication with SASL is handled by BOSH. The use of BOSH enables our design to meet NONF-5 more than the previous solutions.

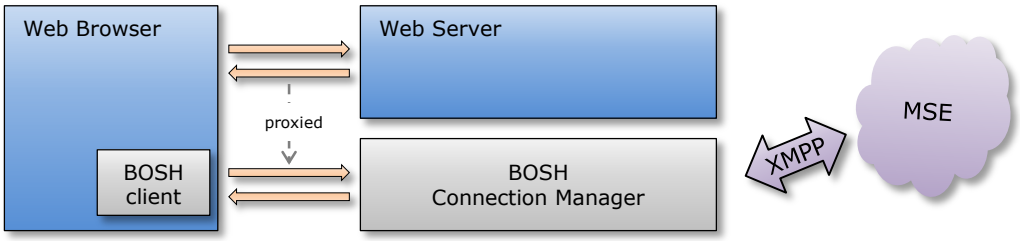

Figure 4.4.: Architecture using XMPP over BOSH

The browser is able to send/receive presence information. Incoming stanzas for the web client are sent over a long-polled HTTP request and trigger events on the client side which are handled by a BOSH client library.

Since the entire *Mobilis* platform is based on collaborative services that are implemented as regular XMPP clients, all content that needs to be processed by a web client application is available through XMPP messages. It is conceivable to write a fully featured browser-based *Mobilis* client application with no web application server whatsoever. Figure 4.4 illustrates the architecture using BOSH.

The advantage of the BOSH powered approach over the first two designs, is the fact

that on top of the asynchronous interaction mechanism that all three provide, BOSH also facilitates necessary functionality for a long lived connection plus connection management for XMPP.

Even if XMPP over BOSH is an emulation of push and not pure XMPP anymore, it has many benefits over plain Comet since it contains other characteristics of XMPP, such as security, the use of arbitrary XEPs and persistent connections even if the underlying network connection is unreliable. This is especially important in the mobile web application use case. Schuster et al. mention in [SSS10] that XMPP is not well suited for wireless networking connections due to persistent connection. BOSH addresses this inadequacy.

The main disadvantage is the need for the Connection Manager. It is an integral component of BOSH. For web applications that are not connected to an XMPP network whatsoever, the need of a Connection Manager might pose a problem. Connection Managers are usually not implemented within a web server, but rather within an XMPP server. This is different from Comet where most of the major application servers support Comet protocols (e.g. Bayeux). However in our system we are connected to an XMPP network and therefor have the Connection Manager at hand.

A framework based on BOSH and BOSH over XMPP bring numerous advantages. It enables us to met the requirements stated in Section 3. Disadvantages are non-existent for our use-case.

4.2. Architecture

After the previous section discussed the pros and cons for different real-time web technologies with regard to our framework design, we present in the following section the architecture of the *Mobilis* web-gateway. Section 4.1 clearly drew the conclusion that the most advantageous technology is BOSH and XMPP over BOSH. Following this design decision, Figure 4.5 shows the architecture of the *Mobilis* platform augmented with the web gateway. This figure provides a very coarse grained view of the architecture to introduce the main components and communication protocols between them.

All communication in this architecture is TCP/IP based. Android *Mobilis* client applications directly open a regular TCP socket to establish a native XMPP connection to the XMPP server.

On the server side, the *Mobilis Server*, which incorporates the Mobilis Service Environment, is also connected to the XMPP server. In this diagram, the XMPP server and the *Mobilis Server* run on the same machine enabling direct communication between them. Since *Mobilis Services* are implemented as regular XMPP clients, the

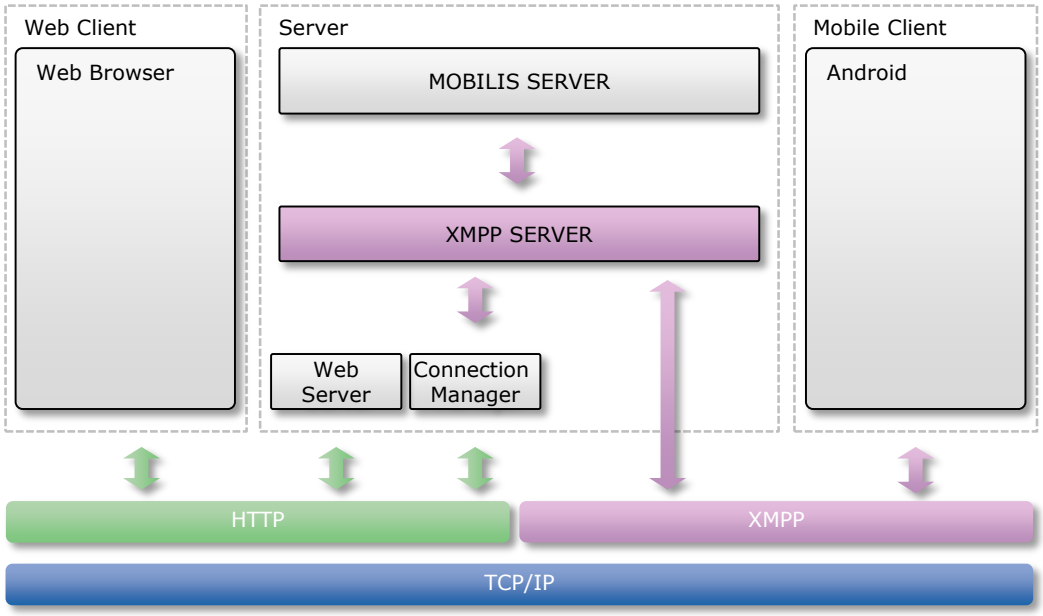

Figure 4.5.: Mobilis Web Gateway Architecture

Mobilis Server and even individual services can be distributed on different physical machines.

For the web client, we are limited to the APIs and the sandbox the web browser provides. Therefore, it is impossible to open up a TCP socket and establish a native XMPP connection. We split the communication in two separate logical channels. First, all static content such as HTML, CSS and JavaScript is provided by a regular web server that responds to the initial HTTP request. Second, all following request/response cycles are between the browser and the Connection Manager emulating a long-lived bi-directional connection. This connection is used to transport XMPP messages to and from the browser. Details concerning the protocol and the exchange of messages will be provided later in this chapter.

With this architecture, we meet several of the non-functional requirements listed in Chapter 3. For the web client we are exclusively using native web technologies, namely HTTP as the transport protocol, HTML to specify semantics, CSS for the layout and JavaScript for application logic, such as the BOSH client library (NONF-4). The design also provides a bi-directional asynchronous connection between the browser and the Connection Manager, and therefore XMPP (NONF-5).

To understand this design in detail, we will next drill down into the architecture and present the individual components and their interaction. The Connection Manager basically acts as a proxy for the web client towards the XMPP network. Ev-

erything beyond this point from the web client's perspective is exactly like discussed in [Lüb11]. All service interfaces are well defined through specific protocols encapsulated in *Mobilis Beans*. Communication between the Connection Manager and the *Mobilis Server* is equivalent to the Android-based client applications. Therefore, we omit this part of the architecture for further discussion and concentrate on the web client and its communication with the Connection Manager.

4.3. Mobilis Web Client

Figure 4.6 illustrates the structure of a web-based client for a *Mobilis* application. The objective of this chapter is to present the conceptual design of a framework for *Mobilis* web client development. A major design goal for the web gateway framework is extensibility and reusability (NONF-1.1, NONF-1.2). This means we must provide components that encapsulate functionality, such as real-time communication management, and interact via well defined interfaces. The framework must also provide mechanisms to implement and integrate new protocols in case the Mobilis Service Environment is extended with new services. In the following section we take an in-depth look at necessary components and libraries that comprise our framework. A web-based client for *Mobilis* applications that is built with our framework has three main parts.

- The low-level *XMPP Web Client*, which is responsible for the communication between client and server.
- The *Mobilis Services Web Client*, which consists of the backbone and core functionality of the framework. Protocol implementations that are specific to *Mobilis Services* are also part of this component.
- The *Mobilis* application logic, which is different for each *Mobilis* application. It holds the user interface and the static content of the web client.

4.3.1. Event-driven Architecture

Before we discuss the individual components, it is important to mention that control flow in *Mobilis* web clients is event-driven. In essence, our framework provides abstraction from any kind of communication details and XMPP stanza exchange. The framework takes care of establishing an XMPP connection, creating stanzas, sending them over BOSH to the Mobilis Service Environment and triggering callback events for incoming stanzas. Web client developers will only deal with asynchronous APIs. This concept is important in order to understand how to use the proposed framework. The way to use functionalities of the framework is mostly one of two approaches:

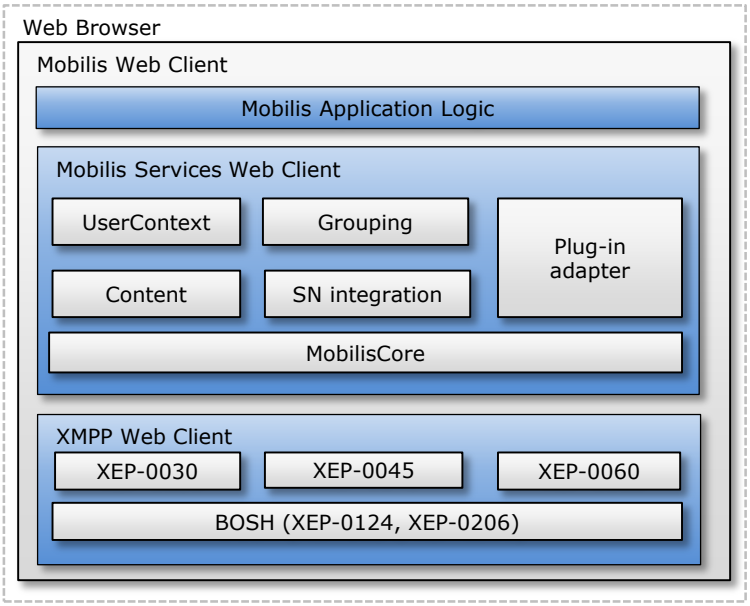

Figure 4.6.: Mobilis Web Client Architecture

- JavaScript functions are called to build and send stanzas, while callback functions are passed as arguments to handle the event of incoming responses. The following pseudocode demonstrates this type:

  ```
  function (attr, callback, errback)
  ```

  `attr` holds all neccessary information for the stanza to be communicated, `callback` and `errback` are functions that handle the positive and negative response stanza respectively.

- General event handlers can be defined to be invoked for incoming stanzas that are received not as a response, but rather standalone messages such as message or presence stanzas. The framework checks attributes of incoming stanzas and triggers the appropriate event.

  ```
  function (handler, namespace, name, type, id, from)
  ```

  `handler` is the function that is called in case all parameters match. `namespace`, `name`, etc. are the parameters to define which incoming stanzas trigger the event.

Events that are triggered by user interactions or timed events are handled by the application specific logic and are not the responsibility of the framework.

This design avoids any kind of blocking *Remote Procedure Call* (RPC) semantic on the client side. An event-driven web client accommodates the processing of real-time data very well. The application is never blocked and reacts in real-time to newly available data. All the developer has to do is register the event handlers and define callback functions. We now move on to introduce the components that make this architecture possible.

4.3.2. XMPP Web Client

The *XMPP Web Client* is responsible for communication via a BOSH connection, therefore implementing a BOSH client library. This library utilizes the XHR object to transfer data via HTTP requests between the browser and the Connection Manager. The Ajax engine of the browser is responsible for all HTTP requests to the Connection Manager. Data is retrieved decoupled from user interaction. This component is required for asynchronous web communication and is implemented in all modern browsers. BOSH relies on the ability to send and receive HTTP messages asynchronously. All communication runs through port 80 or 443 and is therefore able to pass firewalls. With the implementation of XEP-0206 (XMPP over BOSH), standardized authentication and encryption mechanisms can be used between the client and the server. Besides the HTTP header, the `<body/>` element, which is augmented with special attributes to control the session, is composed here. Listing 4.1 and 4.2 show an initial BOSH message that is sent by the client.

```
POST /http-bind HTTP/1.1
Host: mobilis.inf.tu-dresden.de
Connection: keep-alive
Referer: http://mobilis.inf.tu-dresden.de/webclient/
Content-Type: text/plain; charset=utf-8
```

Listing 4.1: Initial BOSH HTTP POST header

```
<body hold="1"
      secure="true"
      rid="2202968608"
      xmlns="http://jabber.org/protocol/http-bind"
      to="mobilis.inf.tu-dresden.de"
      wait="60"
      content="text/xml"
      charset="utf-8"
      xml:lang="en"
      xmpp:version="1.0"
      xmlns:xmpp="urn:xmpp:xbosh"/>
```

Listing 4.2: Initial BOSH HTTP POST body

The HTTP header indicates a regular `POST` request to `http://mobilis.inf.tu-dresden.de/http-bind` which is the URL used to reverse proxy the Connection Manager(more on that in section 5.2). The `<body/>` element features special attributes. `hold` for example, indicates the maximum number of HTTP connection the Connection Manager is allowed to hold open. `secure` indicates that TLS and SASL are mandatory for this connection. `rid` is a request ID that is incremented for each following request.
The server answers with an HTTP response that looks similar to Listing 4.3 and 4.4.

```
HTTP/1.1 200 OK
Content-Type text/xml; charset=utf-8
Content-Length 673
Keep-Alive timeout=5, max=99
Connection Keep-Alive
```

Listing 4.3: BOSH HTTP response header

```
<body xmlns="http://jabber.org/protocol/httpbind" authid="4cee67e8" sid="4
    cee67e8" secure="true" requests="2" inactivity="30" polling="5" wait="60"
    hold="1" ack="2202968608" maxpause="300" ver="1.6">
  <stream:features xmlns="http://etherx.jabber.org/streams">
    <mechanisms xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
      <mechanism>DIGEST-MD5</mechanism>
      <mechanism>PLAIN</mechanism>
      <mechanism>ANONYMOUS</mechanism>
      <mechanism>CRAM-MD5</mechanism>
    </mechanisms>
    <compression xmlns="http://jabber.org/features/compress">
      <method>zlib</method>
    </compression>
    <bind xmlns="urn:ietf:params:xml:ns:xmpp-bind"/>
    <session xmlns="urn:ietf:params:xml:ns:xmpp-session"/>
  </stream:features>
</body>
```

Listing 4.4: BOSH HTTP response body

The response initiates the security handshake between client and server that opens the XML stream for exchanging stanzas. Supported SASL authentication mechanisms are listed for example. The client answers the handshake with another HTTP request initiating the SASL challenge response cycle. Once the handshake is completed, the client opens a new encrypted stream which is then used to exchange stanzas, for example to request a resource, which is answered by the server with a `bind` stanza of type `result`. Further HTTP requests must at least provide the `sid` (Session ID), `rid` (Request ID) and the namespace `http://jabber.org/protocol/`

`httpbind`, so the Connection Manager can keep the long-lived connection semantic.

