Integration of HTTP Push with a JSF AJAX framework

Master's thesis

Engin Bozdag

Integration of HTTP Push with a JSF AJAX framework

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Engin Bozdag
born in Malatya,Turkey

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Backbase BV
Stephensonstraat 19
1097 BA
Amsterdam,
the Netherlands
www.backbase.com

Integration of HTTP Push with a JSF AJAX framework

Author: Engin Bozdag
Student id: 1119869
Email: v.e.bozdag@student.tudelft.nl

Abstract

A new breed of web application, dubbed AJAX, is emerging in response to a limited degree of interactivity in large-grain stateless Web interactions. However, AJAX still suffers from the limitations of the Web's request/response (pull) architecture. This prevents servers from "pushing" real-time alerts such as market data, news headlines or auction updates. Several libraries, such as Cometd and DWR bring push support to existing AJAX applications. It is a known fact that these libraries will cause scalability problems in most web application servers. However, there has been no empirical study conducted to find out the actual trade-offs of applying push on browser-based or AJAX applications and comparing it with a pull approach. The integration of the mentioned libraries with existing AJAX frameworks is also far from trivial. This thesis first introduces a comparison of two free open-source push libraries. It later presents the results of an empirical study that compares push and pull approaches in terms of scalability, network usage and latency. Finally it shows how current push solutions can be integrated with an AJAX framework that is based on Java Server Faces (JSF) architecture by providing several extensions and sample applications.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	ir. Ali Mesbah, Faculty EEMCS, TU Delft
Company supervisor:	Dimitra Retsina, Backbase
Committee Member:	Dr. Eelco Visser, Faculty EEMCS, TU Delft

Preface

Several people have helped and supported me in writing this thesis.

First of all, I would like to thank my supervisor Ali Mesbah for his continuous help and assistance during the whole project. Without his feedbacks and suggestions, this thesis would not have the current structure and quality. Ali has been an excellent mentor.

Thanks to Arie van Deursen for his support when we needed resources during testing. He stepped in quickly, and thanks to him we obtained the numbers and results. He kept his support, gave valuable feedbacks and took over the key role to solve the problems within the project.

Thanks to Maikel Lobbezoo, a fellow student at Delft University. He helped us with the initial test setup, and shared his knowledge of performance testing. I think our paths during our research came across a lot and we both have benefited from each other.

Thanks to Kees Broenink of Backbase, for his help with JSF integration and other implementation issues. Our daily conversations however were not limited to JSF, and it included Thomas Mann, Orhan Pamuk, Patrick Suskind, South-West Germany wine, Turkish lokum, Dutch drops, etc., etc. I also would like to thank Kees for all the "homemade" eggs.

Thanks to Vedran Vego of Backbase, for his help with JSF demo applications. He also has been a good lunch buddy and cofounder of "East European Coffee Tasting Club".

Thanks to Dimitra Retsina of Backbase for her feedback on the draft version of this thesis.

Thanks to my girlfriend Josine for her patience and support. She kept me motivated during the whole research and never gave up.

Thanks to my parents, who always supported me, even though they were thousands of kilometers away.

Thanks to my friends and family in Holland: Koray, Deniz, Erdem, Moran, Cafer, Huseyin, Nihal, Mahmut and others that I forgot to mention. They brought color to black & white days.

Engin Bozdag
Delft, the Netherlands
December 4, 2007

Contents

List of Figures

Chapter 1

Introduction

Recently, there has been a shift in the direction of web development. A new breed of web application, dubbed AJAX (Asynchronous JavaScript and XML) is emerging in response to the limited degree of interactivity in large-grain stateless Web interactions. The intent is to make web pages feel more responsive by exchanging small amounts of data with the server behind the scenes and making changes to individual user interface components. This way, the entire web page does not have to be reloaded each time the user makes a change. This *Single Page Interface* (SPI) approach makes AJAX a serious option not only for newly developed applications, but also for existing web sites if their user friendliness is inadequate [31].

The term AJAX spread rapidly from a Weblog to the Wall Street Journal within weeks. The new web applications under the AJAX banner have redefined end users' expectations of what is possible within a Web browser. However, AJAX still suffers from the limitations of the Web's request/response architecture. The classical model of the web called REST [18] requires all communication between the browser and the server to be initiated by the client, i.e., the end user clicks on a button or link and thereby requests a new page from the server. This "pull" scheme helps scalability, but precludes servers from sending asynchronous notifications. There are many use cases where it is important to update the client-side interface in response to server-side changes. For example:

- An auction web site, where the users need to be averted that another bidder has made a higher bid. Figure 1.1 shows a screenshot taken from EBay. In a site such as EBay, the user has to continuously press the 'refresh' button of her browser, to see if somebody made a higher bid.
- A stock ticker, where stock prices are frequently updated. Figure 1.2 shows a screenshot taken from Yahoo! Stock-Ticker widget.
- A chat application, where new sent messages are delivered to all the subscribers. Figure 1.3 shows a sample chat application without push support.
- A news portal, where news are pushed to the subscriber's browser when they are published.

Today, these types of applications are usually implemented using a client-pull style, and the client component actively requests the state changes using client-side

timeouts. These solutions have many drawbacks; i.e., if the clients pull too often, this might lead to high server load. If the updates are not frequent, clients will make unnecessary requests. On the other hand, if the clients pull infrequently, they might miss some updates.

An alternative to this is the push-based style, where the server broadcasts the state changes to the clients asynchronously every time its state changes. Most AJAX implementations use the pull-based style. AJAX applications are designed to have high user interactivity and a low user-perceived latency. Using a push-based style can further improve these properties. There are several solutions used in the practice that still allow the client to receive (near) real-time updates from the server [10]. The HTTP based push solution COMET is a technique that allows real-time updates to be sent by the server. COMET is quite portable since it does not require any plug-in download and has no problems with firewalls, unlike a TCP socket based solution, such as Flash Sockets[1] or Java Applets[2]. The COMET approach is quite popular and its protocol BAYEUX is already implemented in several application servers, such as Jetty [12], Grizzly [4] and IBM Websphere [23]. The protocol defines several concepts (such as "channel") and steps (connect, subscribe, publish) that allows a generic implementation that can be used to create specific applications.

Visibility in software engineering is determined by the degree in which an external mediator is able to understand the interactions between two components [31]. BAYEUX protocol [35] is quite simple and if more developers implement it, this will have a positive effect on visibility. However, the current version of BAYEUX protocol has limitations, such as a lack of server-components support. The trade-offs of "push" using BAYEUX and its advantages & disadvantages against the pull approach is another issue that requires attention.

Another solution that brings push support is Direct Web Remoting (DWR). DWR is a Java open source library which allows code in a browser to use Java functions running on a web server just as if it was in the browser. DWR generates a JavaScript client file for every Java class on the server, which allows synchronization between the client and server. DWR defines its own custom messaging protocol in order to achieve this synchronization. This library is a possible candidate for frameworks that would like to add push support into their product. However, DWR also has limitations that require further study.

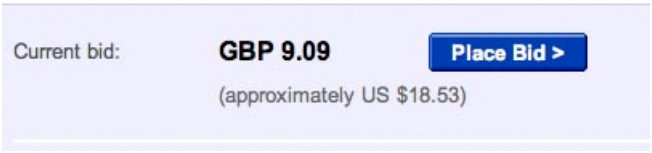


Figure 1.1: A screenshot taken from EBay.

This thesis first analyzes the limitations of the current BAYEUX protocol and DWR, and introduces several extra features which will extend them. It presents design and

[1] www.adobe.com/products/flash/

[2] http://java.sun.com/applets/

Figure 1.2: Stock ticker widget of Yahoo!.

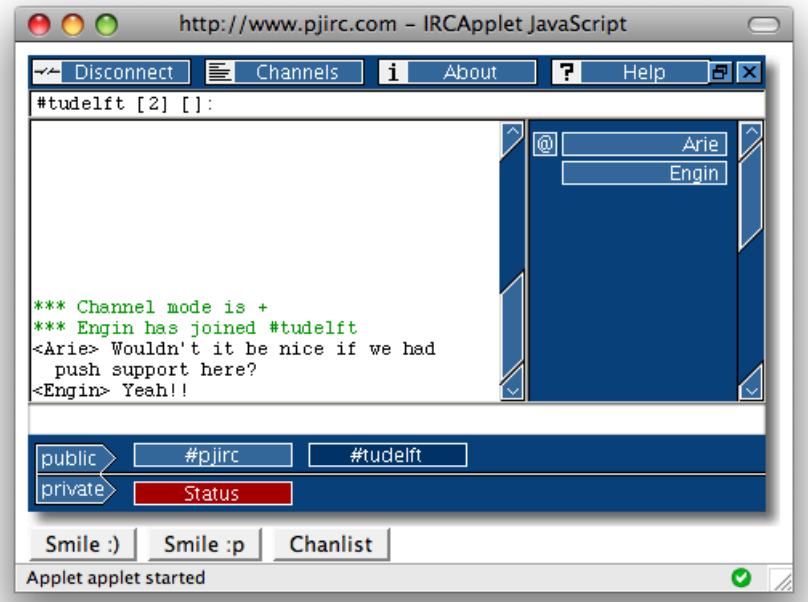


Figure 1.3: A typical chatroom application using Java Applets

implementation of a push component by providing sample applications. It also shows the results of an empirical study that compares push and pull.

The thesis is organized as follows. Chapter 2 shows the focus and the goal of this study. Chapter 3 gives a short introduction of AJAX and Java Server Faces (JSF) technology. Chapter 4 introduces our example JSF framework and lists the functional and non functional requirements for its push implementation. Chapter 5 lists some free open source push solutions and compares them using the listed requirements. Chapter 6 shows the results of an empirical study comparing push and pull using a Bayeux implementation. Chapter 7 discusses the design of Bayeux/Jetty and our extensions. Chapter 8 introduces the Bayeux implementation of Jetty, and discusses the extensions we have provided. Chapter 9 introduces sample applications that use both the BAYEUX extension and DWR. Chapter 10 discusses the current implementation and mentions several load balancing options. Finally, Chapter 11 ends the thesis with a conclusion and future work.

Chapter 2

Problem Statement

As a response to the lack of communication standards [31] for AJAX applications, the Cometd group[1] released a COMET protocol draft called BAYEUX [35]. BAYEUX defines how an HTTP push server and its clients should communicate with each other. BAYEUX is an important step on the way of standardizing the push approach. Another open source solution called DWR also brings push support, by synchronizing data between the client and the server. However, these solutions come with their limitations, i.e. they lack full server components support. The updates are directly sent to the push client. Most AJAX frameworks on the other hand, have server components. In these frameworks, the state of the AJAX client and the server are synchronized by using "pull" requests. This brings us our first research question:

How can a push library and an AJAX *framework be integrated, so that the application programmer has the minimum difficulty and the user experiences a minimum latency?*

Another important problem is the performance issues that a push solution will cause. It is generally accepted that a push solution that keeps an open connection for all clients will cause scalability problems. However, as far as we know, there has been no empirical study conducted to find out the actual trade-offs of applying push on browser-based or AJAX applications and comparing it with a pull approach. Such a study will answer questions about scalability, network usage and latency. It will also allow engineers to make rational decisions concerning key parameters such as pull and push intervals, in relation to, e.g., the anticipated number of clients. It will show the weaknesses of both approaches and might collect necessary data in order to implement a hybrid approach that combines both of the techniques. This problem brings up our second research question:

How does a push approach compare to a pull one, in terms of scalability, network usage and latency? How can we setup a controllable, distributed test environment, so that we can obtain empirical data, find answers to these questions and repeat the tests with other applications/servers, if necessary?

Backbase is an Amsterdam-based company that provided one of the first commercial AJAX frameworks [5]. The framework is still in continuous development, and in use by numerous customers world wide. The addition of a push implementation will improve the current Backbase framework. There are already AJAX solutions available

[1] `http://www.cometd.com`

that use the push technology and the demand for such a solution is high. Backbase would like to keep up with the changes that are already happening in the market and the technology.

Server components support is an important requirement in Backbase's push implementation, since the framework works with JSF components. Integrating Backbase's JSF edition with a push solution is not trivial, since these solutions are not designed with JSF in mind. Such an integration requires an investigation of Backbase JSF and push solutions such as Bayeux and DWR.

An empirical study comparing push and pull is also relevant for Backbase. Such a study will allow Backbase to advise a particular solution to its customers, based on the requirements of the application. For example, if the application has to serve a certain amount of users, and the empirical data shows that a single server cannot do this, then Backbase can provide this info to the client and suggest a load balancing solution.

In this thesis we discuss the limitations of BAYEUX and DWR, and suggest several improvements. We compare the performance of COMET with pure pull approach. We present the design and implementation of a push prototype for Backbase, based on this improved BAYEUX protocol. We also present several demos, showing how a JSF framework and push solutions can be integrated.

Chapter 3

Background Information

In this chapter we provide a summary of AJAX and JSF architecture.

3.1 AJAX

AJAX is a combination of several web application development technologies, previously known as Dynamic HTML (DHTML) and remote scripting, to provide a more interactive web-based user interface [31]. It is composed of the following technologies [20] :

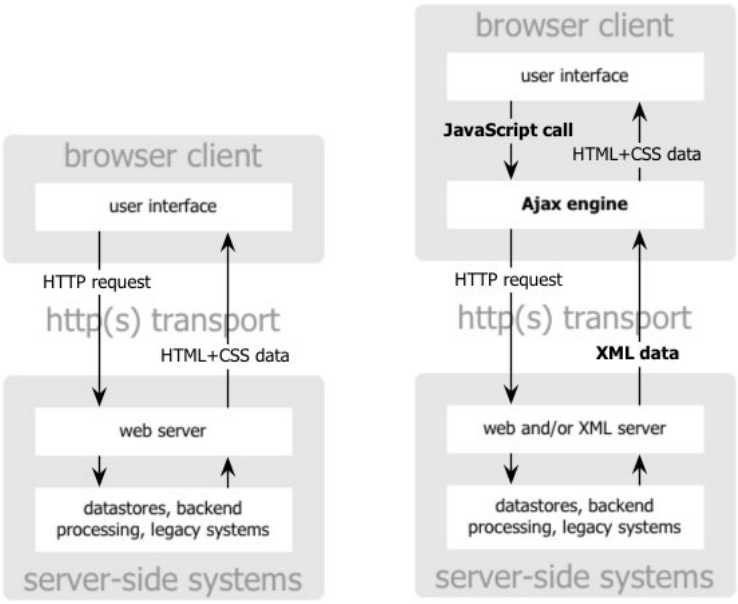
- Standards-based presentation using XHTML and CSS;
- Dynamic display and interaction using the Document Object Model (DOM);
- Data interchange and manipulation using XML and XSLT;
- Asynchronous data retrieval using XMLHttpRequest;
- JavaScript to bind everything together.

In the following subsection we compare AJAX approach with the classical web application model. Later we mention SPIAR [30], an architectural style for AJAX applications.

3.1.1 Classical web vs. AJAX

In the classical web application model (Figure 3.1a), the user performs an action that causes the browser to make an HTTP request to the web server. After receiving the request, the server does some processing (i.e., retrieving data, doing some calculations, connecting to various legacy systems) and then returns an HTML page to the client. However between making the request and receiving the response (Figure 3.2a), the user has to wait and a new page is needed to be reloaded in every request. This *Multi Page Interface* (MPI) approach has generally exhibited problems such as slow performance and limited interactivity, particularly when compared to typical desktop applications [34].

AJAX [20] tries to solve this problem by introducing an intermediary AJAX engine between the user and the server (Figure 3.1b). The engine is loaded at the start of the



(a) Classical Web Application Model

(b) AJAX Web Application Model

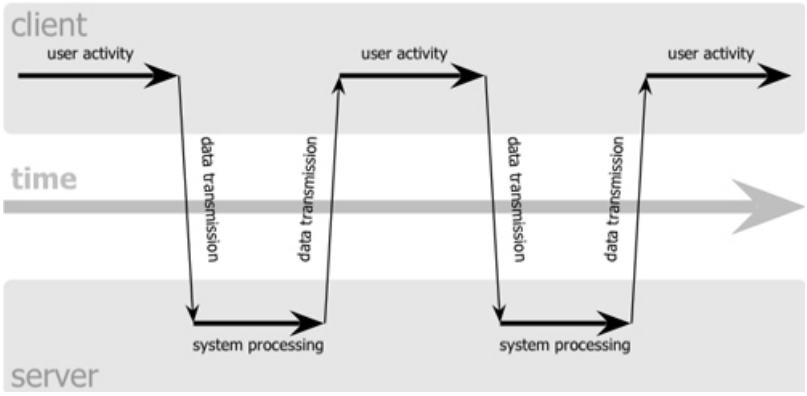
Figure 3.1: The traditional model for web applications compared to the AJAX approach. Images are taken from [20].

session and it handles the events initiated by the user, communicates with the server, and has the ability to perform client-side processing [29]. The user actions now do not generate an HTTP request to the server, but takes the form of a JavaScript to the AJAX engine instead (Figure 3.2b). If the user action does not need data from the server (i.e., simple data validation, editing data in memory, and even some navigation), the engine handles it on its own. If the engine needs something from the server in order to respond, it makes those requests asynchronously, without stalling the user's interaction with the application. The user stays in the same page, and the page does not need to be reloaded for every request.

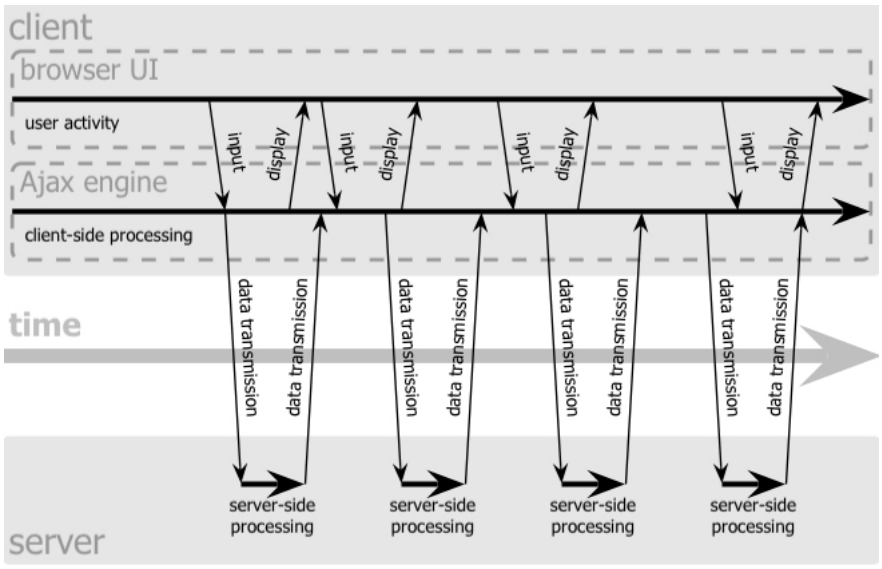
This *Single Page Interface* approach has been used in the software industry, even before the term AJAX was coined. It is widely used now by web applications such as Google Map, Gtalk, Flickr, and the new versions of Yahoo Mail and Hotmail.

3.1.2 An Architectural Style for AJAX

In the interest of scalability, the world-wide web adopts a *stateless* approach to client-server communication. This style of application is called REpresentational State Transfer (REST, [18]). In this style, clients request resources from servers (or proxy servers) using the resource's name and location, specified as a Uniform Resource Locator (URL,[7]). All interactions for obtaining a resource's representation are performed by synchronous request-response messages over an open, standard network protocol (HTTP, [17]). In this scheme, each interaction between the client and the server is independent of the other interactions. No "permanent" connection is established be-



(a) Classical Web Application Model



(b) AJAX Web Application Model

Figure 3.2: The sequence of events in different web application models. Images are taken from [20].

tween the client and the server and the server maintains no state information about the clients.

AJAX architectures on the other hand are not so easily captured in REST, due to the following reasons [29]:

- REST is suited for the transfer of large data, but AJAX works with small data exchanges.
- In REST, a client requests a specific resource. In AJAX, a response is required to a specific action.
- In REST, all interactions in order to obtain a resource are performed in a synchronous request-response scheme. With this scheme the user has to wait until a response is returned from the server. AJAX applications, however, require a model for asynchronous communication.

- No permanent connections are established in REST, so the server has to be stateless. This increases scalability, but the trade-offs with respect to network performance and user interactivity are of greater importance in AJAX.

In order to deal with these limitations of REST, SPIAR [30, 29] identifies the following architectural elements and shows their interaction in a processing view, where data flow and communication can be observed (See Figure 3.3).

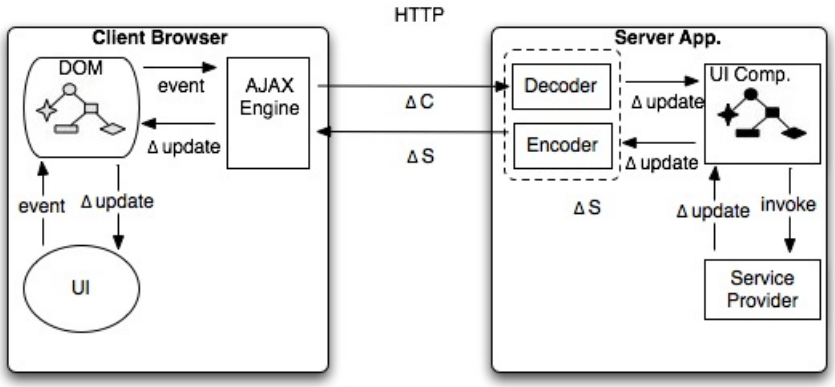


Figure 3.3: Processing View of a SPIAR-based architecture. Taken from [29]

- Processing Elements: These are the components that supply the transformation on the data elements. They are formed by the Ajax Engine and User Interface (UI) widgets on the client side, Delta Encoder/Decoder, UI Components and Service Provider on the server side.
- Data Elements: They contain the information that is used and transformed by the processing elements. They are formed by DOM on the client side and Delta communication messages, which are exchanges between the client and the server.
- Connecting Elements: They hold the components together and enabling them to communicate. Events, which are initiated by a user actions, Delta connectors which are light-weight communication media connecting the engine and the server, and Delta updates, which reflect state changes all belong to connecting elements.

According to Figure 3.3, user activity on the user interface fires off an event which is delegated to the AJAX engine. If a listener on the server side has registered itself with the event, the engine sends a DELTA-CLIENT messages of the current state changes and send it to the server, if needed, asynchronously. On the server, the message is decoded, and relevant components are identified in the component tree. The changed components invoke the event listeners of the service provider. The service provider will handle the action, update the corresponding components with the new

state, which will be rendered by the encoder. a DELTA-SERVER message is sent back to AJAX engine on the client side, and the engine will update the DOM and the interface. If no request to the server is needed, the engine may also update the DOM directly.

The asynchronous interaction and client side processing of AJAX improve user interactivity and user-perceived latency. Delta-communication style only interchanges state changes, instead of full-page retrievals, reducing data redundancy. UI components improve simplicity because developers can use existing components.

3.2 Java Server Faces (JSF)

JavaServer Faces (JSF) technology [1] is a framework for building user interfaces for web applications. JavaServer Faces technology includes:

- A set of APIs for: representing UI components and managing their state, handling events and input validation, defining page navigation, and supporting internationalization and accessibility.
- A JavaServer Pages (JSP) custom tag library for expressing a JavaServer Faces interface within a JSP page.

JSF technology is a user-interface framework for Java web applications. It is focused on the view tier of a Model View Controller (MVC) based architecture. JSF assembles reusable UI components in a page, connects these components to an application data source, and wires client-generated events to server-side event handlers.

[1] `http://wiki.java.net/bin/view/Projects/Javaserverfaces`

Chapter 4

Backbase JSF Edition and Push Requirements

The push implementation will be created for Backbase JSF Edition. In this chapter we will briefly describe Backbase JSF Edition, which is discussed in detail in [30, 1]. Later, we will list the requirements set by Backbase, for the push implementation.

4.1 Backbase JSF Edition 4.0.1

The Backbase JSF Edition [1, 30] implements and extends Java Server Faces (JSF) technology (See Section 3.2) , allowing programmers to create Rich Internet Applications (RIA) on the Java Platform, Enterprise Edition (JEE) platform. Backbase uses AJAX technologies to implement a RIA. In the Backbase JSF Edition, the rendering of the data (the presentation) is generated server-side and sent to the client, where the response is rendered by the client run-time in the client browser. The data to be rendered results in a partial update of the application, and not a full reload.

4.1.1 Server

Backbase JSF Server utilizes all standard JSF mechanisms such as validation, conversion and event processing through the JSF life-cycle phases. Any Java class that offers getters and setters for its properties can be directly assigned to a UI component property. Classes bound to UI component properties and events are called Backing Beans. Backing Beans are defined in a faces-config file, just like in a standard JSF application. Within a Backing Bean the developer has full control over all available UI components. JSF server tracks changes in a value, method, or a component, and later sends these changes as SERVER-DELTA (ΔS) back to the Backbase Client Runtime.

Just like any JEE server, JSF Server recognizes a user browser by the supplied SESSION-ID, which is sent by the browser in every request. With this id a corresponding session object is obtained, which allows the JSF Server to get access to the client's UI Components and manage state alterations.

4.1.2 Client

Backbase Client is an engine called Backbase Client Run-time (BCR), written in JavaScript. BCR acts as an "application server" running in the browser. On top of the BCR, the following components are located:

- Widgets: Widgets are user interface elements that the end-user interacts with. They are located on the client (browser). Backbase Tag Library (BTL) contains a UI library of widgets compliant with W3C standards, such as CSS and SMIL. BTL Widgets share common usability patterns and functions such as resizing, and keyboard navigation support. Widgets are represented as tags that can be mixed with existing HTML code. Examples include listGrid, tabBox, etc.

- Commands: Commands wrap complex functionality in a single call. Commands are available both as declarative tags as well as JavaScript. Examples include load, sort, show, etc.

- Behaviors: Behaviors are added functionalities that are applied to tags in any tag library. They are applied to a tag by using attributes. Examples include resize and drag/drop.

- Modules: Modules provide advanced application functionality by implementing W3C tag libraries and encapsulating shared tasks across multiple tag libraries. They include animation (SMIL), File Inclusion and Validation.

BCR's main functionalities are listed below:

- Creates a single page interface and manages the widget tree (view tree).

- Interprets JavaScript but also Backbase's XML based languages (XEL and TDL).

- Takes care of synchronization and state management by using delta-communication with the server, and asynchronous interaction with the user through the manipulation of the representational model. This is vital for an SPI application.

4.1.3 Client Server Synchronization

Backbase has defined a client/server protocol that allows for synchronization of components (e.g., input field values, attributes, style and structural changes) through decoding and encoding elements. Every component has the ability to tell the BCR which changes should be tracked and reported to the server. On the server, JSF Components are a set of well-defined UI components. Each component wraps a client-side widget and is capable of rendering the corresponding BTL code; i.e., there is a one-to-one relation between the component on the server and the one on the client for components that need state synchronization. The client can also consist of visual/non visual widgets that do not take part in the state management of the application. For such widgets there is no need for server-side components. Backbase delta communicating data are the synchronizing medium between the client and server. Backbase uses HTTP POST properties for the ΔC:

```
clientDelta: [evt=COMPONENT-ID|EVENT-NAME|EVENT-TYPE]?
|[att=COMPONENT-ID|ATTRIBUTE-NAME|ATTRIBUTE-VALUE]+
```

"clientDelta" is the name of the HTTP parameter with its value being a list of items. There are two types of items: the event item captures the data about the event triggered on the component, where as, an attribute item encapsulates information about the state changes of the client component itself. Each attribute item has three parts, namely, the unique identifier of the component (COMPONENT-ID), the name (ATTRIBUTE-NAME) and the new value (ATTRIBUTE-VALUE) of the affected attribute (e.g., [att=city|value|amsterdam]). Similarly, the event item contains the identifier of the component, the event name and the type of the event.