The *XMPP Web Client* provides an interface to send arbitrary XMPP messages over the established connection. It also passes back all incoming stanzas for further processing.

Besides the BOSH client, the *XMPP Web Client* holds implementations of three XEPs that are used by the *Mobilis Services*. Section 2.1.3 introduced all three XEPs:

- Service Discovery (XEP-0030)
- Multi User Chat(XEP-0045)
- Publish/Subscribe(XEP-0060)

The functional requirements FUNC-2 and FUNC-4 are met with the implementation of XEP-0060 (Publish Subscribe) and XEP-0045 (Multi User Chat) respectively. The components employ the protocols discussed in those XEPs and enable the *Mobilis* web client to use the extensions of XMPP easily. All extension implementations use the interfaces provided by the BOSH client library.

XMPP Web Client is an integral component for our framework to support web client development for *Mobilis* applications, since all real-time communication with the *Mobilis Server* is encapsulated here.

4.3.3. Mobilis Services Web Client

The *Mobilis Services Web Client* is the main component of our framework. In Section 2.3 we introduced *Mobilis Beans*. *Mobilis Beans* are a collection of structures that implement service specific protocols which describe valid messages and rules for exchanging these messages with *Mobilis Services*. The *Mobilis Services Web Client* provides the same functionality for the web client. The same protocols and rules are implemented here to compose valid service requests to the Mobilis Service Environment. In the following section we present the interfaces the individual components provide.

Mobilis Core

The *Mobilis Core* is the base library for the *Mobilis Services Web Client*. Functionalities that every *Mobilis* web client needs are implemented here. After the *XMPP Web Client* has established a connection to the XMPP network, *Mobilis Core* provides an abstraction from the connection details (`Mobilis.connection`) and exposes simple methods to send stanzas and define callbacks. *Mobilis Core* presents a completely asynchronous API to higher components that implement the service specific protocols. *Mobilis Core* allows for defining event handlers to react to incoming stanzas. XMPP messages that match the defined attributes, such as the `id`, `type` or `namespace`,

trigger events that have handlers bound to them. Event handlers are implemented as JavaScript callback functions that handle responses (or incoming messages). *Mobilis Core* exposes a comfortable interface to the specific service implementations, such as *User Context*. XMPP introduces three distinct types of stanzas that are used for communication:

- `<message/>` provides information exchange between entities via a push mechanism
- `<presence/>` broadcasts presence information of entities through a publish/subscribe pattern
- `<iq/>` implements `get` and `set` semantic in a request/response fashion

FUNC-1 and FUNC-3 are met through handling the first two types of stanzas. `<iq>` stanzas are the building blocks for XEPs and service specific protocols. In order to meet further requirements *Mobilis Core* provides following methods:

Connect establishes an emulated long-lived connection to an XMPP server over BOSH. Security policies, such as TLS or SASL are considered during the hand shake process with the XMPP server (NONF-6). Parameters are the JID and the password of the user. Additionally we can pass a callback function that reacts to status changes of the connection. Possible statuses include: `connecting`, `authorized`, `connected`, `disconnected`, etc. For example, for the status `connected` a service discovery could be performed to set namespaces and JIDs for all *Mobilis* services.

Disconnect starts a graceful tear down process of the connection, including an unavailable presence broadcast. A reason for the disconnect can be passed as an argument. The connection status callback is notified of the disconnect process.

Send takes an XML element as an argument and prepares the exchange of a corresponding stanza over the BOSH connection. This method is for message and presence stanzas since those types are not acknowledged and do not required any response handling.

SendIQ allows to send IQ stanzas. Beside the `<iq/>` element this action takes two callback functions as parameters, a callback for the successful response of type `result` and another callback for response type `error` and for the timeout case.

AddHandler adds handlers for incoming stanzas to the connection. The following arguments can be passed. The handler function and attributes, such as `namespace`, `name`, `type`, `id` and `from`. The handler callback is called for any incoming stanza that matches the parameters. A stanza has to meet all parameters

that are supplied for the handler to be invoked. A boolean return value of the handler determines if it is invoked everytime (`true`) or just once (`false`) a matching stanza arrives. The return value of `addHandler` is a reference to the newly added handler.

DeleteHandler takes a reference to a handler as the argument and deletes the handler from the connection. Incoming stanzas do not trigger this handler after it has been deleted.

Another very important responsibility of *Mobilis Core* is the service discovery within the Mobilis Service Environment. Since services of the *Mobilis* platform are versioned, we cannot rely on just the XEP-0030 (Service Discovery). *Mobilis Core* supports the discovery protocol used by the *Coordinator Service* [Lüb11] and allows following methods (FUNC-9):

MobilisServiceDiscovery performs a general discovery of services that are running in the Mobilis Service Environment. The client is able to retrieve a complete list of services or request information about a specific service. In order to retrieve information about each instance of an *App-specific service* the latter option is used.

CreateNewServiceInstance allows the creation of new service instances within the *Mobilis Server*. Some services can be instantiated multiple times, such as *App-specific Services*. The client can provide a name and a password for the newly created service. Deleting a service instance is not supported since the *Mobilis Server* implements garbage collection functionality for services that are no longer in use.

In order to achieve better understanding of the event-driven nature of web clients and how the different components interact, Figure 4.7 shows a sequence diagram for a service discovery. All function calls in this example are of asynchronous nature. Within the application logic either a user interaction event, timed event or other event triggers the `Mobilis.core.mobilisServiceDiscovery` function call. As arguments we pass two callback functions that handle each possible response type respectively (`result`, `error` or timeout). `Mobilis.core` registers these handlers and then builds an stanza that follows the protocol of the *Coordinator Service*. `Mobilis.connection.sendIQ` send the IQ stanza within a BOSH `<body/>` wrapper to the Mobilis Service Environment. IQ stanzas require a response `<iq/>` which is sent by the *Coordinator Service* containing a list of all registered services. The incoming IQ stanza triggers the callback event. The function that was added as the callback for a response of type `result` is called by `Mobilis.core`. Since functions are first order objects in

JavaScript they can be passed as arguments. Therefore, the callback is called in `Mobilis.core` but defined and executed in the App logic.

Details about each XML stanza that is exchanged will not be discussed in this work. All stanzas have to comply with protocols that are defined in other works. For example, `connect` and `disconnect` are defined in the BOSH protocol.
`MobilisServiceDiscovery` stanzas are defined by the *Coordinator Service* in [Lüb11]. For each namespace, we mention the associated work which defines the structure of the stanzas.

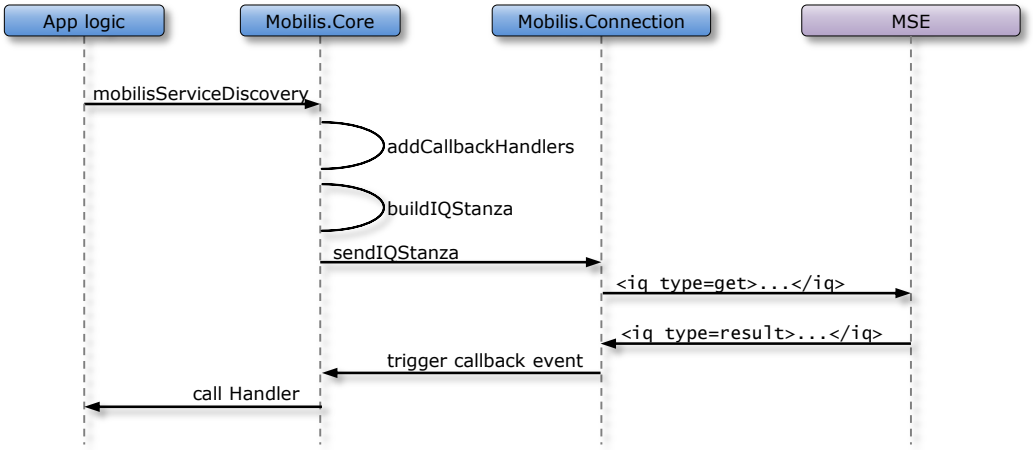

Figure 4.7.: Sequence diagram for MobilisServiceDiscovery

Section 2.3 listed services of the Mobilis Service Environment that are reused by application specific services or employed by client applications directly. In the following section we present client implementations of those services within the *Mobilis Services Web Client*.

User Context

The *User Context* component includes methods to satisfy requirement FUNC-5. Section 2.3.1 introduced the *User Context Service*[Lüb11]. The following methods are provided by the *Mobilis Services Web Client* to communicate with the *User Context Service*.

Subscribe enables the client to communicate interest in automatic update notifications for a specific node in a context tree. The JID and a relative path are passed as arguments to define the exact context information the user wants to subscribe to. Every subscription requires an authorization by the publisher.

Unsubscribe cancels the existing subscription. The JID and path to the node are required parameters for this method. Unsubscribe does not require any authorization.

Authorization is a method to respond to an authorization request to a client's node. The result can either be a positive response (`result`) to indicate that authorization has been granted or an `error` response for declining the request.

AddAuthorizationHandler handles incoming authorization requests. Parameters include the path to the node and a handler function. The return value is a reference to the newly created handler.

DeleteAuthorizationHandler removes the authorization request handler that is addressed by the passed reference.

Publish allows the client to broadcast changes to one of its context nodes. Parameters for this method include the path to the node, a key/value pair and the associated type for the node.

AddUpdateHandler provides an interface to register event handlers for incoming message stanzas that represent updates to context information the client has subscribed to. The parameters are the path to the node and a callback function that encapsulates the handler logic. This method is important since update notifications are broadcasted via regular message stanzas that are not sent as a response. A reference to the created update handler is returned.

DeleteUpdateHandler deletes the handler that handles incoming update notifications. The reference to the handler is passed as an argument.

Grouping

We introduced the *Grouping Service*[Lüb10] of the Mobilis Service Environment in 2.3.1. The *Grouping* client provides methods to follow the protocol of the *Grouping Service* in order to meet requirement FUNC-6. The following methods are incorporated in the *Grouping* client:

GroupQuery allows the client to perform a query request to discover groups that are near the user's location. Latitude and Longitude values of the user's location are passed as arguments. Result stanzas include a list of groups with respective details.

GroupInfo augments the GroupQuery method by providing a query mechanism for specific groups which are passed as an argument (`Group-ID`). GroupInfo responses include more fine-grained information, such as a list of members.

GroupMemberInfo is the method to set and get context information of users. It allows the user to provide information about his or her own context and to retrieve information of other users. Context information is passed as an array argument with key/value pairs for each context item. The result type depends on the type of request and is handled by the passed callback functions for each type.

GroupCreate creates a new group on the server. All necessary information regarding the new group is passed as an attribute array argument. The server sends the `Group-ID` as a result. Updates to information about the group are also handled by this method.

GroupDelete removes a group with the given `Group-ID` from the server. Successful responses are of type `result` while `error` responses need to be handled also.

GroupInvite enables a user to put other users on an invite list for a specific group. For closed groups, only users on the invite list are able to join the group. Parameters are the `Group-ID` and the JIDs of an unbound number of invitees.

GroupJoin allows the user to join a group. Parameters include `Group-ID` and location information of the user. The successful join is acknowledged with an empty `result` response while several `error` responses are possible.

GroupLeave indicates the user's wish to leave a specific group. The `Group-ID` is passes as a required argument. In case the user was the only member, the group is deleted from the server after the user has left.

Content

The Content client follows the protocol for the *Repository Service* and the *Content Service* in the Mobilis Service Environment[Söl09]. It makes heavy use of the Si File Transfer(XEP-0096) component implemented in the *XMPP Web Client*.

RepositoryQuery allows the client to issue a query to the repository in order to browse enlisted content items. As a parameter, an array is passed that represents conditions to narrow the intended result set. The `result` response contains a list with repository items that can be referenced through an *Unique Identification* (UID).

RepositoryDelete causes the *Repository Service* to issue a delete request to the *Content Service*. The passed UID references the repository item.

ContentDownlod initiates the content transfer of the repository item defined by a UID.

AddIncomingContentHandler allows the client to implement a handler routine for incoming content. After the *ContentDownload* method, the *Content Service* initiates an SI file transfer which has to be handled and responded to by the client.

ContentUpload indicates the client's wish to upload content that is referenced by a UID.