The ΔS consists of BTL + XEL and some commands. Backbase uses DOM events to delegate user actions to BCR which handles the events asynchronously. The events can initiate a client-side (local) change in the representational model but at the same time these events can serve as triggers for server-side event listeners.

The encoded CLIENT-DELTA (ΔC) will be posted to the server on certain defined events. These can be action events like clicking a button, or value change events such as checking a radio button the server-side decoder translates the ΔC and identifies the corresponding component(s) in the JSF component tree. The server-side encoder later renders a ΔS of the changes to be responded to BCR.

As we mentioned in Section 3.1, Mesbah and van Deursen introduced a style called SPIAR [31] in order to represent AJAX frameworks. According to Figure 3.3, AJAX Engine corresponds to BCR. *UI* represents client side widgets. *Decoder/Encoder* with *UI Components* represent Backbase JSF Server.

4.1.4 Conclusion

Backbase's server part is based on JSF architecture, which creates a binding between the client and server components. Whenever the client sends a Δ C, the server responds with a ΔS. Currently all open source standalone push libraries send raw data directly to the client. The challenge is to integrate those solutions with this synchronization architecture.

4.2 Push Requirements

In this section we will be discussing the requirements set up by Backbase, for the push implementation. We will be referring to the push server as "the server".

4.2.1 J2EE support

The server must be written in Java and should conform to the Java EE standard.

4.2.2 Open Source

The server must extend an already available free Open Source solution. The solution should be actively developed and extensible. The solution should also have a large user-base for support. Even though an evaluation of current open source push solutions was conducted in [10], a new solution known as DWR became available. Back-

base requires a further research of this solution and a comparison with Jetty's Bayeux implementation.

4.2.3 Supported Application Servers

The server should work with Apache Tomcat, IBM Websphere, BEA Weblogic, Oracle OC4J and JBoss.

4.2.4 Server Components Support

The server should work WITH and WITHOUT server components. The server should support Backbase Client Edition, but also Backbase JSF and Struts Editions.

4.2.5 Publisher

A publisher is the origin of the data, i.e. a chat user, a database, etc. Different publishers create different use cases. The server should allow two types of publishers:

Subscribed Publisher

A subscribed publisher is interested in broadcasted events of other publisher. Thus, a subscribed publisher is an event producer, but at the same time also a consumer. A typical use case is a chat application. In this scenario all subscribers are also publishers. This scheme usually consists of many channels, and everyone can send a message to others.

Nonsubscribed Publisher

A nonsubscribed publisher is not interested in broadcasted events of others. This type of publisher is not subscribed to a channel, but is allowed to publish messages . A typical use case is a stock ticker application. In this scenario, the subscribers are NOT publishers. Stock data generator is the only publisher, and does not need to be subscribed to the channel.

4.2.6 Clustering and load balancing support

The server should be extensible for use with standard Java EE techniques, such as load balancing and clustering.

4.2.7 Security

The server should be extensible to support authentication and encryption (SSL)

4.2.8 JRE Version

There are existing Backbase clients which use JVM 1.4.2, so the server should support JVM 1.4 and onwards.

4.2.9 Message Format

Backbase requires push communication messages to be wrapped in XML.

Chapter 5

Open Source Push solutions

HTTP Streaming is a basic and old method that was introduced on the web first in 1992 by Netscape, under the name 'dynamic document' [32]. The application of the HTTP Streaming scheme under AJAX is now known as COMET [39] or Reverse AJAX [14]. As mentioned in Section 4.2, the server must extend an already available free Open Source (OS) COMET solution. According to our research [10], the major free OS solutions consist of Cometd[1], Pushlets[2], Grizzly[3] and DWR[4]. Pushlets require a thread per client, which will become a problem with 100's of clients. The library is also not very popular and did not receive an update over a year. Grizzly's COMET support is still experimental and in transition of migration into Cometd's Bayeux. This leaves us with only two options: Cometd and DWR. In the next two sections we introduce both solutions and list their limitations. Later, we compare them, on basis of the listed requirements.

5.1 Jetty Cometd

COMET relies on XMLHttpRequest object [42] in order to perform streaming. The client opens a connection to the server by XMLHttpRequest, and the server holds on to this request to send an update later on. This long-lived XMLHttpRequest connection enables the server to send a message to the client when the event occurs, without the client having to explicitly request.

As a response to the lack of communication standards [31] for AJAX applications, the Cometd group released a COMET protocol draft called BAYEUX [35]. In the following subsection we present the protocol specification.

5.1.1 Protocol Specification

The BAYEUX message format is defined in JSON (JavaScript Object Notation)[5] which is a data-interchange format based on a subset of the JavaScript Programming Language. BAYEUX defines several connecting elements such as "channels" and list of

[1] http://www.cometd.com/

[2] http://www.pushlets.com

[3] https://grizzly.dev.java.net/

[4] http://getahead.org/dwr

[5] http://www.json.org

stages in order to establish a connection. Figure 5.1 shows these stages. In the figure one sided arrow from State A to B symbolizes a transition from A to B. If the arrow is two sided, that means a transition from state B to A is also possible. The sequence of state transition is as follows:

- In the handshake state, the client performs a handshake with the server, receives a client id and list of supported connection types.
- In the connect state, the client sends a connection request with its id and its preferred connection type.
- Once connected, the client can subscribe/unsubscribe to/from a channel, publish message, or simply reconnect to the server
- In order to keep the connection open all the time, the client will continuously reconnect to the server.
- If the client switches to disconnect state, it will have to start the handshake process once again. From the "reconnect" stage, the client switches to other states temporarily. After that particular stage is complete (for example subscription), the client will immediately switch to "reconnect" stage. This is vital to keep the connection open at all times.

During the reconnect stage (See Figure 5.2), the client will send a "reconnect" request with its client id and the timestamp of the last message it has received. This allows the server to compare the timestamp of the client and timestamp of the other messages, and resend them to the client if necessary. This way the client does not miss an update. If an event occurs on the server side (database update or a message published by another user), it will simply be returned to the client. If no event occurs during a predefined interval (by default this is 45 seconds in Bayeux), a timeout will occur and the server will send just the timestamp, asking the client to update its own timestamp and to reconnect. Since the users reconnect every time they receive data, the system acts as a mix of push and pull. This scheme is known as *long polling* and results in less threads usage at a given time, which helps scalability. However, under heavy load a delay can occur, if the client can not reconnect. This is not acceptable in certain applications, where the time of the delivery is critical.

This protocol follows the 'topic-based' [16] publish-subscribe scheme. which groups events according to their topic (name) and map individual topics to distinct communication channels. Participants subscribe to individual topics, which are identified by keywords, Like many modern topic-based engines, BAYEUX offers a form of *hierarchical addressing*, which permits programmers to organize topics according to containment relationships. It also allows topic names to contain *wildcards*, which offers the possibility to subscribe and publish to several topics whose names match a given set of keywords.

The protocol has recently been implemented and included in a number of web servers including Jetty[6] and IBM Websphere[7]. While DOJO Toolkit[8] supplies the

[6] http://www.mortbay.org

[7] www.ibm.com/websphere

[8] http://dojotoolkit.org/

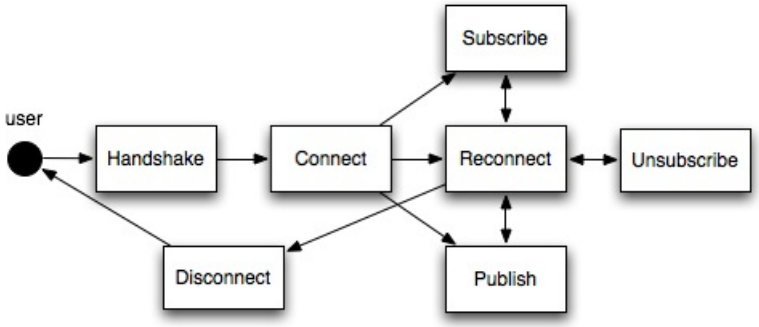


Figure 5.1: Stages in Bayeux

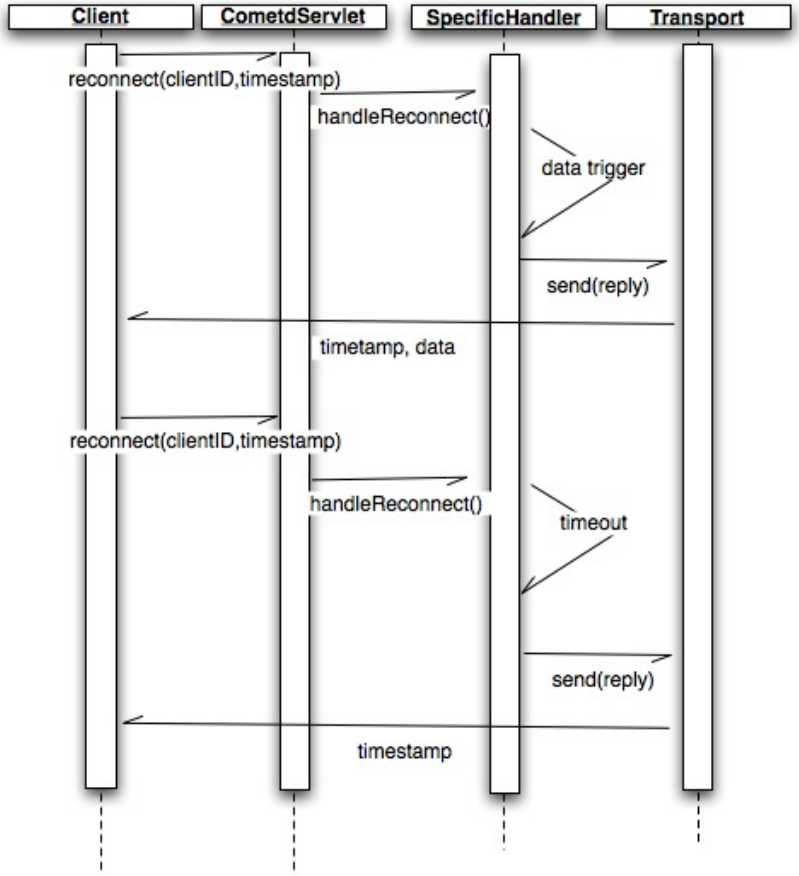


Figure 5.2: Bayeux reconnect

client library, Jetty's Bayeux module forms the Java part of the Cometd framework. Jetty defines different "handlers" to take care different Bayeux stages and supports Continuations [13], which allows less thread usage for a big number of clients. Although the protocol is open for the implementation of different connection types, Jetty currently only supports *long polling*. Cometd has other server side implementations such as Twisted. However, throughout this thesis we focus on Jetty's implementation, since Java EE support is required according to Backbase (See 4.2). We will use the words Jetty and Cometd interchangeably, since Jetty's COMET module forms the Java part of the Cometd library.

5.2 DWR

DWR[9] is a Java open source library which allows code in a browser to use Java functions running on a web server just as if it was in the browser. DWR works by dynamically generating JavaScript based on Java classes. The user feels like the execution is happening on the browser, but in reality the server is executing the code and DWR is marshalling the data back and forwards. DWR works like an RPC mechanism (such as Java RMI[37]), but without requiring any plugins.

DWR consists of two main parts:

- A Java Servlet running on the server that processes requests and sends responses back to the browser.
- JavaScript running in the browser that sends requests and can dynamically update the webpage by using the received response from the server.

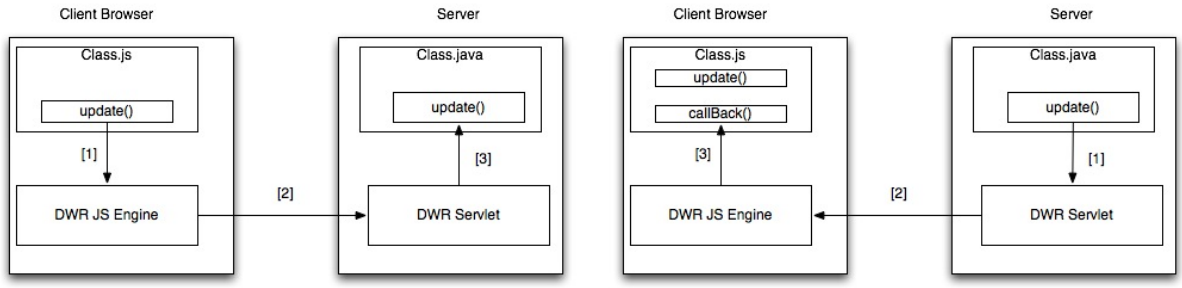
As we see from Figure 5.3a, on the server side we have DWR servlet and a Java Class (Class.java) with an example method "update()". On the client side, there is DWR JavaScript engine and Class.js, which is generated by DWR by using Class.java. In DWR communication, the following sequence of events takes place:

- When there is an update on the client, update() method of Class.js is called
- update() sends a request to call the "update()" method of Class.java.
- DWR JavaScript engine sends this request to DWR Servlet
- DWR Servlet calls update() method of Class.java

It is also possible to receive data back to the client (See Figure 5.4b). The servlet can call a callback function of the client by using DWR.

In order to use DWR, the user has to add the dwr.jar into its Web application's library directory and add the servlet to its configuration file (web.xml). The user also has to specify the Java classes that DWR will use by adding them into dwr.xml file. For every specified Java class, an appropriate JavaScript file will be generated. The user than has to include that file, and DWR's JavaScript engine in the application's web page (such as index.jsp)

[9] `http://getahead.org/dwr`



(a) Simple request (b) Callback mode

Figure 5.3: DWR architecture

With version 2.0 and above DWR supports COMET. DWR calls this type of communication "Active Reverse AJAX" [14] and allows the following modes:

- **Full streaming (Pure push):** As can be seen in Figure 5.4a, the server sends data in HTTP chunked mode [41], without closing the connection. In this mode, the same connection is used to send multiple responses. In order to deal with scalability, Jetty Continuation library [13] is used.

- **Early closing:** This is the same technique as "long polling" of Bayeux. Figure 5.4b shows that when there is data, or if a timeout has occurred, the server will return a response and the client has to make a new request.

- **Poll:** This mode is also known as push by continuous pulling [10]. The client checks with the server at regular user-definable intervals (See Figure 5.4c).

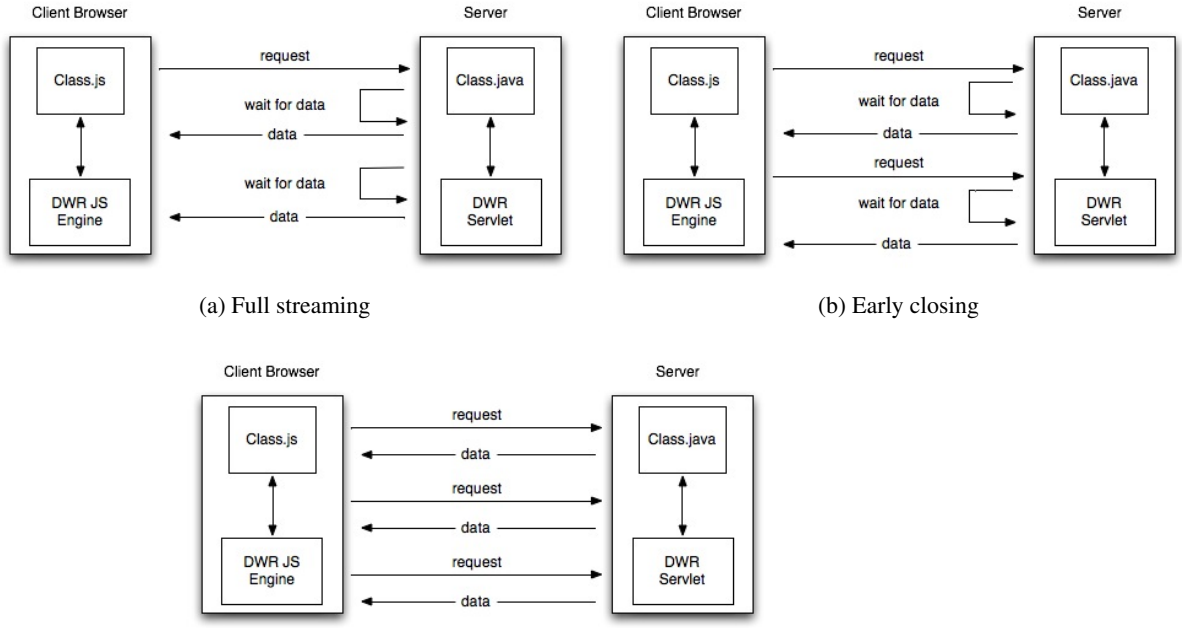
DWR also supports a "passive mode", in which the server waits the next time the browser makes a request, and then sends all the updates along with the request. This mode is also known as "PiggyBack mode".

Unlike Cometd, DWR does not define any connecting elements such as "channels", nor handlers for different stages. According to DWR's website [10], the library has been looked at by "some fairly serious performance gurus" who told that DWR was basically irrelevant to performance. There are no actual numbers available.

5.3 Comparison

Both Cometd and DWR are free Open Source libraries that can bring push capabilities into Backbase's JSF Framework. In this section we compare them, on basis of Backbase requirements (See 4.2). Note that we will skip the conditions Java EE support, Open Source and Supported Application Servers, since both solutions fulfill these requirements.

[10] `http://getahead.org/dwr/hints,12/11/2007`



(a) Full streaming

(b) Early closing

(c) Polling

Figure 5.4: DWR Active Reverse AJAX modes

5.3.1 Server Component Support

As mentioned in 4.1, Backbase JSF server contains UI components on the server side which correspond to client side widgets. JSF Server and the Client Runtime exchange deltas to synchronize the state of the components (See Figure 3.3). In this subsection we analyze the server component support of Cometd and Bayeux.

Jetty Cometd

In BAYEUX, every published data is sent to all the subscribers directly. Figure 5.5 shows the coexistence of Cometd with Backbase JSF architecture. Comparing this figure with the original architecture in Figure 3.3, we see two new components: Push Server and Push Client. In this scenario, the following events take place:

1. Service Provider publishes data to Push Server
2. Push Server broadcasts this data to all the subscribers by using the established Bayeux channel
3. Push Client on the AJAX Engine receives the data and triggers an event.
4. The event is then sent to the Server App. in a ΔC form.
5. Server side processes this ΔC just like it would do for any JSF request, and replies with a ΔS.

6. AJAX Engine processes the incoming ΔS and updates the widgets.

This scheme works, but is not efficient. For every incoming push data and widgets that have server side components, the client has to synchronize with the JSF server. However, in many cases the push server and the JSF server are in the same machine or network. The desirable solution is the push server to send the data to the application server directly, obtain ΔS and send the ΔS to the client. This way, the client is already synchronized with the JSF Server, and does not need to send a ΔC. Bayeux requires several extensions in order to allow this short-cut. See Chapter 7 for our extension.

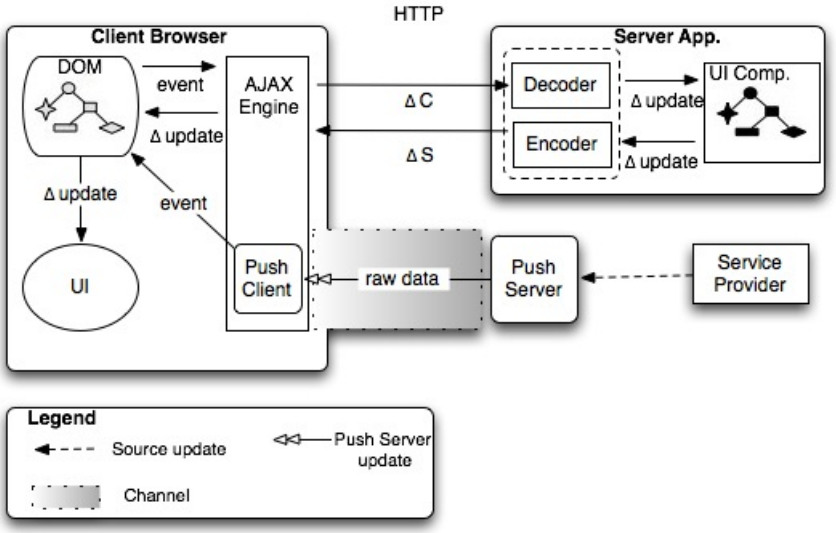


Figure 5.5: Cometd integration with SPIAR

DWR

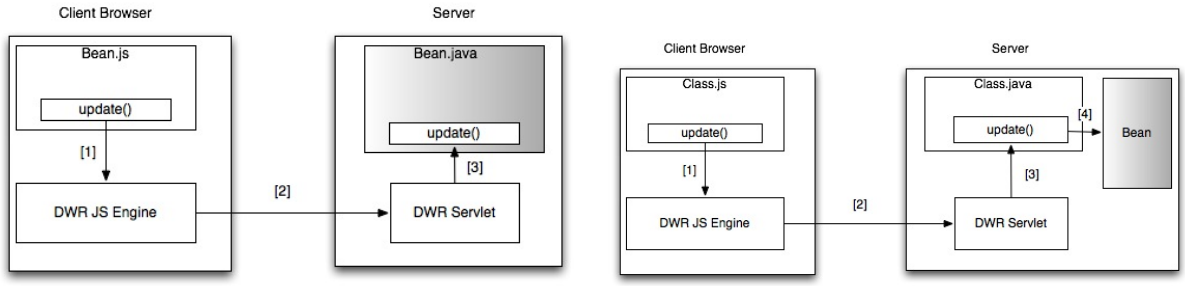
For server components support, DWR provides two experimental tools:

- JSF Creator: This tool allows DWR to use an existing backing bean and generate JavaScript versions of its messages (See Figure 5.6a). In this scenario, an extra method is added to the existing bean (In the figure, *update*), to handle JavaScript calls from the client.
- FacesFilter: This filter allows the Java class to reach a bean, without being part of JSF lifecycle (See Figure 5.6b). This way the Java class can call the bean's public methods.

JSF creator allows to add push functionality to an existing bean, however it also generates JavaScript functions for all methods in the bean, which are usually not used/needed. This might increase the page load time. FacesFilter does not have this downside; however it requires some extra action to reach the bean, which might be tedious for the application developers.

Both methods update the server side (the bean). In order to update the client, some JavaScript action is needed on the client DOM. This requires JavaScript knowledge, which Backbase JSF edition is trying to avoid. DWR also provides some server side utility tools that allow the servlet to directly update HTML elements on the client, without needing JavaScript knowledge. However, the tool cannot edit all HTML elements and its usage is currently very limited. For example, a JSF "dataTable" element cannot be edited with such function.

DWR does not fully integrate with the "Client Server Synchronization" of Backbase JSF Framework. For example, if a URL of a *graphicImage* element is changed, a different image will be displayed on the user's browser. DWR can change the URL on the bean, however in order to update the browser, it either has to make some JavaScript calls, or refresh the page. So, DWR does not integrate with JSF, but exists along JSF. DWR has to update the server side and client side both, in order not to break the JSF binding between the client and the server.



(a) by using JSF Creator (b) by using Faces Filter

Figure 5.6: DWR + JSF integration

5.3.2 Publisher

Backbase requires two different types of publishers:

- A user that is publishing into a channel, but also interested in other published messages. A typical use case is a chat room application.
- A user that is publishing, but not interested in an other published messages in the channel. A typical scenario is a stock ticker application.

Cometd allows both modes, a non subscribed user can also broadcast messages, as long as it is authenticated. In DWR, user subscription mechanism is left to the application programmer, so it will support both.

5.3.3 Clustering and Load Balancing Support

Cometd

Jetty has defined interfaces for JMS support, in order to allow servers communicate with each other. Jetty developers are also collaborating with the developers of the

Open Terracotta framework[11]. A fully working solution is to be expected.

DWR

Current version of DWR does not work in a clustered environment. There are efforts [15] to make it work with JGroups [12] and Terracotta. No release is made yet.

5.3.4 Security

Cometd

Cometd provides an authorization interface in which custom authorization algorithms can be created. It provides the encryption of messages sent. Cometd supports SSL, as any Java EE Application.

DWR

According to their website[13] DWR has the following security measures:

- DWR can be integrated with the Acegi security framework[14].
- DWR uses a secret in a POST body to allow the server to deny forged requests.
- Older versions of Safari (up to 1.2) have buggy XHR implementations that claim to support GET and POST, however the body of POST requests is lost. To work around this limitation DWR automatically detects buggy versions of Safari and switches from POST (the default) to GET.
- DWR allows granting access using Java EE based mechanisms.
- DWR will not let the user create or convert access to any of its internal classes. This is designed to make it difficult to for an attacker to manipulate the DWR core files to elevate permissions.

5.3.5 JRE Version

Cometd

Jetty's BAYEUX implementation uses JDK 1.5 specific elements that do not work on JVM 1.4. Problems include the use of generics and 1.5 specific elements. These JDK 1.5 specific elements need to be converted to JDK 1.4 supported components.

DWR

According to their website [15] DWR supports JDK 1.3 and onwards.

[11] http://www.terracotta.org/
[12] http://www.jgroups.org
[13] http://getahead.org/dwr/security
[14] http://www.acegisecurity.org/
[15] http://getahead.org/dwr/browser/environment

5.3.6 Message Format

Cometd

Cometd uses JSON protocol to encapsulate the push data. JSON has the following advantages:

- JSON is quite simple comparing to XML, and is easier to read in most debugging tools.
- JSON has support for Unicode.
- JSON format is self-documenting: it describes structure and field names as well as specific values.
- JSON represents records, lists and trees.

JSON also has several disadvantages, they are listed below:

- Given that XML has been around for years, there are number of XML data-binding APIs to create XML in several programming languages. On the other hand, APIs to create JSON responses are fairly new.
- Browsers do not have built-in JSON parsers.
- JSON does not have namespaces.
- JSON has No validator.
- JSON does not support different character encodings.
- JSON could also pose security problems [11]. A malicious user can include script along with data in JSON responses, and while processing the response, the user will also execute the script, which may pose security risks. Solutions such as JSON Commented are experimental.
- JSON is not extensible.

DWR

DWR uses its own protocol to communicate. For example a message sent by the DWR JavaScript engine looks like the following:

```
page=/StockTickerDWR/index.jsf
httpSessionId=917C51DCA697AECA8815B2C4BBA5B5D2
scriptSessionId=9D5E607CF765F526E575CCB6BD306047550
client.publish("/stocks/backbase","7");
```

Page represents the path to the application's web page. *httpSessionId* represents the cookie received from the server. *scriptSessionId* is a unique id for the given page, so that server side knows from which page the client came from.

Server side responds to this request with this data:

```
dwr.util.setValue("IBM_stock","21");
```

In this code, DWR Servlet tells the JavaScript Engine to change the value of the field with the id “Sunstock” to 21.

This example is trivial, however for some interactions the message can get complex, which makes it difficult to debug the current state and sent messages. DWR proprietary protocol has a negative effect on code visibility. An external mediator is not able to understand the interactions between two components easily.

5.4 Conclusion

Both Cometd and DWR are push solutions that have their advantages and disadvantages. DWR is more application oriented and the user has to define classes even for simple concepts such as “channel”. The user also has to implement certain functionalities such as subscription, unsubscription, disconnection, etc. in every application. These are already available in Cometd with certain handlers, which the user can use immediately. DWR supports differing push techniques, while Cometd is only supporting “long polling” at the moment.