AddUploadContentHandler adds a function as a handler for an incoming upload request. Similar to *ContentDownload*, the actual file transfer is initiated by the *Content Service* on the *Mobilis Server*.

Social Network Integration

Social network services, such as Foursquare, Gowalla and Qype provide APIs to serve service specific data. For example, Foursquare provides information about venues that users have created. Web applications can initiate requests for a list of venues near a certain location and all necessary information is included in the response. In order to ensure authorization of users most of these APIs use OAuth2. Social Network Integration is independent from all other XMPP communication. Existing JavaScript libraries to connect to service specific APIs are used. Therefore social network integration is independent from all other XMPP communication as it can be realized entirely on the client-side.

Plug-in Adapter

The service-based design of the *Mobilis* platform makes it easily extensible. To incorporate new services into *Mobilis* applications, the service providers have to specify *Mobilis Beans* for their service. This way, the use of the service is independent from the actual implementation. To accommodate that degree of extensibility, which is formulated in NONF-1.1, the *Mobilis Services Web Client* includes a plug-in adapter. All aforementioned service clients are actually implemented as plug-ins to the *Mobilis Core*. Plug-ins are dynamically loaded by the core component. This design also makes interchangeability of service client implementations possible (NONF-1.3).

4.4. Summary

In this chapter we addressed the functional and non-functional requirements of chapter 3 and provided a conceptual design for a framework supporting web-based client development for *Mobilis* applications. Table 4.2 lists all requirements and indicates whether the proposed concept addressed the issue. All requirements prioritized as high are met and considered in our design. Unfortunately FUNC-8 (low) could not be

ID	Description	Addressed
FUNC-1	Instant Messaging (user-to-user, client-to-server)	✓
FUNC-2	Publish Subscribe	✓
FUNC-3	Presence Management	✓
FUNC-4	Multi-User-Chat	✓
FUNC-5	User Context Management	✓
FUNC-6	Group Management	✓
FUNC-7	File/Media Sharing	✓
FUNC-8	Shared Editing	—
FUNC-9	Service Discovery	✓
FUNC-10	Social Network Integration	✓
NONF-1	Generic Framework	✓
NONF-1.1	Extensibility	✓
NONF-1.2	Reusability	✓
NONF-1.3	Interchangeability	✓
NONF-2	A-Grade Browser Compatibility	✓
NONF-3	A-Grade Mobile Browser Compatibility	✓
NONF-4	Native Web Technologies (HTML, CSS, JavaScript)	✓
NONF-5	Real-Time Bidirectional Communication	✓
NONF-6	Security	✓

Table 4.2.: Requirements consideration

addressed properly, due to the complexity of the service specific protocol. However, this service can be integrated into the framework via the plug-in adapter.

5. Implementation

In this chapter we introduce certain aspects and details of the implementation of the concept presented in the previous chapter. First, we discuss the specifics of the web client architecture. We then cover the implementation of the *XMPP Web Client* including the BOSH client library. A presentation of the most important details of the *Mobilis Services Web Client* implementations is next. We conclude this chapter with a presentation of the extensibility features the framework provides.

5.1. Single Page Application

Before the discussion of implementations specific to each component, we need to state some facts about the intended use of the framework. The design decision, to built the system based on BOSH, shifts the entire client application logic into the web browser. We state in Section 4.1.3, that it is conceivable to build a *Mobilis* web client without an application server whatsoever. This is possible by implementing the web clients as a *Single-page Application* (SPA). An SPA consists of static content, such as HTML, CSS, images and JavaScript files, which comprise the user interface in the form of a web page. For the entire application session the initial web page is not reloaded or left. All dynamic content, such as collaborative messages, is retrieved in the background using scripted HTTP and then injected into the run-time representation of the web page, called DOM.

Our framework provides the *XMPP Web Client* for retrieving content in real-time. SPAs take it a step further than familiar Ajax web applications that still require the occasional page load in order to advance the interaction of the user with the application. This architecture is especially useful for *Mobilis* web clients, since the established BOSH connection lives only within the DOM of the web page. A page reload results in aborting the BOSH connection. BOSH provides functionality that allows passing of connection details, such as `sid` and `rid`, between DOMs which would make a single connection over multiple page loads possible. However, for the implementation of this framework we decided in favor of the SPA architecture. But an extension to support page reload is certainly possible.

All functionality of our framework is provided by a single JavaScript file (`mobilis.js`). Application specific logic is implemented in separate JavaScript code. To avoid incompatibility issues between browser implementations, our framework is based on

the de-facto standard JavaScript library for DOM manipulation and event handling, called *jQuery* [The10]. The `mobilis.js` file includes all necessary libraries to run the *Mobilis* web application, such as libraries associated with the *XMPP Web Client* and the *Mobilis Services Web Client*. JavaScript engines immediately interpret code on page load. We exploit this fact to make our libraries instantaneously available for the web application. We facilitate so-called anonymous self-executing functions to do so. Listing 5.1 shows such a function defining a constructor for a class called `Mobilis`.

```
1 (function($){
2     if (typeof window.Mobilis == "undefined"){
3         var Mobilis = window.Mobilis = function(){
4             // constructor code here
5         }
6     window.Mobilis = new Mobilis();
7     }
8 })(jQuery);
```

Listing 5.1: Anonymous self-execution function in Javascript

Then a new object (`Mobilis`) in the `window` scope is created and added to the DOM. This is common practice to overcome the design flaws of JavaScript in regard to global variables and provides namespacing semantics. We define our own namespace and the entire framework logic is encapsulated in the `Mobilis` object. The `Mobilis` object implements a function called `extend`, which takes a namespace and an object as arguments. It extends the `Mobilis` namespace with the logic encapsulated in the passed object under `Mobilis.{namespace}`. `Mobilis` holds the entire functionality provided by the framework, such as the BOSH connection management, the event handling and client implementations for *Mobilis Services*. `Mobilis` exists in the `window` object, so in case of a page reload or closing of the browser window the `Mobilis` object does not exist anymore, therefore the design decision in favor of the SPA architecture. Especially our long-lived BOSH connection with the *Mobilis* platform is lost on page reload. *Mobilis* client web applications perform an initial HTTP request for static content served by the web server. Further communication like exchanging XMPP messages with the *Mobilis* platform is handled by the BOSH connection. All content, the client application processes, is transfered over the BOSH-managed connection. Many popular web applications follow the SPA architecture, particularly the popular Google Maps[1] web application.

[1] http://maps.google.com/

5.2. XMPP Web Client

Section 4.3.2 states that the *XMPP Web Client* is supposed to implement the entire BOSH connection handling. In order to provide this, we employ a JavaScript BOSH client library instead of implementing XEP-0124 (Bidirectional-streams Over Synchronous HTTP (BOSH)) and XEP-0206 (XMPP over BOSH) from scratch. The following three JavaScript client libraries for BOSH are available.

- JSJaC - JavaScript Jabber/XMPP Client Library [Str08]
- xmpp4js [Ive08]
- Strophe.js [Mofb]

While JSJaC and xmpp4js are not actively maintained anymore and never really reached sufficient maturity, we found in Strophe.js [Mofb] an excellent BOSH client library. The authors paid special attention to error handling to mask connection errors due to an unreliable network, such as wireless connections. The main class in Strophe is `Strophe.Connection`. This class has methods for establishing the connection to the Connection Manager and for sending stanzas back and forth. The entire library is `47KB` and is included in `mobilis.js`.

The BOSH Connection Manager can either be a stand alone server or be a component of an XMPP server. A separate Connection Manager server is able to establish a connection with any XMPP server, allowing users registered with different XMPP domains to use it. For development and testing we choose the second option. We use the Connection Manager that is incorporated with Openfire[2], the XMPP server that is used for the *Mobilis* platform. Openfire names its Connection Manager *HTTP Binding* and exposes it normally at TCP port 7070. Due to the *same-origin policy* that JavaScript is subject to, the *XMPP Web Client* (retrieved from port 80) is not allowed to connect to a resource on port 7070. There are numerous options to work around this policy. We decided to reverse proxy the *HTTP Binding* address through a URL on port 80, such as `SERVERNAME:80/http-bind`. In our set up we use Apache HTTP Server for handling static content and the module `proxy_http_module` to handle the reverse proxy. Listing 5.2 shows the configuration of an Apache HTTP Server. The *XMPP Web Client* needs the URL as a configuration parameter. Since all other parameters can be found in `Mobilis.core` we store the URL under `Mobilis.core.HTTPBIND`

Additionally, the *XMPP Web Client* is supposed to hold implementations of the four mentioned XEPs. Strophe.js provides an excellent plug-in interface. Implementations for various XEPs are available as Strophe.js community plug-ins [Mofa].

[2] http://www.igniterealtime.org/projects/openfire/

```
ProxyRequests Off
ProxyPass /http-bind http://{XMPPSERVER}:7070/http-bind/
ProxyPassReverse /http-bind http://{XMPPSERVER}:7070/http-bind/
```

Listing 5.2: Reverse proxy Openfire HTTP Binding

We incorporate libraries for XEP-0030 (Service Discovery), XEP-0045 (Multi User Chat) and XEP-0060 (Publish Subscribe) into `mobilis.js`

The BOSH JavaScript client and XEPs implementations are used by the *XMPP Web Client* to provide an API for asynchronous communication with XMPP.

5.3. Mobilis Services Web Client

The aforementioned object `Mobilis` only provides the backbone for our framework. Actual application logic is added via the `extend` function. *Mobilis Core* is implemented in `Mobilis.core` and discussed in detail in the following section.

5.3.1. Configuration

In order to make use of the framework, we need to configure `Mobilis` for each *Mobilis* application. Three parameters are required to connect a web client to the *Mobilis* platform:

- `Mobilis.core.HTTPBIND` specifies the URL of the Connection Manager.
- `Mobilis.core.NS` is an object that hold all namespaces for *Mobilis Services*. This information needs to be provided prior to execution.
- `Mobilis.core.SERVICES` lists all available *Mobilis Services* identified by its namespace and provides the version and JID. Except for the *Coordinator Service*, the data captured here is retrieved by a service discovery after the initial connect. In order to perform the discovery request, the JID and version of the *Coordinator Service* need to be defined prior to execution.

The framework supports three ways for web client developers to set these parameters. The first way is illustrated in Listing 5.3. All parameter are set in the `Mobilis.core` object source code. This option limits all users of this web client to one specific *Mobilis Server* and allows just one Connection Manager. The second option uses HTML5 `localStorage`. The user is required to set these parameters when using the web client for the first time. Parameter values are then stored in the local storage of the web browser and can be retrieved each time the user opens the web

```
1 HTTPBIND: '/http-bind',
2 NS: {
3     COORDINATOR: 'http://mobilis.inf.tu-dresden.de#services/CoordinatorService',
4     USERCONTEXT: 'http://mobilis.inf.tu-dresden.de#services/UserContextService',
5     GROUPING: 'http://mobilis.inf.tu-dresden.de#services/GroupingService',
6 },
7 SERVICES: {
8     'http://mobilis.inf.tu-dresden.de#services/CoordinatorService': {
9         version: '1.0',
10        jid: 'mobilis@mobilis.inf.tu-dresden.de/Coordinator'
11    },
12 }
```

Listing 5.3: Configuration of the Mobilis Client Library

application. For this option the web browser must support `localStorage` which the newest versions of all major browsers do. The third way requires the user to specify the values for these parameters everytime the web application is opened. This way provides independence from a specific *Mobilis Server*, but comes with a decrease of user experience.

`Mobilis.core.NS` is actually an array that holds key/value pairs of the following form {JavaScript namespace}:{XMPP namespace} for each *Mobilis* service. The Mobilis Service Environment allows services to be distributed on different physical machines. The JID (found in `Mobilis.core.SERVICES.{XMPP namespace}.jid`) is the defining parameter to be able to address each service. Also, for each service we need the namespace to be able to interpret the payloads of the exchanged stanzas. The values for JIDs and namespace are set by the service discovery operation that is performed after the BOSH connection is established. Once these values are set, the web client is able to use all collaboration services of the *Mobilis* platform.

Mobilis.core

In this section, we provide a showcase of implementation details for three very common operations *Mobilis* client applications perform. Besides the actual code, we give examples of how to implement these functions into a client web application.