Cometd comes with no JSF support. However, since it clearly separates the push process into separate stages, it is easy to implement such an extension. DWR on the other hand has experimental JSF support that makes it exist with JSF, without fully integrating with it. It uses a custom messaging protocol and is not easily extensible. DWR developer is planning to adapt the BAYEUX protocol in the future releases, which will increase its visibility.

Chapter 6

Comparing push vs pull

This chapter will appear in the proceedings of the 9th IEEE international symposium on Web Site Evolution (WSE'07) [9].

In order to synchronize the data on the browser and the data on the server, most AJAX applications check with the server at regular user-definable intervals known as *Time to Refresh* (TTR). This check occurs blindly regardless of whether the state of the applications has changed [10]. In order to achieve high data accuracy and data freshness, the pulling frequency has to be high. This, in turn, induces high network traffic and possibly unnecessary messages. The application also wastes some time querying for the completion of the event, thereby directly impacting the responsiveness to the user.

In order to solve these problems, COMET approach enables the server to send a message to the client when the event occurs, without the client having to explicitly request. COMET defines a protocol called Bayeux, which follows the 'topic-based' [16] publish-subscribe scheme. This scheme groups events according to their topic (name) and map individual topics to distinct communication channels. Participants subscribe to individual topics, which are identified by keywords.

Even though it is known that push approaches such as COMET causes scalability problems on most web servers, there has been no empirical study conducted to find actual results. There also has been no research to compare pull approach with push, for AJAX applications. In order to make rational decisions concerning key parameters such as pull and push interval, we performed an empirical study. In this chapter we will present the results.

6.1 Experimental Design

In this section we will present our experimental setup.

6.1.1 Goals and Setup

The goals of our experiment consist of exploring the actual performance trade-offs of a COMET push implementation and compare it to a pure pull approach on the web by conducting a controlled empirical study. The experiment has to be repeatable for

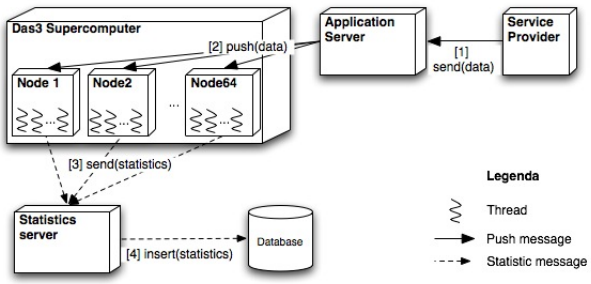


Figure 6.1: Experimental Environment

push and pull but also for different input variables such as number of users, number of published messages and intervals.

We aim at achieving these goals by:

- creating a push application consisting of the client and the server parts,
- creating the same application for pull,
- creating an application which publishes a variable number of data items at certain intervals,
- mimicking many concurrent web clients operating on each application,
- gathering data and measuring: the mean time it takes for clients to receive a new published message, the load on the server, number of messages sent or retrieved, the effects of changing the data publish rate and number of users,
- analyzing and explaining the measurements found.

To see how the application server reacts to different conditions, we use different combinations of three variables:

- Number of concurrent users (100, 200, 350, 500, and 1000). The variation helps to find a maximum number of users the server can handle simultaneously and 1000 seemed to be the upper-bound for our test. This is because the server was already running on 100% CPU with 1000 users. We also tried 2000 and 5000 users, however the server was so saturated that it was not able to send any updates anymore.
- Publish interval (5, 10, 15, and 50 seconds): The frequency of the publishing updates is also important. Because of the *long polling* implementation in BAYEUX (See Section 5.1), the system should act more like pure pull when the publish interval is small, and more like pure push when it is bigger. We chose the interval 50 seconds, because the client timeout of BAYEUX protocol is 45 seconds, and we expect this interval to cause many disconnects, hence affecting the performance.
- Push or Pull: We also made an option in our test script that allowed us to switch between pull and push. To make the total number of combinations smaller, we set the pull interval as 15 seconds.
- Total number of messages: For each combination, we generated a total of 10 publish messages.

6.1.2 Tools

In order to simulate a high number of clients, we evaluated several open source solutions. *Grinder*[1] seemed to be a good option, providing an internal TCPProxy, allowing to record events sent by the browser and later replay them. It also provided scripting support, which allowed us to create a script that simulates a browser connecting to the push server, subscribing to a particular stock channel and receiving push data continuously. In addition, Grinder has a built-in feature that allows us to create multiple threads of a simulating script.

Because of the distributed nature of the simulated clients on different nodes, we used Log4J's SocketServer[2] to set up a logging server that listens for incoming log messages. The clients then send the log messages using the `SocketAppender`.

We used *TCPDump*[3] to record the number of TCP (HTTP) packets sent to and from the server. We also created a script that uses the UNIX *top* utility[4] to record the CPU usage of the application server. This was necessary to observe the scalability and performance of each approach.

6.1.3 Sample Application

In order to respond to publish events and create client-side processing, we developed a Stock Ticker application.

The Push version consists of a JSP page which uses Dojo's Cometd library[5] to subscribe to a channel and receive the Stock data. We use Rico[6] to give color effects to different data values on the web interface. For the server side, we developed a Java Servlet (PushServlet) that pushes the data into the browsers using the Cometd library. The PushServlet manages the client connections, receives data from back-end, and publishes it to the clients.

The pull version has also one JSP page, but instead of Cometd, it uses the normal *bind* method of Dojo to request data from the server. The pull nature was set using the standard `setInterval` JavaScript method. On the server, a PullServlet was made which keeps an internal stock object (the most recent one) and simply handles every incoming request the usual way.

The Service Provider Java application was created which uses the HTTPClient library[7] to publish stock data to the Servlets. The number of publish messages as well as the interval at which the messages are published are configurable.

Simulating clients To simulate many concurrent clients we use the TCPProxy to record the actions of the JSP/Dojo client pages for push and pull and create scripts for each in Jython[8]. Jython is an implementation of the high-level, dynamic, object-oriented language Python, integrated with the Java platform. It allows the usage of

[1] `http://grinder.sourceforge.net`
[2] `http://logging.apache.org/log4j/docs/`
[3] `http://www.tcpdump.org/`
[4] `http://www.unixtop.org/`
[5] `http://dojotoolkit.org/`
[6] `http://www.openrico.org/`
[7] `http://jakarta.apache.org/commons/httpclient/`
[8] `http://www.jython.org`

Java objects in a Python script and is used by Grinder to simulate web users. In our tests, Jython scripts are actually imitating the JSP/Dojo client pages.

6.1.4 Testing Environment

We use the Distributed ASCI Supercomputer 3 (DAS3)[9] to run various numbers of web clients on different distributed nodes. The DAS3 cluster at the Delft University consists of 68 dual-CPU 2.4 GHz AMD Opteron DP 250 compute nodes, each having 4 GB of memory. The cluster is equipped with 1 and 10 Gigabit/s Ethernet, and runs Scientific Linux 4.

The application server runs on a Pentium IV, 3 Ghz (Hyperthreading) machine with 1 Gb memory, and Linux Fedora as its Operating System. We use Jetty 6.1.2 as our application server, because it is the only open-source Java EE application server that currently implements the COMET BAYEUX protocol. Jetty uses Java's new IO package (NIO). NIO package follows the event-driven design, which allows the processing of each task as a finite state machine (FSM). As the number of tasks reaches a certain limit, the excess tasks are absorbed in the server's event queue. The throughput remains constant and the latency shows a linear increase. The Event-driven design is supposed to perform significantly better than thread-concurrency model [43, 44].

The connectivity between the server and DAS3 nodes is through a 100 Mbps ethernet connection.

6.1.5 Sequence of events

A routine test run consists of the following steps (See Figure 6.1):

1. The Service Provider publishes the stock data to the application server via an HTTP POST request, in which the creation date, the stock item id, and the stock data are specified.
2. For push: The application server *pushes* the data to all the subscribers of that particular stock. For pull: the application server updates the internal stock object, so that when clients send pull requests, they get the latest data.
3. Each simulated client logs the responses (after some calculation) and sends it to the statistics server. Grinder also processes the data from each client and sends the statistics, such as response time, to the statistics server, which runs on a separate machine.

It is worth noting that we use a combination of the 64 DAS3 nodes and Grinder threads to simulate different numbers of users.

6.1.6 Data Analysis

We created a Data Analyzer that reads the data from Grinder and Logging Server logs and writes all the info into a database using Hibernate[10]. This way, different views of the data can be obtained easily using queries to the database.

[9] http://www.cs.vu.nl/das3/overview.shtml

[10] http://www.hibernate.org

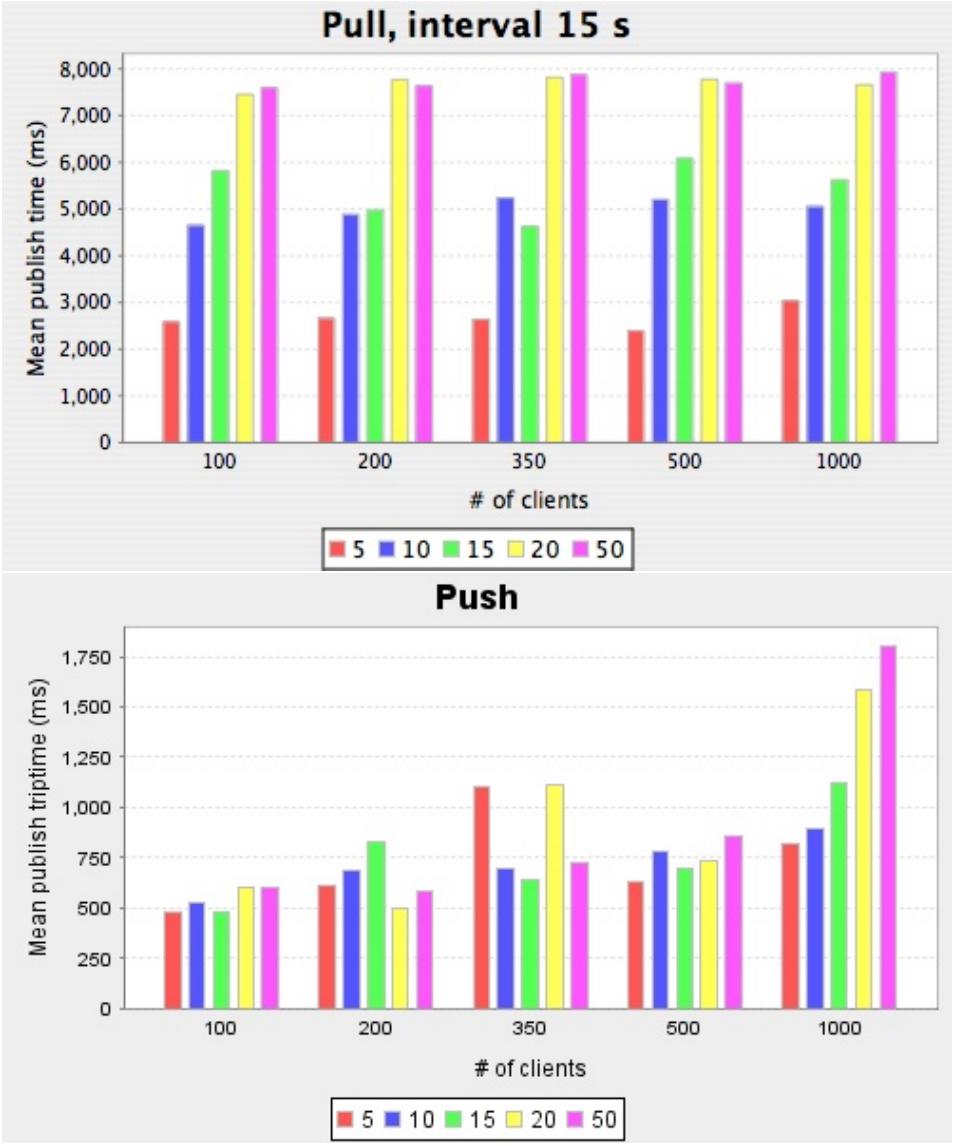


Figure 6.2: Mean publish triptime.

6.2 Results

In the following subsections, we present the results which we obtained using the combination of variables mentioned in 6.1.1. Figures 2–5 depict the results. Note that for each number of clients on the x-axis, the five publish intervals in seconds (5, 10, 15, 20, 50) are presented.

6.2.1 Publish triptime

We define triptime as follows:

Triptime = | *Data Creation Date* − *Data Receipt Date* |

Data Creation Date is the date on the Service Provider (Publisher) the moment

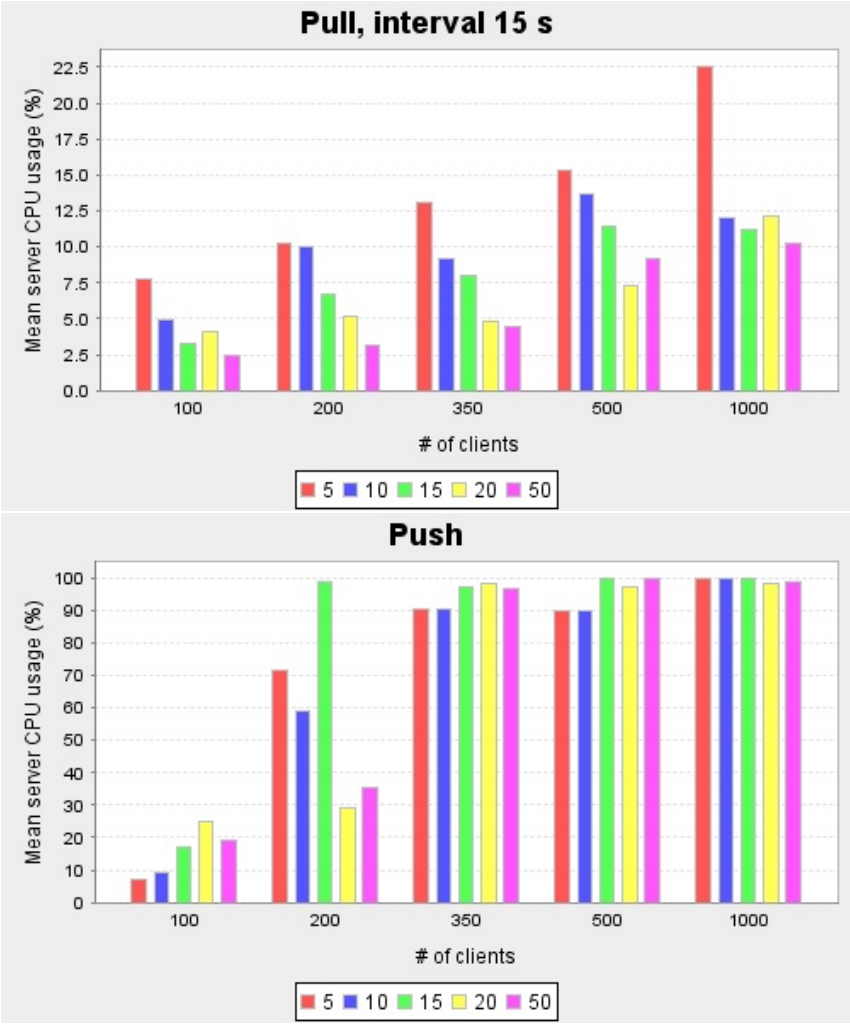


Figure 6.3: Server application CPU usage.

it creates a message, and *Data Receipt Date* is the date on the client the moment it receives the message. Triptime shows how long it takes for a publish message to reach the client and can be used to find out how fast the client gets notified with the latest events. Note that it is very important to synchronize the datetime for both the Service Provider and the clients.