`Mobilis.core.connect` is the single most essential method provided by the framework. In order to exchange collaborative messages with the Mobilis Service Environment every user has to connect to the XMPP network. `connect` uses the `Mobilis.core.HTTPBIND` parameter and connects the user (JID and password) with the XMPP server. `Mobilis.core.connect` takes, in addition to the JID and password, a callback function as the third parameter. The connection process goes through different stages,

such as connecting, authenticating, connected, disconnecting, etc. The callback gets called for each state change. Listing 5.4 shows a typical call of `Mobilis.core.connect`.

```
1 Mobilis.core.connect(
2      'user1@mobilis.inf.tu-dresden.de',
3      'mobilis',
4      function (status) {
5           if (status == Mobilis.core.Status.ERROR) {
6           } else if (status == Mobilis.core.Status.CONNECTING) {
7           } else if (status == Mobilis.core.Status.AUTHENTICATING) {
8           } else if (status == Mobilis.core.Status.CONNECTED) {
9          // send presence, update user inteface, ....
10     }
11 );
```

Listing 5.4: Example call of `Mobilis.core.connect`

`Mobilis.core.mobilisServiceDiscovery` In Section 4.3.1 we stated that there are two main types of interaction with the framework. `mobilisServiceDiscovery` represents the first type that sends a stanza that incorporates the passed arguments and registers callbacks for the response. Listing 5.5 illustrates the call of the service discovery method for a specific service and version. `[Mobilis.core.NS.USERCONTEXT, "1.0"]` are attributes to create an appropriate request stanza for the *User Context Service* version 1.0. Two functions taking the response `<iq/>` as the argument for further processing are defined. The passed object represents the received stanza as an XML DOM element and is easily processable via standard jQuery functions.

```
1 Mobilis.core.mobilisServiceDiscovery(
2    [Mobilis.core.NS.USERCONTEXT, '1.0'],
3    function(resultIQ){
4       // result handler
5    },
6    function(errorIQ){
7       // error handler
8    }
9 );
```

Listing 5.5: Example call of `Mobilis.core.mobilisServiceDiscovery`

In order to provide more detailed insight into the implementation, Listing 5.6 illustrates the `Mobilis.core.mobilisServiceDiscovery` method. In line 4 we use the `$iq` function to create a `Strophe.Builder` for convenient XML handling. Lines 11 to 15 extend the stanza with passed arguments. The stanza is sent in line 17 with two callback functions that process the `resultIQ` and `errorIQ` respectively. The first function (line 19) finds all `mobilisService` elements and sets `Mobilis.core.SERVICES` accordingly. This routine is performed to configure the *Mobilis Services Web Client* during runtime as soon as a connection to an XMPP server has been established.

```
1 mobilisServiceDiscovery: function(attr, resultcallback, errorcallback) {
2     if (!resultcallback) {resultcallback = Mobilis.core.defaultcallback;};
3     if (!errorcallback) {errorcallback = Mobilis.core.defaulterrorback;};
4     var discoiq = $iq({
5         to: Mobilis.core.SERVICES[Mobilis.core.NS.COORDINATOR].jid,
6         type: "get"
7     })
8     .c("serviceDiscovery", {
9         xmlns: Mobilis.core.NS.COORDINATOR
10    });
11    if (attr[0] && attr[0] !== null) {
12        discoiq.c('serviceNamespace').t(attr[0]);
13    };
14    if (attr[1] && attr[1] !== null) {
15        discoiq.up().c('serviceVersion').t(attr[1]);
16    };
17    Mobilis.core.sendIQ(
18        discoiq,
19        function(resultiq) {
20            $(resultiq).find("mobilisService").each(function() {
21                Mobilis.core.SERVICES[$(this).attr('namespace')] =
22                {
23                    'version': $(this).attr('version'),
24                    'jid': $(this).attr('jid')
25                };
26            });
27        },
28        function(erroriq) {}
29    );
30 };
```

Listing 5.6: Implementation of `Mobilis.core.mobilisServiceDiscovery`

`Mobilis.core.addHandler` provides functionality to add a handler to be invoked in case incoming stanzas match certain attributes. This method represents the second type of interaction within the framework, registering a general handler. Listing 5.7 shows how to add a handler for the incoming `<iq/>` with the namespace `http://mobilis.inf.tu-dresden.de#services/GroupingService` and of type `result`.

```
1 Mobilis.core.addHandler(
2     function (stanza){
3       // handler logic
4     },
5   Mobilis.core.NS.GROUPING,
6   'iq','result'
7 );
```

Listing 5.7: Example call of `Mobilis.core.addHandler`

Available Plugins

Table 5.1 lists all available components in the `mobilis.js` library.

Plug-in	Description
`Mobilis.core`	Connect/Disconnect, Send stanzas, Service Discovery
`Mobilis.context`	Publish/Subscribe of context information
`Mobilis.grouping`	Client for the *Mobilis Grouping Service*
`Mobilis.content`	Client for the *Mobilis Repository Service*
`Mobilis.snintegration`	Social Network Integration

Table 5.1.: Available Plug-ins for Mobilis Services Web Client

5.4. Extensibility

Extensibility and reusability are non-functional requirements with high priority in order to accommodate the service oriented nature of the *Mobilis* platform. The Mobilis Service Environment allows introduction of both, new *Generic Mobilis Services* and *App-specific Services*. Web clients for *Mobilis* applications need APIs to communicate with new services. The proposed web gateway framework provides a plug-in adapter to extend the functionality. In this section we introduce the adapter in detail and present examples of usage.

New services appended to the Mobilis Service Environment implicate new protocols and stanza structures. Web client developers are not supposed to handle the creation of stanzas or details of message transfer. Therefore, we need to be able to extend the framework to provide a comfortable abstraction of XMPP communication for the *Mobilis* application logic. Plug-ins are the way to do this. Listing 5.8 illustrates the skeletal structure of a plug-in.

```
1 (function(){
2     var core = {
3         // application logic here
4     };
5     Mobilis.extend("core", core);
6 })();
```

Listing 5.8: Plug-in skeleton

In a plug-in, we can encapsulate the protocol dictated by a new *Mobilis* service. Customized stanzas are created here. In order to exchange the stanzas `Mobilis.core` provides `send` and `sendIQ`.

All available plug-ins are implemented in separate JavaScript files. We supply a makefile in addition to the source code that generates the `mobilis.js` that is employed

in the HTML of the web page. The script produces `mobilis.js` and `mobilis.min.js` which is a minified and compressed version of the source code to reduce the file size for more efficient page load. With this process, developers can choose which plug-ins they need and want to include in web client. All libraries included in `mobilis.js` are pure JavaScript. The overwhelming majority of modern websites use JavaScript, and all modern web browsers on desktops, game consoles, tablets and smartphones include JavaScript interpreters, making JavaScript the most ubiquitous programming language in history.

5.5. Summary

In this chapter we provided insights into the implementation of the framework. We presented the architecture of *Mobilis* web clients and covered the *XMPP Web Client* implementation. For the *Mobilis Services Web Client* we showcased three commonly used methods and the respective usage. We wrapped this chapter up with a description of extensibility features of our framework. In Chapter 4 we mentioned components representing client implementations for *Mobilis Services*. Table 5.2 illustrates which plug-ins were implemented to their full extent in the course of this work.

Plug-in	**Implemented**
`Mobilis.core`	✓
`Mobilis.context`	✓
`Mobilis.grouping`	✓
`Mobilis.xhunt` (Chapter 6)	✓
`Mobilis.content`	—
`Mobilis.snintegration`	—

Table 5.2.: Fully implemented plug-ins

Unfortunately `Mobilis.content` is not fully implemented yet. Realizing this component, though, will not differ from other plug-in implementations. The same line of action with respect to stanza creation and handler registration can be taken. `Mobilis.snintegration` was originally considered (conceptual design) to be an individual component in the *Mobilis Services Web Client*. However, during implementation it became clear that social network integration is part of the *Mobilis* application specific logic. No encapsulation is necessary, nor would it be advantageous. Existing frameworks to connect to various APIs available in JavaScript are to be employed.

6. Evaluation

In this chapter we evaluate the concept and implementation presented in this work. We start with a presentation of QUnit Test results to show the browser compatibility of our framework. In order to demonstrate the functionality, we use it in a real-world use case. We developed web clients based on our framework for two *Mobilis* applications. First, we present a spectator web application for *MobilisXHunt*. The second web client is optimized for mobile web browsers with a touch-based interface. The web application facilitates a client for *MobilisGroups*. After a brief performance evaluation, the last section includes an error handling evaluation of our framework with special attention to connection and therefore communication errors.

6.1. Browser Support

We introduced this work with an illustration of the present situation of mobile social application development. Platform fragmentation and the consequential bad ratio between development cost and reach, make native application development not thrifty. Web-based mobile application development is considered a serious alternative with many benefits, such as platform independent development and enormous reach. The proposed framework lets application developers of mobile social software enjoy these two amenities of web-based development. While the following sections evaluate the functionalities of the framework, we present in this section platform independence.

NONF-2 and NONF-3 require A-grade compatibility for both major desktop and mobile browsers. In order to evaluate the compatibility of our framework, we provide a test application that incorporates all distinct types of operation. We built the application on the jQuery QUnit test suite.

Figure 6.1 illustrates the test application which tested the entire connection cycle to an XMPP network and performs a service discovery request to the *Mobilis Server*. In the course of the evaluation we performed this QUnit for all major desktop and mobile browsers. Appendix A provides a comprehensive list of test results while Table 6.1 summarizes the results. Our framework features complete browser and platform independence and facilitates an excellent basis for real-time collaboration client applications. Safari iOS support is particularly valuable unlocking the iPhone and iPad for *Mobilis* client applications.

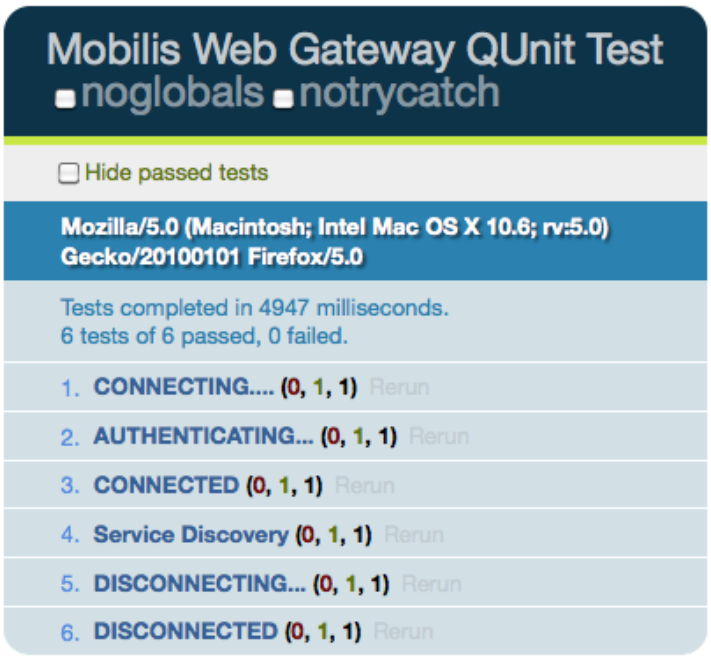

Figure 6.1.: QUnit Test: Firefox 5, Mac OS X 10.6

We now move on to evaluate the real-time capabilities and generic nature of the framework by developing two collaborative web applications.

6.2. MobilisXHunt Spectator Web Application

MobilisXHunt is the first *Mobilis* application that we want to augment with a web based client application. Kiefner introduces this location-based game in [Kie11]. *MobilisXHunt* implements the famous board game *Scotland Yard* [Rav] as a real-world game in which the players use mobile devices to participate in this turn-based game. The main goal is to hunt the villain (Mister X) as a team by moving along predetermined paths (bus, train, taxi) until at least one player is in the same location as the villain, whose route is normally kept secret.

Kiefner presents the *App-specific Service* and an Android-based client application for *MobilisXHunt*. We use our framework to develop a web-based spectator application to monitor the game action. As one of the applications based on the *Mobilis* platform, communication in *MobilisXHunt* is realized via XMPP messages. Therefore, location updates, group chat data and other game related information is transported in stanzas. Distribution of information is subject to limitations based on the roles of users. For example, Mister X is able to see the location of all participants and their movement, but keeps his own location secret. In order to provide

Browser	A-grade Support
Chrome	✓
Firefox	✓
Safari	✓
Internet Explorer	✓
Opera	✓
Safari Mobile (iPhone)	✓
Safari Mobile (iPad)	✓
Blackberry Playbook	✓
Opera Mini	✓
HP WebOS	✓
Android Phone	✓
Android Tablet	✓

Table 6.1.: Browser Support

complete monitoring functionality the *MobilisXHunt Service* provides a spectator user role. All game related information is forwarded to spectators. The web client presented in this section provides a user interface for monitoring the game action. The main goal of the web client is to place the participants on the map in real-time, display the multi-user chat and provide basic information about the running game.

MobilisXHunt employs the *App-specific Service MobilisXHunt Service* [Kie11] which implements an application specific protocol. Custom stanzas are exchanged to initiate a game, broadcast turn-based actions of participants and control the game flow. The spectator web application must be able to understand and process these customized stanzas. In the following section we describe the *MobilisXHunt* web client plug-in and its features. By extending `Mobilis` with the functionality provided by `Mobilis.xhunt` we demonstrate the extensibility features of our framework.

6.2.1. Mobilis.xhunt

Listing 6.1 illustrates a small snippet of the *MobilisXHunt* plug-in. We demonstrate two functions.

```
1 (function () {
2     var xhunt = {
3         //...
4         joinGame: function (attr, resultcallback, errcallback, timeoutcallback) {
5             //..
6             Mobilis.connection.sendIQ(stanza, callback, errback);
7         },
8         addRoundStatsHandler: function (callback) {
```

```
 9            //..
10            Mobilis.connection.addHandler(handler, 'mobilisxhunt:iq:roundstatus', '
                  iq', 'set')
11        },
12    };
13    Mobilis.extend("xhunt", xhunt);
14 })();
```

Listing 6.1: `Mobilis.xhunt`

- `joinGame` creates a customized stanza following the protocol as described in [Kie11], sends it and registers various handlers. This reoccurring procedure of interacting with the framework is familiar and described in Chapter 5
- `addRoundStatsHandler` is an addHandler function for incoming *RoundStatsIQ* stanzas [Kie11]. *RoundStatsIQ* are broadcasted to all participants in order to inform about the last turn's actions.

`Mobilis.xhunt` uses `Mobilis.core` functionality and provides a simple API to developers implementing the *MobilisXHunt* spectator web application. For completion, Table 6.2 lists all necessary methods of `Mobilis.xhunt` in order to send outgoing and handle incoming XMPP messages.

Function	**Parameter & Description**
JoinGame	User joins as spectators
GameDetails	Information about the running game
UsedTickets	Status of ticket quota
Snapshot	Information update on game status
PlayerExit	Spectator exit the game
AddLocationHandler	Location update requests from the server
AddPlayersHandler	Updates to status of players
AddPlayerExitGameHandler	Players leaving games
AddStartRoundHandler	The "start round" broadcast
AddRoundStatsHandler	Round stats broadcast

Table 6.2.: Available methods of `Mobilis.xhunt`

6.2.2. xhunt.js

The JavaScript file `xhunt.js` implements the application logic for the *MobilisXHunt* web client, including the UI. Processing incoming data from the game and displaying it is also part of `xhunt.js`. We use jQuery for DOM injection and user event handling.

Additionally, we use the Google Maps API to include maps and display location-based information, such as the position of all players. *Gmap 3* [Dem11] is a great jQuery plug-in that augments Google Maps with advanced features. We use it to manipulate the maps.

Figure 6.2 shows a screen shot of the *MobilisXHunt* spectator web application. In the main area we find maps provided by Google Maps and an XML data set as an overlay indicating the pre-determined routes. The right hand site is split into halves. The upper half shows a list of all participants and respective information, while the lower half incorporates a groupchat window for ingame chat.

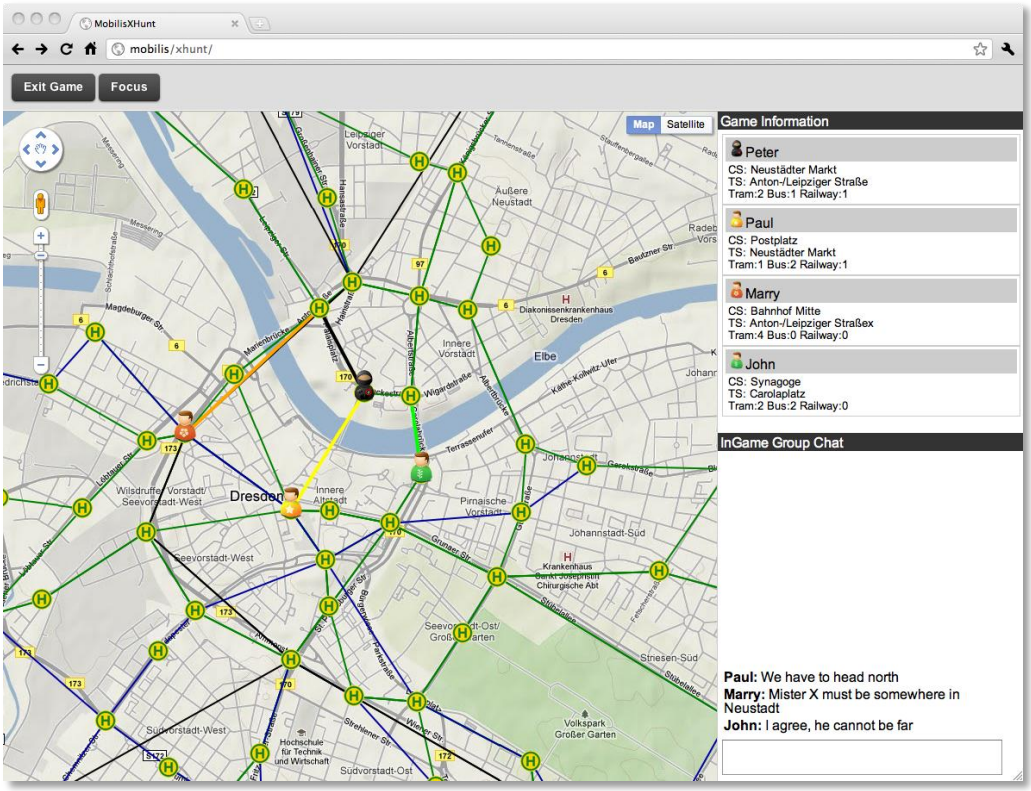

Figure 6.2.: Screenshot MobilisXHunt spectator web application

6.2.3. Summary

The implementation of the *MobilisXHunt* spectator web application proves the extensibility of the framework. We introduced a new *App-specific Service* and provided a web client component that was integrated into the framework easily. Clearly defined interfaces are used to interact with core functionality. Real-time game related data is displayed with low latency, while the entire web application is browser-based

and uses nothing but native web technologies. The display and interaction possibilities of a desktop browser are exploited to provide a rich UI for monitoring the game action.

6.3. MobilisGroups Mobile Web client

The second web client we built on our framework is a *MobilisGroups* client. *MobilisGroups* makes use of the *Grouping Service* in the Mobilis Service Environment to provide location- and time-based grouping functionality. Users are able to create, browse and join groups. Multi-user chat is supported between users of the same group and enables, together with information about group membership of friends, social networking capabilities. The service considers two pieces of user context information to determine access privileges, location and time. In order to exploit the full potential of location data, mobile devices must be used. We decided therefore to develop a mobile web client for *MobilisGroups* to demonstrate that the proposed framework opens up the *Mobilis* platform for many more mobile operating systems via web-based client applications. A *MobilisGroups* web client that provides real-time social networking functionality proves that the web stack is an appropriate technology for platform independent client development for the *Mobilis* platform. Mobile browsers differ from desktop browsers in two characteristics:

1. screen size
2. interaction mechanism

To accommodate these differences, the user interface of a mobile web application has to be significantly different compared to a regular web application. Three major frameworks are available to develop a touch based UI that is optimized for small screens. JQ Touch and jQuery Mobile are very capable JavaScript libraries with special attention to browser compatibility. However jQuery Mobile has not yet reached sufficient maturity and JQ Touch is no longer actively maintained. Therefore, we decided to use Sencha Touch to develop a *MobilisGroups* web client that is optimized for mobile browsers. Sencha Touch provides a comprehensive framework for high performance web applications and follows the concept of starting with an empty HTML `<body/>` and injecting every UI component dynamically via JavaScript.

Collaborative messages are exchanged with the *Mobilis Grouping Service* using `Mobilis.grouping`. No *App-specific Service* is used for *MobilisGroups*. Including `mobilis.js` in the HTML source is sufficient to develop the web client. The deployment of the framework with no customizations (just configuration) shows the generic nature and reusability of the design.

Figure 6.3 shows two screenshots of the *MobilisGroups* web client. We find the navigation bar at the bottom of the screen. Besides the map view that places all

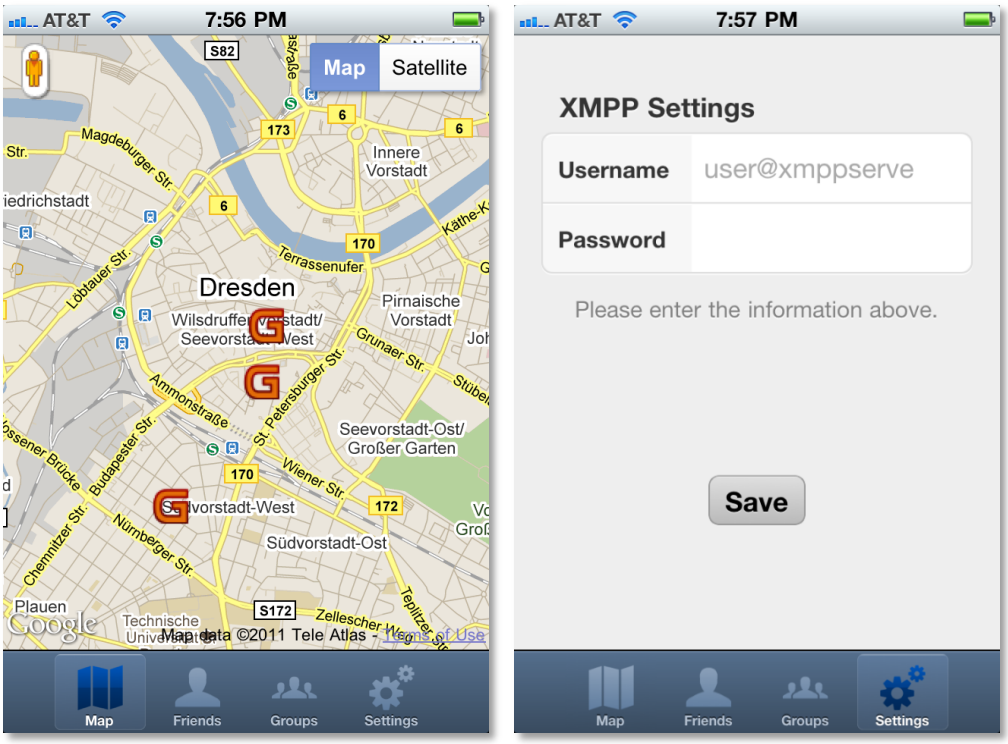

Figure 6.3.: Screenshot MobilisGroups web client

available groups on the map, the interface provides views for friends and groups connected to the user. Settings are available in the last view.

6.4. Performance

In order to evaluate the performance of the framework, we have to consider the architecture of the underlying communication system. We use a BOSH managed connection to exchange collaboration messages with the *Mobilis* platform. The Connection Manager acts as a proxy for the web client towards the XMPP network. Communication, hence performance characteristics, past the Connection Manager is entirely XMPP based and independent from our framework. Latency up to this point is responsibility of XMPP. Our architecture adds minimal latency to native XMPP messages. The Connection Manager always holds open one HTTP request of the web client and answers it with incoming stanzas. Added latency is of dimensions similar to an HTTP round trip. Figure 6.4 illustrates the performance characteristics of the test application. The entire life cycle of a BOSH managed connection plus a service discovery request took 985 milliseconds. Performing the service discovery took about 55 milliseconds, including building the stanza, exchanging it and displaying

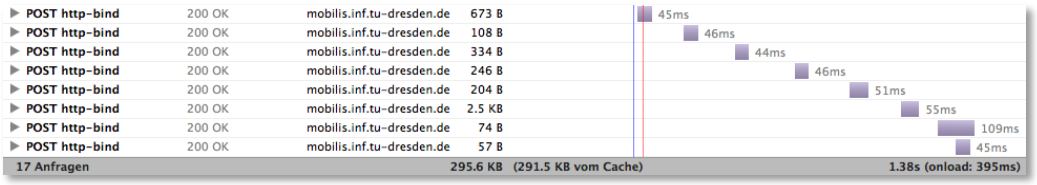

Figure 6.4.: Performance evaluation

the result. Appendix B includes more detailed screenshots of the performance test illustrating the individual HTTP roundtrips and their payload.

It is safe to say that BOSH does not add noticeable latency compared to native XMPP communication. Therefore the user experiences the same real-time interaction as with client applications directly connected via TCP.

6.5. Error Handling

In this Section we evaluate the tolerance of the frameworks to connection errors. XMPP usually runs on a TCP connection which is reliable in nature. However in networks with intermittent connectivity, such as mobile internet, the TCP connection can get interrupted. This forces the client to re-establish the XMPP connection, which includes multiple round trips for encryption, authentication, resource binding, etc.

XMPP over BOSH provides a viable solution for this category of problems. The tunneling of XMPP message exchanges through multiple HTTP request/response cycles masks connection errors of the underlying network within the framework of certain timeouts. The Connection Manager holds any HTTP request for 60 seconds unless some data arrives from the XMPP network to be forwarded to the client. All connection errors that are resolved within this timeout are no harm to the BOSH-managed connection whatsoever. Another advantage of BOSH-managed connections is the Connection Manager's ability to cache and resend HTTP responses in case they fail to reach the client. Therefore, data is never lost even on unreliable networks.

Regular long-lived TCP connections need frequent keepalive packets. Holding this connection open consumes power, which is unfortunate for mobile devices. With BOSH, no energy is consumed by long-polled HTTP requests.

6.6. Summary

This chapter presented an evaluation of the proposed framework. We showcased web clients for two different *Mobilis* applications, MobilisXHunt and MobilisGroups.

Both web clients are based on the framework designed in this work. They follow the architecture described in Section 5.1, while all dynamic content originating in the XMPP-base Mobilis Service Environment is provided by our framework. The two web clients differ in just the user interface and application specific logic, such as interaction event handling and displaying collaborative data. Additionally, we provided an evaluation of browser support, performance characteristics and connection error handling. In all categories the proposed framework excels. With this evaluation we proved the generic nature and real-time capabilities of our framework.

7. Conclusion

This chapter sums up results of this work and provides an outlook.

7.1. Summary

In the beginning of this thesis, we stated that the web technology stack is a viable option to develop platform independent applications. We especially discussed mobile social software and identified real-time communication capabilities as a challenge in web application development. We claimed to provide a framework for web application development that facilitates bi-directional exchange of collaborative messages with the *Mobilis* platform in real-time. In order to provide the fundamental knowledge necessary to comprehend the conceptual design of the framework, we introduced the *eXtensible Messaging and Presence Protocol* and various real-time web technologies. These technologies composed the basis of discussion for the design decisions in this work. After a brief introduction of the *Mobilis* platform, we examined the state-of-the-art in research and industry. A comprehensive requirements analysis proceeded the main part of this work, the conceptual design. Before presenting the architecture of the framework, we conducted an assessment of real-time web technologies with regard to our requirements and concluded that *Bidirectional-streams Over Synchronous HTTP* is the right technology for our framework. With the use of XEP-0206 (XMPP over BOSH), we connected the browser to the XMPP-based *Mobilis* platform. This enabled an event-driven web client architecture to produce outgoing and handle incoming collaborative XMPP messages. We presented the *XMPP Web Client* as the component responsible for real-time communication and stanza handling and the *Mobilis Services Web Client* that implements protocols specific to *Mobilis Services*. For implementation details, we provided code snippets that illustrate the even-driven nature of the framework. Additionally the extensible character of the plug-in-based design was discussed. In order to evaluate the concept we implemented web clients for *MobilisXHunt* and *MobilisGroups* respectively. We also performed browser compatibility and performance tests.