Figure 6.2 shows the mean publish triptime versus the total number of clients, for both pull and push techniques.

6.2.2 Server Performance

Since push is stateful, we expect it to have some administration costs on the server side, using resources. In order to compare this with pull, we measured the CPU usage for both approaches. Figure 6.3 shows the mean server CPU usage as the number of clients grows, for push and pull.

6.2.3 Received Publish Messages

To see how pull compares to pure push in message overhead, we published a total of 10 messages and we counted the total number of (non unique) messages received on the client side. Figure 6.4 shows the mean number of received non-unique publish items versus the total number of clients, for both push and pull. Note that if a pull client makes a request while there is no new data, it will receive the same item multiple times. This way a client might receive more than 10 messages.

6.2.4 Received Unique Publish Messages

It is also interesting to see how many of the 10 messages we have published reach the clients. This way we can determine if the clients miss any publish items. Figure 6.5 shows the mean number of received unique publish items versus total number of clients.

6.3 Discussion

6.3.1 Data Coherence

We define a piece of data as coherent, if the data on the server and the client is synchronized. We check the data coherence of both approaches by measuring the triptime. As we can see in Figure 6.2, the triptime is, at most, 1,750 milliseconds with push. In Pull, this can go up to 25 seconds. This shows us that pull is not as responsive as push, and if we need high data coherence, we should always choose the push approach. In Figure 6.2 we also see that with 1000 users and a publish interval of 50 seconds, the triptime increases noticeably. With such a big interval, no response is being sent to the client, and the client is waiting for data, thus occupying a thread. This makes it hard for other clients to reconnect and get new data, which increases the triptime. With an interval of 5 seconds, the triptime is lower. This is because the clients are quickly receiving responses and disconnecting. This makes some threads available, which makes it possible for other clients to connect.

6.3.2 Server Performance

One of the main issues of all distributed systems and in particular that of web-based applications is scalability and performance. As it is depicted in Figure 6.3, the pull style has a much better performance compared to push and this is valid even for small number of users (e.g., 100). With push, when the number of clients is increased to 350, the server is practically saturated, i.e., CPU is running at almost 100%. This is mainly due to the fact that the push server has to maintain all the state information about the clients and also manage the corresponding threads and connections. A push server based on long polling also needs to generate numerous request/response cycles to keep the connection alive, which impact the resources. With pull only the publish interval has a direct measurable effect on the performance. This shows us that if we want to use a push implementation even for a couple of hundreds of users, some load balancing solution and multiple servers are needed.

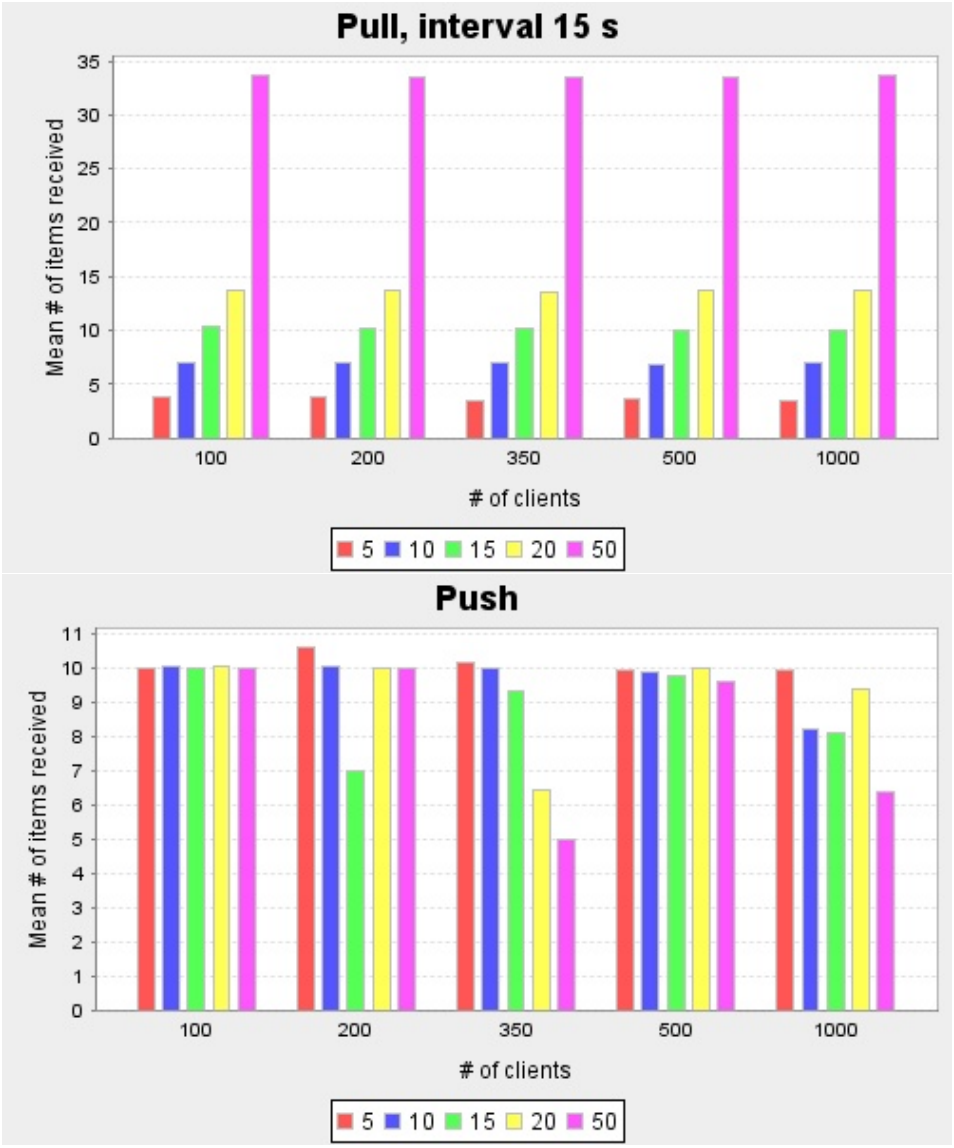


Figure 6.4: Mean Number of Received Publish Items.

6.3.3 Network Performance

In a pure pull system, the pulling frequency has to be high to achieve high data accuracy and data freshness. If the frequency is higher than the data generation interval, the pull client will pull the same data more than once, leading to some overhead.

In Figure 6.4 we see that with a publish interval of 50 seconds, pull clients receive approximately 35 messages, while we published only 10. In the same figure we see that Push clients received approximately a maximum of 10 messages. This means that, more than 2/3 of total number of pull requests was unnecessary. Furthermore, we see that the number of packages received does not depend on the number of clients.

If we look at Push graph in Figure 6.5, we notice that as the number of users increase, not all clients receive all 10 messages. The number of correctly received messages is quite well with 100 users, but, unlike the pure pull approach, it begins to degrade as the users increase. This shows that Jetty's Cometd implementation is not stable and scalable enough.

6.3.4 Data Misses

According to Figure 6.5, if the publish interval is 20 or 50 (i.e., larger than the pull interval), the client receives all the messages. However as we have discussed in the previous subsection, this will generate an unnecessary number of messages. Looking at the figure again, we see that when the pull interval is smaller than the publish interval, the clients will miss some updates, regardless of the number of users. So, with the pull approach, we need to know the *exact* publish interval. However the publish interval tends to change, which makes it difficult for a pure pull implementation. With push, when the number of clients is small, the client will receive all the messages. However if the number of clients increases, and the publish interval is large, some data loss starts to occur. This is again due to high number of idle threads, which affects the server performance.

6.3.5 Threats to Validity

We use several tools to obtain the data. The shortcomings and the problems of the tools themselves can have an effect on the outcome. In addition, implementation issues in the application server Jetty 6.1.2 might lead to the high CPU usage.

Another threat is the pull interval. We use only 1 pull interval, namely 15 seconds. Different pull intervals might have an influence on the performance of the server and the data coherence.

Clients can also have different environments (i.e.: the browser they use, the bandwidth they have, etc.). This can have an influence on the triptime variable. In order to avoid that, we used the same test-script in all the simulated clients and allocated the same bandwidth.

The time can also be a threat to validity. To measure the trip-time, the difference between the data creation date and data receipt date is calculated. However if the time on the server and the clients are different, this might give a false trip time. In order to prevent this problem, we made sure that the time on server and client machines are synchronized by using the same time server.

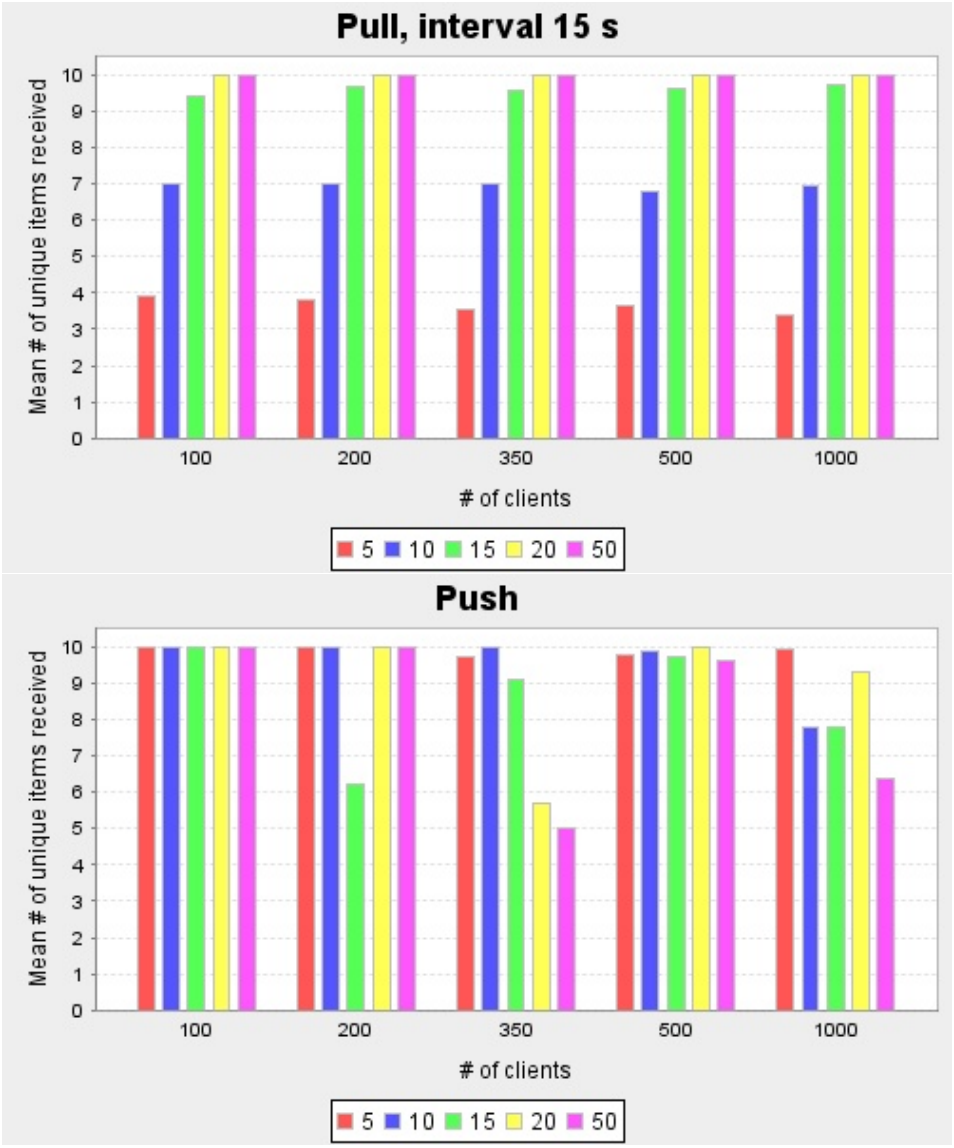


Figure 6.5: Mean Number of Received Unique Publish Items.

We measure the data coherence by taking the trip time. However, the data itself must be 'correct', i.e., the received data must be the same data that has been sent by the server. We rely on HTTP in order to achieve this "data correctness". However, additional experiments must include a self check to ensure this requirement.