7.2. Outlook

Mobile web application development has a promising future. Like every aspect of mobile computing, the capabilities of the mobile web browser is improving rapidly. Brilliant innovation in the introduction of HTML5, such as canvas, webGL, generational garbage collection in JavaScript or Webworkers and WebSockets, make the web stack a powerful and high-performance application platform.

Business logic is traditionally located on back-end application servers and Ajax enables highly interactive user interfaces to disguise the fact that the browser is just a thin client rendering the web page. In the near future however, web browsers are going to be powerful enough to serve as a runtime for extensive applications and business logic while the back-end servers will just provid the data.

Chromium OS[1] is a first attempt in this direction and demonstrates the capabilities of the V8 JavaScript engine. For the mobile world this trend grows even faster. HP[2] and RIM base their entire native application concept on the web stack and let web applications break out of the web browser sandbox. Native webOS and Blackberry OS applications are in-fact web applications. PhoneGab[3] and Appcelerator Titanium[4] provide that same concept of breaking out of browser constraints for all mobile platforms. Applications are developed using web technologies and wrapped in a native application shell exposing device and OS specific functionalities. Although the industry has to figure out a viable deployment model for these kinds of web applications to address the obvious security issues (File system access, etc.), there is clearly a noticeable trend towards web application development.

For the *Mobilis Project*, web application development is a great way to extend the reach to customers and overcome platform and device fragmentation. The design decision to use BOSH for bridging XMPP and the web is the smartest for the time being. With the evolution of WebSockets and better browsers support, however, future systems should incorporate newer web technologies. Scalable event-driven systems solely based on JavaScript (both client- and server-side) and XMPP over WebSockets is possible. For example, node.js[5] as a server-side JavaScript platform receives immense attention in the web engineering community. Projects such as SocketStream[6] emerge rapidly. With these technologies the web is becoming a true real-time platform.

[1] http://www.chromium.org/
[2] http://www.hp.com
[3] http://www.phonegap.com/
[4] http://www.appcelerator.com
[5] http://nodejs.org/
[6] https://github.com/socketstream/socketstream

As of now, the web stack might not be able to compete in some categories with native applications, such as 3D graphics and video games. However, for most use cases, web application development is a serious alternative to native applications.

Bibliography

[BCD+09] Nilanjan Banerjee, Dipanjan Chakraborty, Koustuv Dasgupta, Sumit Mittal, Seema Nagar, and Saguna. R-U-in? - Exploiting Rich Presence and Converged Communications for Next-Generation Activity-Oriented Social Networking. In *Proceedings of the 2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*, MDM '09, pages 222–231, Washington, DC, USA, 2009. IEEE Computer Society.

[BHT97] Richard Bentley, Thilo Horstmann, and Jonathan Trevor. The World Wide Web as Enabling Technology for CSCW: The Case of BSCW. *Comput. Supported Coop. Work*, 6:111–134, May 1997.

[BLH01] Dominik Buszko, Wei-Hsing (Dan) Lee, and Abdelsalam (Sumi) Helal. Decentralized ad-hoc groupware API and framework for mobile collaboration. In *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*, GROUP '01, pages 5–14, New York, NY, USA, 2001. ACM.

[com11] comScore. The 2010 mobile year in review. Press Release, February 2011.

[Dem11] Jean-Baptiste Demonte. Gmap3, 2011. `http://gmap3.net/`.

[DGI11] Rabih Dagher, Cristian Gadea, and Tropper Dan R. Ionescu, Bogdanand Ionescu. Collaborative Web-Based Mapping of Real-Time Flight Simulator and Sensor Data. *OSGeo Journal*, Volume 8:48–52, 2011.

[DR08] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[Dra11] LLC Drakontas. Smyle, 2011. `http://smylenow.com/`.

[DRS00] M. Day, J. Rosenberg, and H. Sugano. A Model for Presence and Instant Messaging. RFC 2778 (Informational), February 2000.

[EHM+07] R. Eatmon, J. Hildebrand, J. Miller, T. Muldowney, and P. Saint-Andre. XEP-0004: Data forms. Technical report, XMPP Standards Foundation, 2007.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266.

[FZ98] Michael Franklin and Stan Zdonik. Data in your face: Push Technology in Perspective. *SIGMOD Rec.*, 27:516–519, June 1998.

[GLG11] Carl A. Gutwin, Michael Lippold, and T. C. Nicholas Graham. Real-time Groupware in the Browser: Testing the Performance of Web-based Networking. In *Proceedings of the ACM 2011 conference on Computer supported cooperative work*, CSCW '11, pages 167–176, New York, NY, USA, 2011. ACM.

[GNS+11] L. Gutierrez, I. Nikolaidis, E. Stroulia, S. Gouglas, G. Rockwell, and M. Boechler, P. andCarbonaro. fAR-PLAY: A framework to develop augmented/alternate reality games. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Confere*, 2011.

[Goo09] Google. Google Wave, 2009. `http://wave.google.com`.

[Goo11a] Google. Google Docs, 2011. `http://docs.google.com`.

[Goo11b] Google. Google Plus, 2011. `http://plus.google.com`.

[Gro11] Vodafone Group. Onesocialweb, 2011. `http://onesocialweb.org/`.

[Her09] D. Hering. Entwicklung eines Dienstes für Real-time Collaborative Editing für die Mobilis-Plattform. Master's thesis, Technische Universität Dresden, Fakultät Informatik, September 2009.

[Hic09] I Hickson. The Web Socket protocol. IETF, Standards Track, April 2009.

[Hic10] I Hickson. The Websocket API. W3C, Editor's Draft, 2010.

[Hic11] Ian Hickson. HTML5 A vocabulary and assciated APIs for HTML and XHTML. W3C Working Draft, May 2011. `http://www.w3.org/TR/html5/`.

[HJ99] Manfred Hauswirth and Mehdi Jazayeri. A component and communication model for push systems. *SIGSOFT Softw. Eng. Notes*, 24:20–38, October 1999.

[HMESA08] J. Hildebrand, P. Millard, R. Eatmon, and P. Saint-Andre. XEP-0030: Service discovery. Technical report, XMPP Standards Foundation, 2008.

[Inc11] Convore Inc. Convore, 2011. `https://convore.com/`.

[Ive08] Harlan Iverson. xmpp4js, 2008. `http://xmpp4js.sourceforge.net/`.

[Kie11] Danny Kiefner. Runden-basierte Orts-bezogene Spiele auf mobilen Endgeräten. Master's thesis, Technische Universität Dresden, Fakultät Informatik, April 2011.

[Lüb10] Robert Lübke. Mobile Social Networking - Location Based Group Formation. Master's thesis, Technische Universität Dresden, Fakultät Informatik, Oktober 2010.

[Lüb11] Robert Lübke. Ein framework zur entwicklung mobiler social software auf basis von android. Thesis, March 2011.

[LSS11] Robert Lübke, Daniel Schuster, and Alexander Schill. Mobilisgroups: Location-based group formation in Mobile Social Networks. In *PerCom Workshops*, pages 502–507, 2011.

[LT09] Soren Lassen and Sam Thorogood. *Google Wave Federation Architecture*. Google, Inc., 2009. `http://wave-protocol.googlecode.com/hg/whitepapers/google-wave-architecture/google-wave-architecture.html`.

[Ltd09] Storm Ideas Ltd. Collab, 2009. `http://colaab.com/`.

[Mofa] J. Moffitt. Strophe community plug-ins. `https://github.com/metajack/strophejs-plugins.git`.

[Mofb] Jack Moffitt. *Strophe.JS*. `http://code.stanziq.com/strophe`.

[Mof10] J. Moffit. An XMPP Sub-protocol for WebSocket. Technical report, Internet Engineering Task Force, 2010.

[MW10] Stephen Mogan and Weigang Wang. The Impact of Web 2.0 Developments on Real-Time Groupware. In *Proceedings of the 2010 IEEE Second International Conference on Social Computing*, SOCIALCOM '10, pages 534–539, Washington, DC, USA, 2010. IEEE Computer Society.

[MZ06] A. Melnikov and K. Zeilenga. Simple Authentication and Security Layer (SASL). RFC 4422 (Proposed Standard), June 2006.

[Ora10] Oracle. Oracle Webcenter Real-time Collaboration, 2010. http://www.oracle.com/technetwork/middleware/wc-real-time-collaboration/overview/webcenter-real-time-collaboration-165168.pdf.

[PG11] Christy Pettey and Laurence Goasduff. Gartner identifies 10 consumer mobile applications to watch in 2012. Press Release, 2011. http://www.gartner.com/it/page.jsp?id=1544815.

[Poh09] Mikko Pohja. Server Push with Instant Messaging. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 653–658, New York, NY, USA, 2009. ACM.

[PSA10] Ian Paterson and Peter Saint-Andre. XEP-0206: XMPP over BOSH. Technical report, XMPP Standards Foundation, 2010.

[PSSAM10] Ian Paterson, Dave Smith, Peter Saint-Andre, and Jack Moffitt. XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH). Technical report, XMPP Standards Foundation, 2010.

[Rav] Ravensburger. Scotland Yard. http://www.ravensburger.com/usa/products/games/family_games/scotland_yard_26117/index.html.

[RCBH10] Mattias Rost, Henriette Cramer, Nicolas Belloni, and Lars Erik Holmquist. Geolocation in the Mobile Web Browser. In *Proceedings of the 12th ACM international conference adjunct papers on Ubiquitous computing*, Ubicomp '10, pages 423–424, New York, NY, USA, 2010. ACM.

[RG96] Mark Roseman and Saul Greenberg. Building real-time groupware with groupkit, a groupware toolkit. *ACM Trans. Comput.-Hum. Interact.*, 3:66–106, March 1996.

[Rus] A. Russel. Comet: Low latency data for the browser. alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/. March 3, 2006. Web. March 15, 2011.

[RWDN07] A. Russell, G. Wilkins, D. Davis, and M. Nesbitt. The Bayeux Specification - Bayeux Protocol. Standards Track, 2007. http://svn.cometd.com/trunk/bayeux/bayeux.html.

[SA04a] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920 (Proposed Standard), October 2004.

[SA04b] P. Saint-Andre. Extensible Messaging and Presence Protocol (xmpp): Instant Messaging and Presence. RFC 3921 (Proposed Standard), October 2004.

[SAP11] SAP. Streamwork, 2011. http://www.sapstreamwork.com/.

[SKS+09] Daniel Schuster, I. Koren, T. Springer, D. Hering, B. Söllner, M. Endler, and A. Schill. *Handbook of Research on Mobile Software Engineering: Design, Implementation and Emergent Applications*, chapter Creating Applications for Real-time Collaboration with XMPP and Android on Mobile Devices. IGI Global, 2009.

[Söl09] Benjamin Söllner. XMPP-based Media Sharing for Mobile Collaboration with Android Phones. Master's thesis, Technische Universität Dresden, Fakultät Informatik, Oktober 2009.

[Sof11] Jive Software. Jive Engage Platform, 2011. http://www.jivesoftware.com/products.

[SSB+08] T. Springer, D. Schuster, I. Braun, J. Janeiro, M. Endler, and A. A. F. Loureiro. A flexible architecture for mobile collaboration services. In *Companion '08: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, pages 118–120. New York, NY, USA, 2008. ACM.

[SSS10] D. Schuster, T. Springer, and A. Schill. Service-based Development of Mobile Real-time Collaboration Applications for Social Networks. In *PerCom Workshops*, pages 232–237, 2010.

[Str08] Stefan Strigler. JSJaC - JavaScript Jabber/XMPP Client Library, 2008. http://blog.jwchat.org/jsjac.

[The10] The jQuery Project. jQuery, 2010. http://jquery.com/.

[The11] The Dojo Foundation. Open Cooperative Web Framework, 2011. http://opencoweb.org/.

[Twi10] Twiddla. Twiddla, 2010. http://www.twiddla.com/.

[Wan08] Weigang Wang. Powermeeting on common ground: web based synchronous groupware with rich user experience. In *Proceedings of the hypertext 2008 workshop on Collaboration and collective intelligence*, WebScience '08, pages 35–39, New York, NY, USA, 2008. ACM.

[WML10] David Wang, Alex Mah, and Soren Lassen. Google Wave Operational Transformation, 2010. http://wave-protocol.googlecode.com/hg/whitepapers/operational-transform/operational-transform.html.

[YAH11] YAHOO! Graded browser support, 2011. http://developer.yahoo.com/yui/articles/gbs/index.html.

[YCW09] Chuan Yue, Zi Chu, and Haining Wang. RCB: A simple and practical framework for real-time collaborative browsing. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 29–29, Berkeley, CA, USA, 2009. USENIX Association.

[YI03] Muhammad Younas and Rahat Iqbal. Developing Collaborative Editing Applications using Web Services. In *Proc. 5th Int. Workshop on Collaborative Editing*, 2003.

List of Figures

List of Tables

Listings

Acronyms

Notation	Description
AARG	*Augmented/Alternate Reality Games*. 24
API	*Application Programming Interface*. 2, 15, 20, 25, 27, 31, 35, 37–39, 42, 43, 47, 53, 58, 62, 63, 68
BOSH	*Bidirectional-streams Over Synchronous HTTP*. 18, 19, 27, 30, 36, 39–43, 45, 47–50, 55–59, 71–73, 75, 76
CPU	*Central Processing Unit*. 15
CSS	*Cascading Style Sheets*. 33, 42, 55
DNS	*Domain Name System*. 7
DOM	*Document Object Model*. 17, 55, 56, 60, 68
HTML	*Hypertext Markup Language*. 16, 33, 42, 55, 63, 70
HTTP	*Hypertext Transfer Protocol*. 9, 13–20, 24, 26, 27, 30, 35–40, 42, 45, 46, 55, 56, 71–73
IETF	*Internet Engineering Task Force*. 5, 6, 9, 20, 39
IM	*Instant Message*. 6–8
IP	*Internet Protocol*. 7, 13, 17, 39, 41
IQ	Info/Query. 8–10, 48, 49
IRC	*Internet Relay Chat*. 11
JID	*Jabber ID*. 6, 7, 10, 11, 21, 22, 48, 50–52, 58, 59
JMS	*Java Messaging Service*. 25
JVM	*Java Virtual Machine*. 15
LTE	Long Term Evolution. 1
MIME	*Multipurpose Internet Mail Extensions*. 16
MUC	*Multi User Chat*. 11, 23, 32

Notation	Description
MXA	*Mobilis XMPP on Android*. 35
RFC	*Request for Proposal*. 5
RIM	Research In Motion. 1
RPC	*Remote Procedure Call*. 45
SASL	*Simple Authentication and Security Layer*. 7, 34, 40, 46, 48
SPA	*Single-page Application*. 55, 56
TCP	*Transport Control Protocol*. 6, 7, 13, 15, 17–20, 35, 37, 39, 41, 42, 57, 72
TLS	*Transport Layer Security*. 7, 34, 40, 46, 48
UI	user interface. 1, 12, 15, 43, 55, 67, 68, 70, 76
UID	*Unique Identification*. 52, 53
URL	*Uniform Resource Locater*. 40, 46, 57, 58
W3C	*World Wide Web Consortium*. 2, 20
XEP	*XMPP Extension Protocol*. 5, 8–11, 18, 21, 30, 38, 39, 41, 47–49, 52, 58
XHR	*XMLHttpRequest*. 16, 17, 26, 30, 45
XML	*eXtensible Markup Language*. 7–10, 19, 26, 39, 40, 46, 48, 50, 60, 69
XMPP	*eXtensible Messaging and Presence Protocol*. 1, 5–11, 15, 18, 19, 21–23, 25–28, 30, 31, 35–43, 45, 47, 48, 52, 53, 55–60, 62, 63, 65, 66, 68, 71–73, 75, 76
XSF	*XMPP Standards Foundation*. 5, 10