6.4 Related Work

There are a number of papers that discuss server-initiated events, known as *push*, however, most of them focus on client/server distributed systems and non HTTP multimedia streaming or multi-casting with a single publisher [2, 24, 19, 3, 40]. The only work that focuses on AJAX is the white-paper of Khare [25]. Khare discusses the limits of the pull approach for certain AJAX applications and mentions several use cases where a push application is much more suited. However, the white-paper does not mention possible issues with this *push* approach such as scalability and performance. Khare and Taylor [26] propose a push approach called ARRESTED. Their asynchronous extension of REST, called A+REST, allows the server to broadcast notifications of its state changes. The authors note that this is a significant implementation challenge across the public Internet.

The research of Acharya *et al.* [2] focuses on finding a balance between push and pull by investigating techniques that can enhance the performance and scalability of the system. According to the research, if the server is lightly loaded, pull seems to be the best strategy. In this case, all requests get queued and are serviced much faster than the average latency of publishing. The study is not focused on HTTP.

Bhide *et al.* [8] also try to find a balance between push and pull, and present two dynamic adaptive algorithms: *Push and Pull* (PaP), and *Push or Pull* (PoP). According to their results, both algorithms perform better than pure pull or push approaches. Even though they use HTTP as messaging protocol, they use custom proxies, clients, and servers. They do not address the limitations of browsers nor do they perform load testing with high number of users.

Hauswirth and Jazayeri [22] introduce a component and communication model for push systems. They identify components used in most *Publish/Subscribe* implementations. The paper mentions possible problems with scalability, and emphasizes the necessity of a specialized, distributed, broadcasting infrastructure.

Eugster *et al.* [16] compare many variants of *Publish/Subscribe* schemes. They identify three alternatives: *topic-based*, *content-based*, and *type-based*. The paper also mentions several implementation issues, such as events, transmission media and qualities of service, but again the main focus is not on web-based applications.

Flatin [28] compares push and pull from the perspective of network management. The paper mentions the publish/subscribe paradigm and how it can be used to conserve network bandwidth as well as CPU time on the management station.suggests the 'dynamic document' solution of Netscape [32], but also a 'position swapping' approach in which each party can both act as a client and a server. This solution, however, is not applicable to web browsers. Making a browser act like a server is not trivial and it induces security issues.

As far as we know, there has been no empirical study conducted to find out the actual trade-offs of applying pull/push on browser-based or AJAX applications.

6.5 Conclusion

Our experiment shows that if we want high data coherence and high network performance, we should choose the push approach. However, push brings some scalability issues; the server application CPU usage is 7 times higher as in pull. According to our results, the server starts to saturate at 350-500 users. For larger number of users, load balancing and server clustering techniques are unavoidable.

With the pull approach, achieving total data coherence with high network performance is very difficult. If the pull interval is higher than the publish interval, some data miss will occur. If it is lower, network performance will suffer. Pull performs well only if the pull interval equals to publish interval. However, in order to achieve that, we need to know the exact publish interval beforehand. However, the publish interval is rarely static and predictable. This makes pull useful only in situations where the data is published frequently according to some pattern.

These results allow engineers to make rational decisions concerning key parameters such as pull and push intervals, in relation to, e.g., the anticipated number of clients. Furthermore, the experimental design allows them to repeat similar measurements for their own (existing or to be developed).

Chapter 7

Design of the application

As we have shown in Section 5.3.1, simply adding Cometd into an existing AJAX framework results in a non-optimal situation, where for every incoming push data, the client has to synchronize with the JSF server. This can be improved by providing several extensions to Jetty's Cometd implementation.

In this chapter we first present the design of Jetty Cometd and later discuss possible extensions. We also discuss possible deployment options.

7.1 Design of Jetty Cometd

7.1.1 Classes

Since we will extend Jetty (Version 6.1.2RC2), we need to describe its design first. Jetty Cometd contains the following classes (See Figure 7.1):

- **CometdServlet:** servlet that accepts the client and delegates tasks to other modules.
- **Bayeux:** the core of the system that keeps a list of clients, channels, transports, by acting as a Channel dispatcher. It also contains handlers that take care of different Bayeux stages.
- **Transport:** the interface representing the actual response to the client
- **AbstractTransport:** abstract class that implements Transport. It is extended by specific Transport classes, that depend on connection type of the client
- **SecurityPolicy:** class that defines authorization levels of the clients. It determines who can subscribe to a channel and who can broadcast a message
- **Client:** the representation of a push client. It contains the client's id and subscribed channel list. It also stores all the messages for a particular channel and remove them when they are delivered.
- **Channel:** the connecting element between the push consumers and the producer.

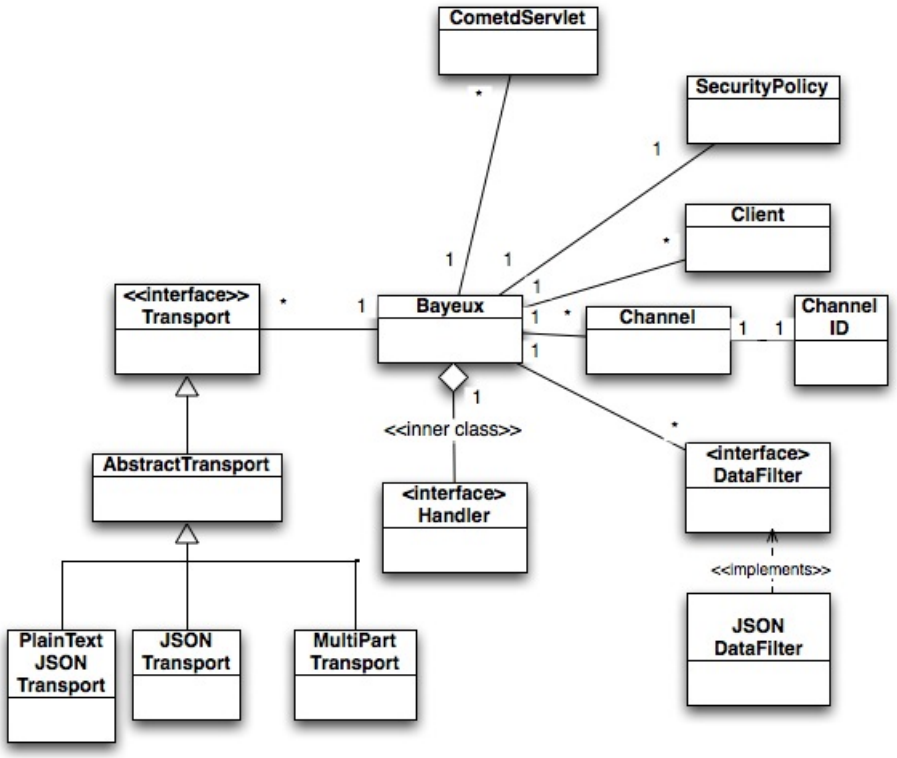


Figure 7.1: Jetty 6.1.2RC2 Class Diagram

- **ChannelId:** class that organizes a hierarchical channel structure, i.e.: /chatrooms/room1, /chatrooms/room2, etc.
- **DataFilter**: Interface that allows filtering of messages. It can be implemented depending on the message structure.
- **Handler:** Inner interface that takes care of a Bayeux stage. It is implemented by handlers that correspond to specific Bayeux stages. These handlers include HandshakeHandler, ConnectHandler, DisconnectHandler, SubscribeHandler, UnsubscribeHandler and PublishHandler.

In a routine Bayeux stage, the following events take place (See Figure 7.2):

- CometdServlet accepts a request from the user, and calls Bayeux.
- Bayeux in turn parses the request to determine the Channel, Client info, etc. and forwards the request to the appropriate Handler
- Handler processes the request and sends a reply to the user by using the Transport class.

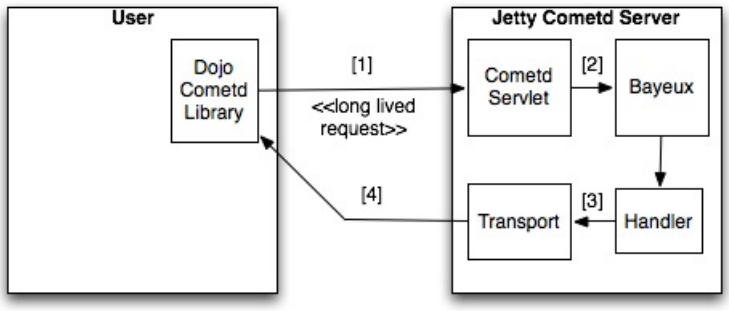


Figure 7.2: Sequence of events in Jetty Cometd

7.2 Extensions

7.2.1 Cometd + JSF Integration

As we have discussed in Section 5.3.1, a Push Server => Client => JSF Server sequence is not optimal. If the Push Server sends raw data to the client, and if that raw data changes a UI component on the client side (i.e. widget) that is synchronized with the server side, an event will be triggered. This will cause a ΔC to be sent to JSF server (See Figure 5.5). However, in most cases, the push server either will be in the same server, same machine or same LAN as the JSF Server. Therefore, it is more efficient if Push Server sends the update to JSF Server, forming a Cometd-JSF integration. This integration is depicted on Figure 7.3.

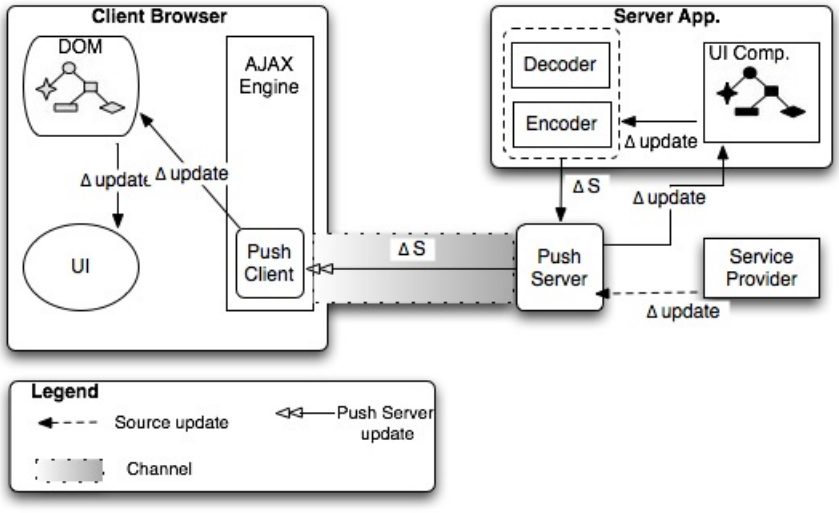


Figure 7.3: Proposed Cometd integration with SPIAR

In the figure, the following sequence of events takes place:

1. Service Provider sends Δupdate to Push Server
2. Push Server forwards Δupdate to JSF Server (Server App.)
3. JSF Server updates the UI Components of all subscribers and generates ΔS, which it later returns it to the Push Server
4. Push Server broadcasts the ΔS to all subscribers via the established channel
5. AJAX Engine processes the ΔS, as it would if it received it from the JSF Server, and updates the DOM and UI.

There are two advantages of this solution. First of all, it allows ΔS to be sent directly to the user in one step. The communication between the Push Server and JSF Server is negligible, comparing to the trip to the client.

Second advantage is the simplicity for the application programmer. Without this solution, the programmer has to write JavaScript functions in order to process the incoming push data and trigger an event. In our solution, no such function is needed, since the only response will be ΔS, which will be processed by the AJAX Engine.

7.2.2 Push Message Types

If we will develop an integration between the JSF Server and the Push Server, we need to distinguish different types of push messages. For instance, a push message that is sent directly to the client is different than one that requires a trip to the JSF Server first. The Push Server will recognize a particular push message sent by the Service Provider (SP) based on this message type (See also Section 7.2.3). We define the following message types:

- **data:** This message type is used when there are no server components present or widgets that get push updates need no synchronization with the server (See Figure 7.4). When the SP sends the new message enclosed in *data* tag, the Push Server will broadcast the data to all the subscribers directly.
- **clientDelta (ΔC):** This message type is used when widgets that receive push updates have server components. As we see from Figure 7.5, the SP sends a push update in ΔC form. Note that in SPIAR [31], only a JSF client can send ΔC, which represents a change to the Document Object Model (DOM) [1]. In our push server, Service Providers (clients with/without DOM) can also send ΔC. Push Server will send this ΔC to the JSF Server, receive a ΔS and broadcast it to all the subscribers. To see how we implemented this, please see Section 8.1.1.
- **jsfData:** This message type is used when widgets that receive push updates have server components, but the SP does not have access to JSF components, therefore cannot generate ΔC. As we see from Figure 7.6, SP sends a data enclosed in *jsfData* tag. After receiving this message, the Push Server will send it to the JSF Server and receive a ΔS, and later broadcast it to all the subscribers. Note that in order to handle this data and generate ΔS, some extensions are needed on the JSF Server. To see our implementation, please see Section 8.2.1

[1] http://www.w3.org/DOM/DOMTR

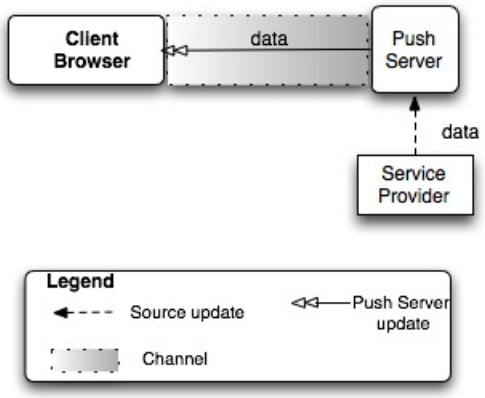


Figure 7.4: Push message type “data”

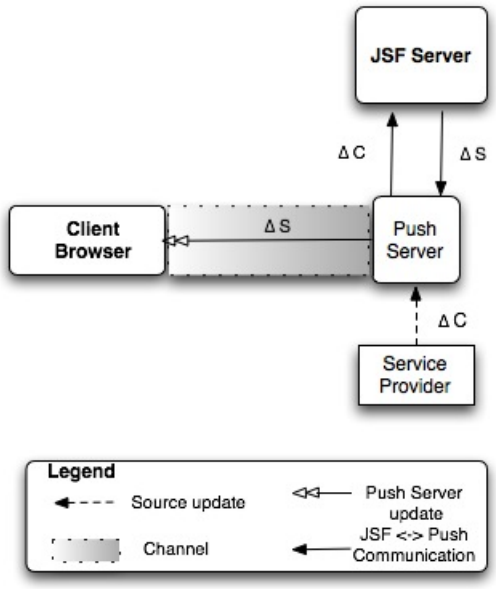


Figure 7.5: Push message type “clientDelta”