A. Mobilis Web Gateway QUnit Tests

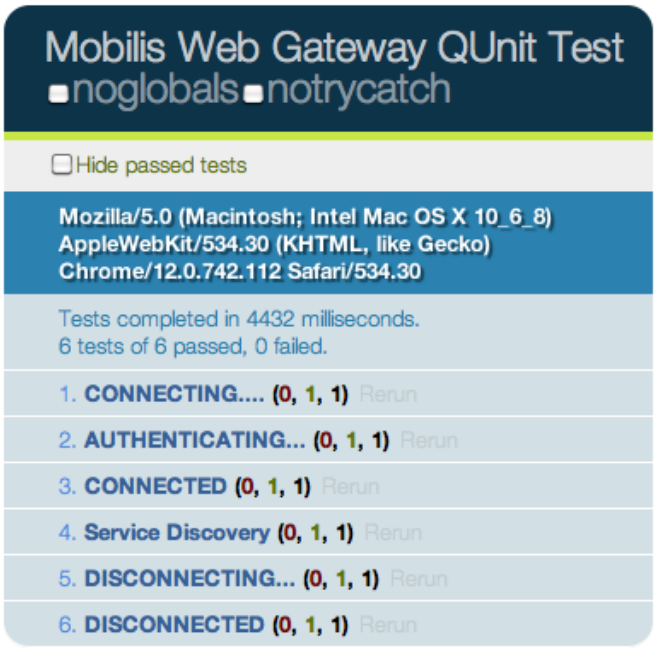

Figure A.1.: QUnit Test: Chrome 12, Mac OS X 10.6

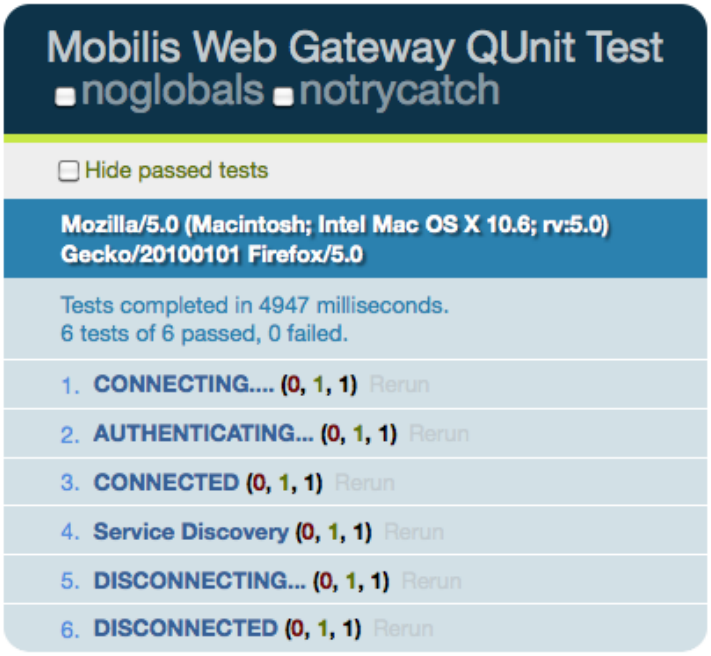

Figure A.2.: QUnit Test: Firefox 5, Mac OS X 10.6

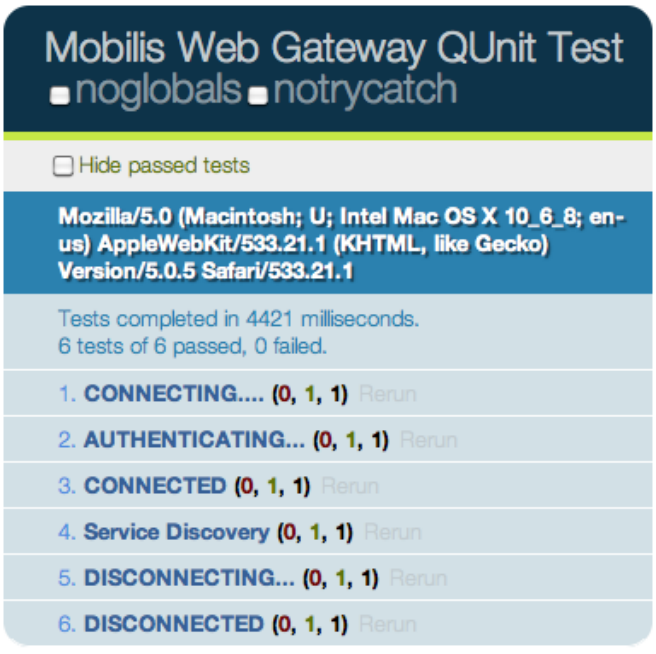

Figure A.3.: QUnit Test: Safari 5, Mac OS X 10.6

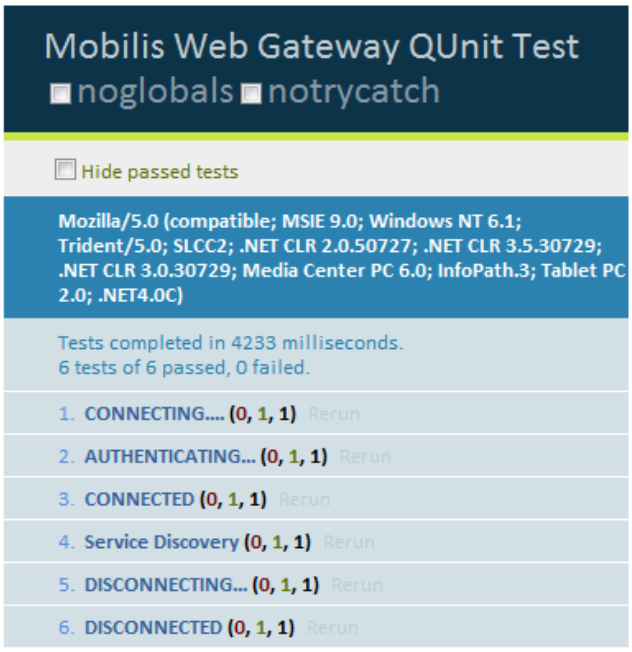

Figure A.4.: QUnit Test: Internet Explorer 9, Windows 7

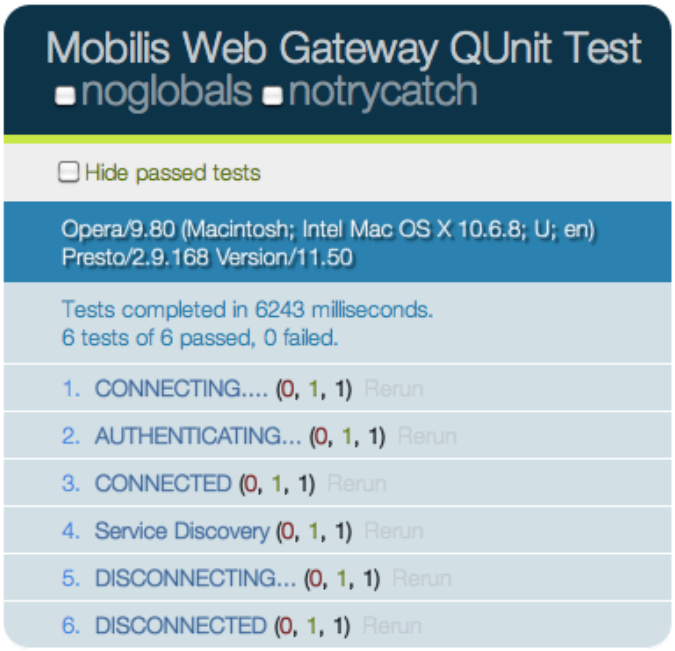

Figure A.5.: QUnit Test: Opera 9, Mac OS X 10.6

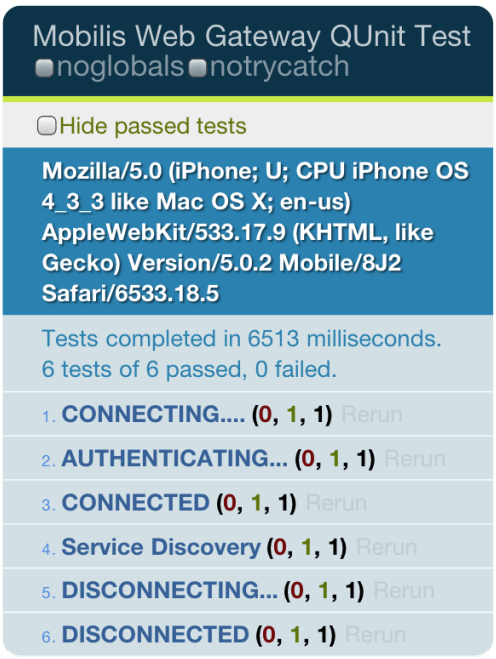

Figure A.6.: QUnit Test: Safari Mobile 5, iPhone OS 4

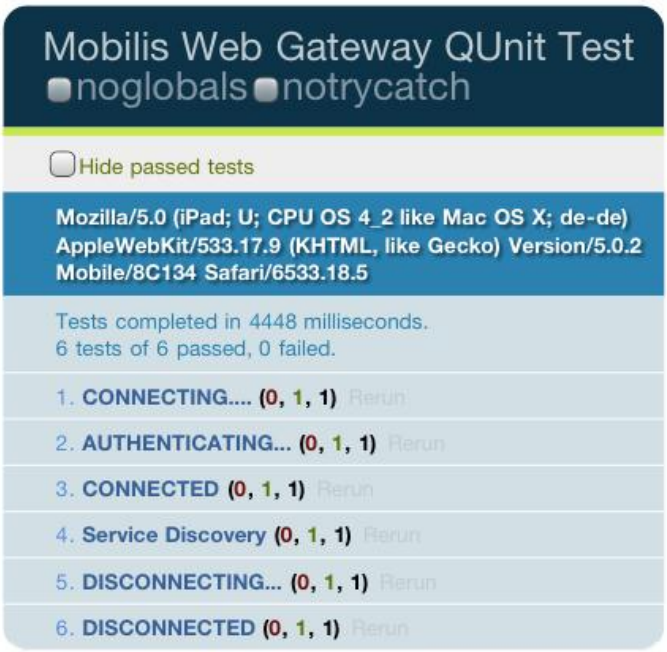

Figure A.7.: QUnit Test: Safari Mobile 5, iOS 4

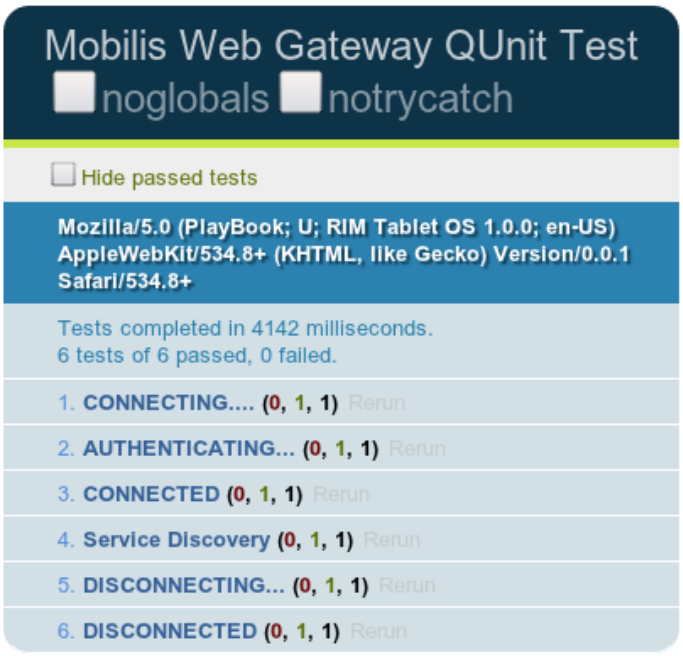

Figure A.8.: QUnit Test: Playbook Browser, RIM Tablet OS 1

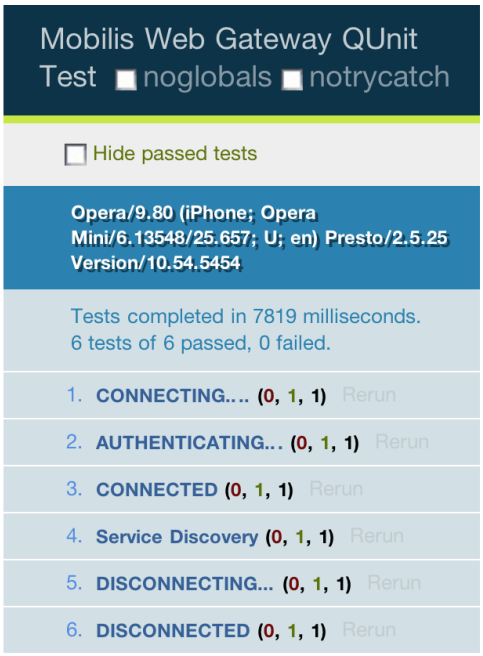

Figure A.9.: QUnit Test: Opera Mini, iPhone OS4 4

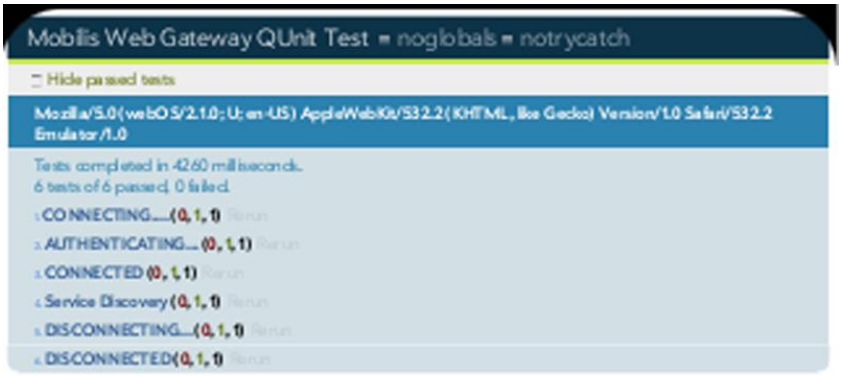

Figure A.10.: QUnit Test: WebOS Browser, webOS 5

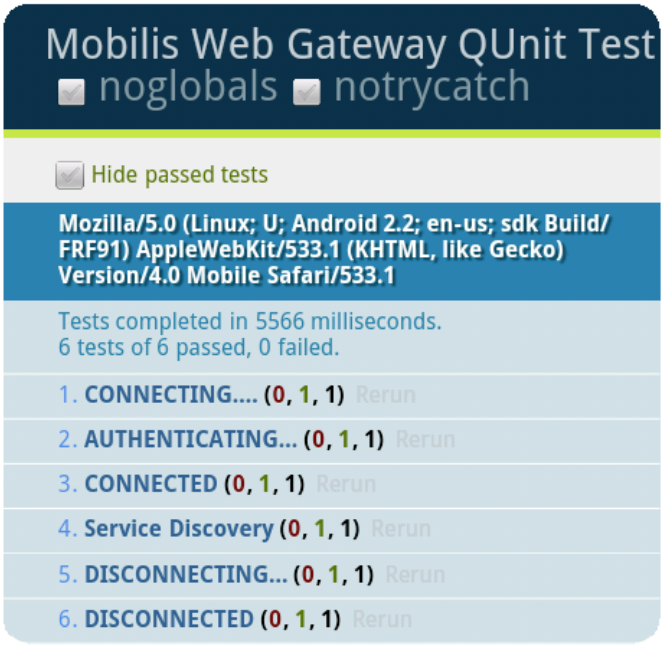

Figure A.11.: QUnit Test: Android Browser, Android 2.2

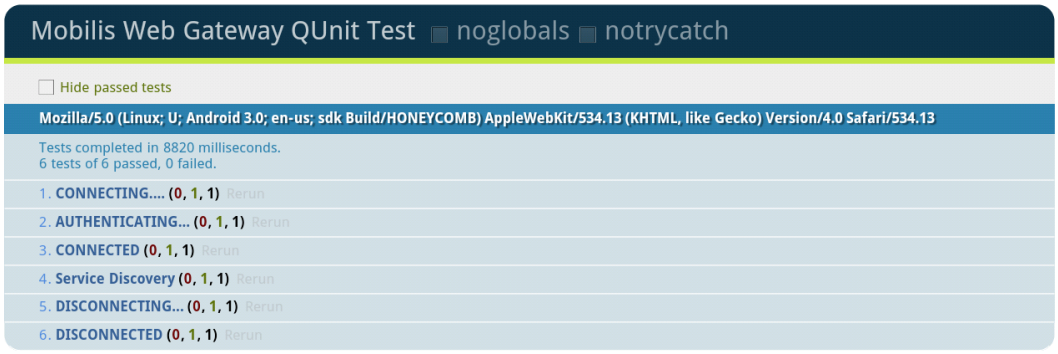

Figure A.12.: QUnit Test: Android Tablet Browser, Android 3

B. Performance Evaluation

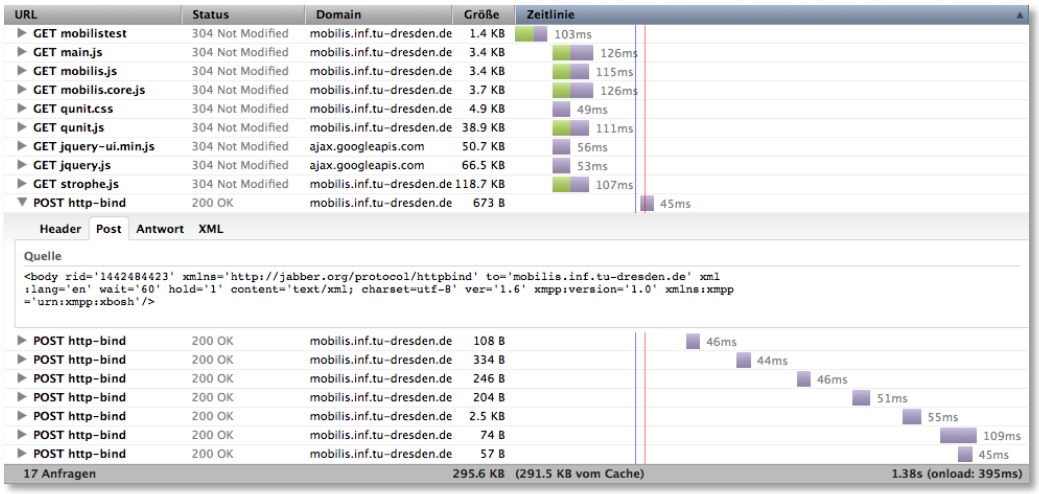

Figure B.1.: Initial HTTP Request

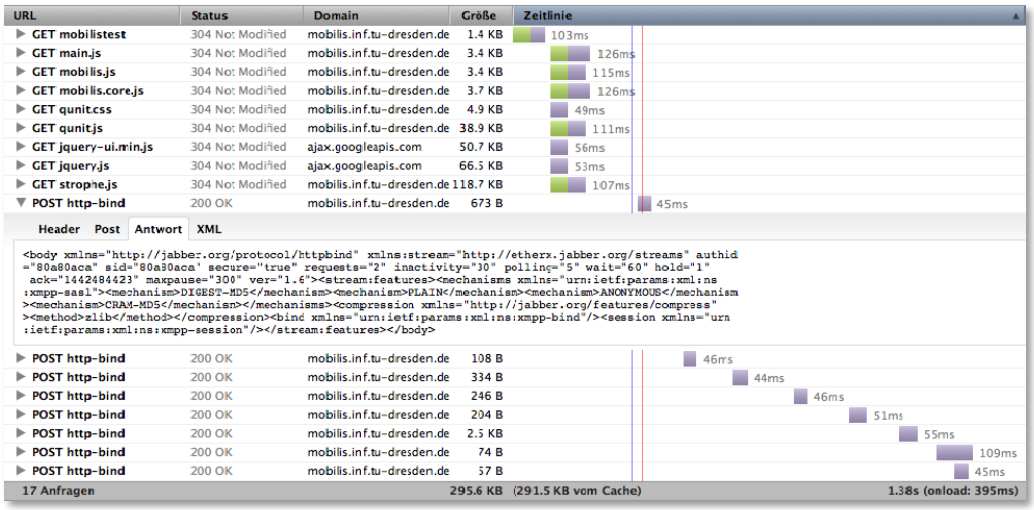

Figure B.2.: Initial HTTP Response

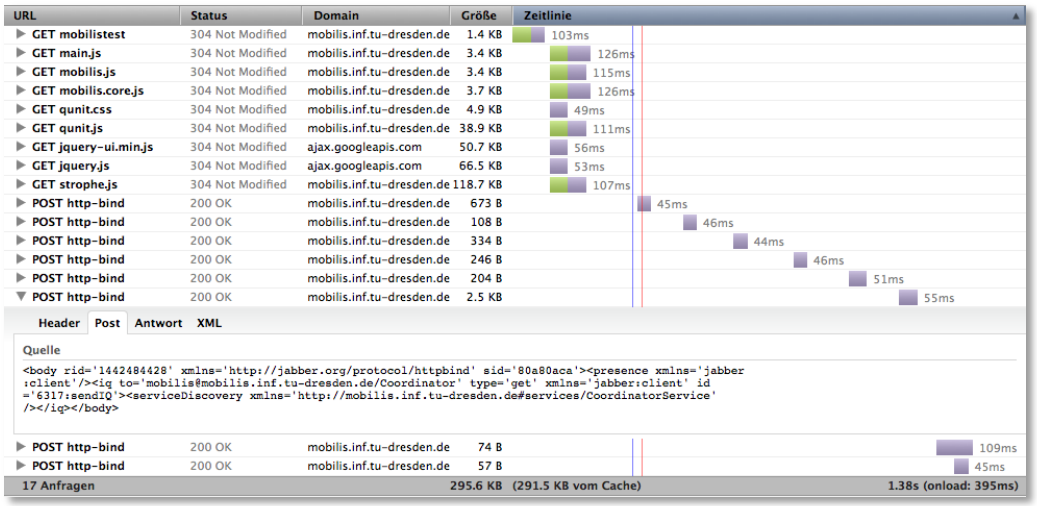

Figure B.3.: Service Discovery Request

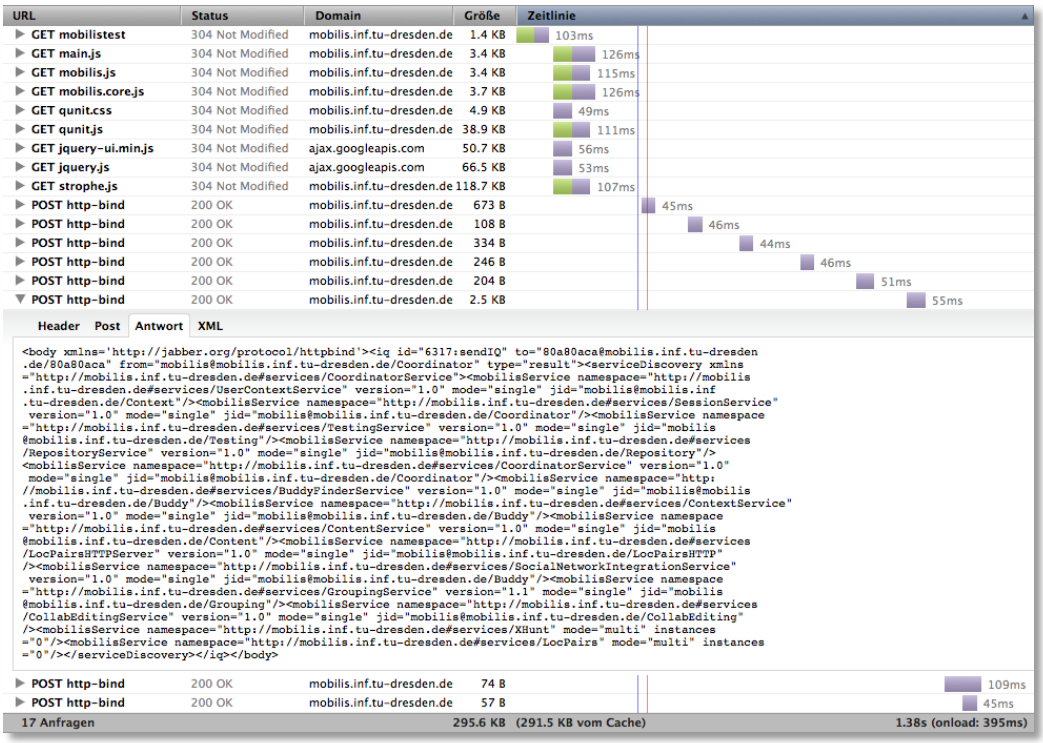

Figure B.4.: Service Discovery Response 55 ms