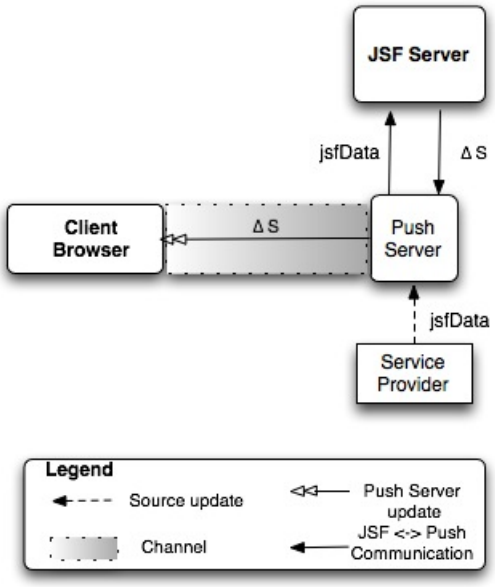


Figure 7.6: Push message type “jsfData”

7.2.3 Service Providers

As we see from Figure 7.3, Service Provider (SP) is the source of the update, in other words it generates push data. SP can be part of the system, or can be an external application. Therefore, we need to support two types of SP’s:

Internal Service Provider

An internal SP is part of the JSF application, in the sense that it has access to JSF components, attributes, etc. It is therefore able to send all types of push messages (See Section 7.2.2), including ΔC. For example, a JSF button press can be represented in ΔC as follows:

```
[evt=form:jsfButton | event | submit].
```

Our Internal SP has access to the clients’ JSF components and attributes. Therefore it can send changes on behalf of the clients and provide real-time event notifications. It can produce all push message types: *data*, *clientDelta* and *jsfData*.

External Service Provider

An external SP is not part of the application, has no access to application components and therefore cannot generate ΔC. This type of SP only sends raw data in String format. Therefore it can only produce *data* and *jsfData* message types.

7.2.4 Other extensions

Backbase requires us to implement sample applications that use our push implementation (See Chapter 9). One of these is a chat application. There are some features missing from Jetty Cometd that should be available for this type of application. They are listed below:

- List of channels: In an application, it is handy for a user to ask a list of channels. For example, in a chat application the user can ask a list of available chatrooms, and then join one.
- Notification: In a chat application, the users should be able to be notified if somebody joins/leaves a certain channel.
- List of subscribed users: It should be possible for a user to ask already subscribed users to a channel. This can be used for example a user to receive a "userlist" when he/she joins a channel.

7.3 Deployment Options

Another design decision that must be made is where to deploy the push server. In this subsection we discuss the options.

7.3.1 Push Server as a JAR file inside the JSF application

The first option is to deploy the push server as a JAR file and include it in every JSF application (WAR). In this option (See Figure 7.7), Service Provider calls the Push Server which is mapped to a different URI inside the JSF application's web.xml file. Push Server will then invoke JSF lifecycle, in which a server-delta will be returned. Push Server will later publish this new server delta to all the subscribers. In this scenario, Push Server has to access the JSF lifecycle by using the FacesContext object, which is available in the JSF server.

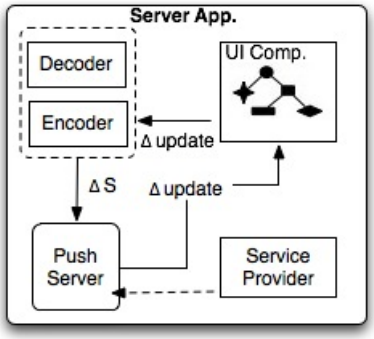


Figure 7.7: Push Server inside the JSF Application

This solution is very easy to install for the client. Including the JAR file in the lib directory is enough to obtain the needed push functionality. However, it comes with the following disadvantages:

- Scalability: Since the JAR is included in the same JSF app, Push Servlet will use threads from the same application server. This might cause scalability problems and an application that makes no use of push, but is located in the same server will also suffer.
- Maintenance: If a small change is made in the Push Servlet, the JAR file has to be recreated and redeployed for every JSF app.

7.3.2 Push Server as a WAR file

The second option is to deploy Push Server as a separate WAR file. As we can see from Figure 7.8, Push Server is a separate WAR file and can be deployed in the same server as the JSF App, but also in another server. Here, Service Provider calls the Push Server, with the URL of the JSF server. Depending on the URL, the Push Server either calls JSF Server 1 or JSF Server 2. Note that the Push Server has to reach the JSF context, in order to send the Δupdate and receive a ΔS. Please see Section 8.1.1to see how we implement this.

This solution has the following advantages:

- Multiple JSF apps: Since it runs separately, the Push Server can serve multiple JSF apps, running on different app servers.
- Flexibility and Tuning: Since the Push Server can be deployed in a separate app server, tuning of the server can be performed easily without affecting the JSF app.
- Scalability: The user can use a different app server for the Push Server, so that it won't use the threads of the JSF app server. For example Tomcat 5.5 can be used for the JSF applications, while Push Server can be ran under Jetty.

The solution comes with a disadvantage. Push Server will call JSF Server of every application via HTTP/TCP. If the Push Server is deployed on a separate machine and if there is a big number of users, this might create huge amount of network traffic. However this should not cause big problems if the machines are located in the same LAN. If the Push Server is deployed in the same machine as the JSF apps, the HTTP/TCP call is transformed into a local memory call. Most modern operating systems implement a loopback interface [36], where the IP datagram is transformed into a memory call in network layer, instead of using ARP to get the destination address and later broadcasting it into the ethernet.

Conclusion

According to the criterion above, including the Push Server as a WAR file seems to be a more scalable and more flexible option. However, the implementation should be flexible enough, so that the Push Server can be packaged as a JAR and included in the JSF Application if needed.

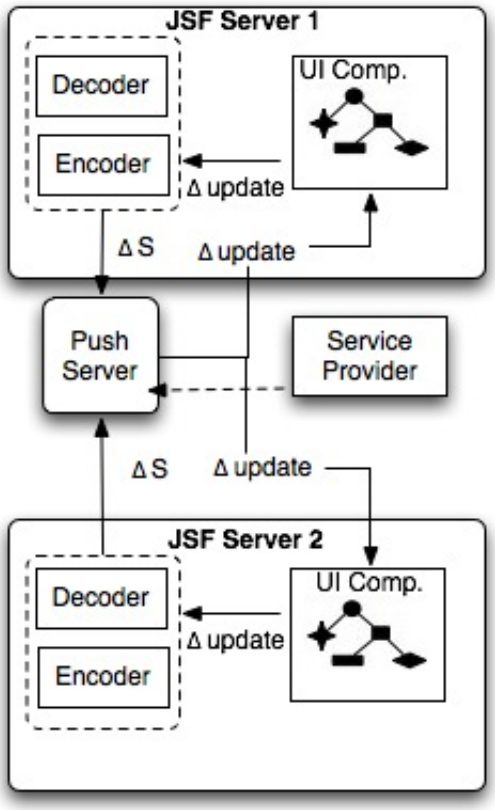


Figure 7.8: Push Server as a WAR.

7.4 Message Format and Protocol

Section 4.2.9 states that the messages should be wrapped in XML. Jetty uses JSON in messaging. This requires us to redefine everything in XML format. For example, the following shows a reconnect request in Backbase Push:

```
<request>
     <message>
          <channel>/meta/handshake</channel>
          <version>0.1</version>
          <minimumVersion>0.1</minimumVersion>
          <supportedConnectionTypes>
              <connectionType>long-polling</connectionType>
              <connectionType>callback-polling</connectionType>
          </supportedConnectionTypes>
     </message>
</request>
```

Jetty equivalent of this message is as follows:

```
[
 {
  "channel":  "/meta/handshake",
  "version":  1.0,
  "minimumVersion":  1.0,
  "supportedConnectionTypes":  ["long-polling", "callback-polling"],
 }
]
```

7.5 Conclusion

Jetty Bayeux provides push capability for Client-to-Client communication, however it requires extensions in order to support an environment with server components (such as JSF Server). In this chapter we first introduced important components of Jetty Bayeux library. We later presented an architecture showing how the JSF server and the Push server can be integrated. We discussed several push message types, different Service Providers, various extensions and possible deployment scenarios. We concluded that deploying the Push Server as a WAR file is a better option, in terms of flexibility and scalability.

Chapter 8

Implementation and Testing

Chapter 7 introduced some concepts in order to introduce JSF + Push Server integration and several other extensions. In this chapter we discuss how we realized these concepts into a working implementation. Figure 8.1 shows the required components in order to implement a Cometd solution. First of all, several extensions are needed on Jetty Cometd implementation, in order to have a Cometd server that fulfills the requirements of Backbase. Additionally a client library is needed on user browser, and two new classes (JSFPushServlet and NotifyPushData) on the JSF server.

In the following sections, we discuss all these components in detail.

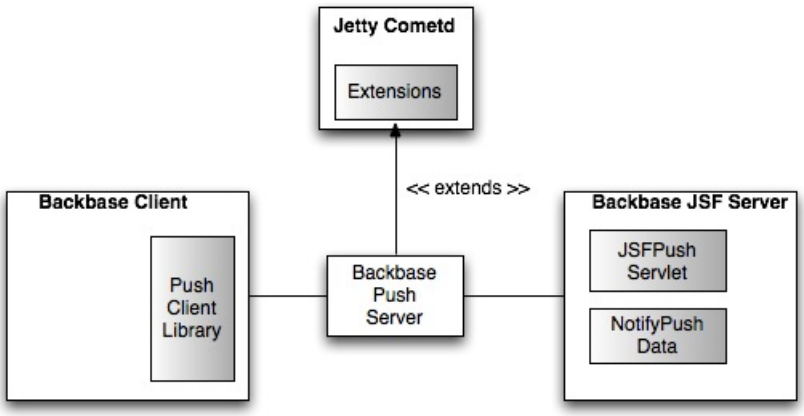


Figure 8.1: Components of Backbase Push Server

8.1 Jetty Cometd Extensions

In order to implement the concepts we have introduced in Chapter 7, we needed to extend Jetty's Cometd implementation (version 6.1.2RC5). However Jetty Cometd contains several limitations that prevented us to simply include it as a JAR file and provide classes that extend it:

- Some classes have private constructors. In order to extend them those modifiers should be changed.
- Some classes have protected constructors. In order to extend them either those modifiers should be changed or the subclasses should be in the same package.
- JSON parsing happens in every function, there is no central class that performs this task. This makes it very difficult to use another message wrapper (such as XML) and requires direct modification of Jetty code, instead of extending it. Backbase requires XML to be used in push communication, which makes it impossible to just include Jetty Cometd as a library.
- If a client performs an undesired behavior, an Exception is thrown and the client cannot reconnect, requiring a server restart
- Versions are unstable. There is a lot of refactoring between two minor releases in a week. API changes very quickly
- Handlers which are responsible of different request (such as handshake, connect, publish) are implemented as inner classes in a single class. Extending or adding new handlers is not possible without modifying this single class. We do need to add extra handlers in order to support JSF integration and other extensions.
- There are many unnecessary synchronize blocks throughout the code, which cause performance issues
- Implementation includes JDK 1.5 elements. Backbase requires support for customers using JDK 1.4.

Because of these limitations, instead of including Jetty Cometd as a JAR file and providing extensions, we had to modify the code and extend it. We also had to perform bug fixes which we have found with testing, code optimization and conversion to JDK 1.4. The following subsections present our extensions.

8.1.1 JSF Extension

In Section 7.2.1, we introduced an approach where the Push Server contacts the JSF Server and updates the UI components of all the subscribed users. As we see from Figure 7.3, Push Server contacts the JSF Application to update the UI components. However, if session scope is used for a Backing Bean, this will pose a problem. In the following subsection we discuss this issue.

Session Problem

As we mentioned in Section 3.1.2, the world-wide web adopts a *stateless* approach to client-server communication. The browser sends a request to the server, the server returns a response, and then neither the browser nor the server has any obligation to keep any memory of the transaction. In order to keep a "state" for the client, servlet containers augment the HTTP protocol to keep track of a session, that is, repeated connections by the same client [21]. There are various methods for session tracking.

The simplest method uses cookies: name/value pairs that a server sends to a client, hoping to have them returned in subsequent requests. In JSF, a *session scope* can be defined, so that the client's state will be valid through the whole session.

In case a backing bean has a session scope, one SESSION-ID will be created for each user. This means that, for each subscribed user, every client widget that contains server side components needs to be synchronized for each push update. Therefore, Push Server has to update each user's UI components. However, as mentioned in Section 4.1.1, JSF Server recognizes the user browser by the supplied SESSION-ID, which is sent by the browser in every JSF request. With this id a corresponding session object is obtained. Without the user sending this SESSION-ID along the request, JSF Server cannot identify and thus cannot access the UI components for the user.

Solution

Our approach is that the Push Server makes a JSF request, with the user's session id. The Push Server can act as a JSF client (user browser) with the use of HTTPClient library[1]. In order to make this an extensible approach, we have implemented this feature in a separate class called HTTPConnector. With this extension, the following events take place (See also Figure 8.2):

1. User browser makes a JSF request to the JSF Server
2. JSF Server assigns a session-id for this user
3. Push Client (which resides within the user browser) sends previously received JSF sessionID to Push Server. This id is registered by ConnectHandler (not shown in the figure).
4. Service Provider publishes ΔC to Push Server, which is delegated to Publish-Handler
5. PublishHandler forwards this ΔC with a list of subscribers to HTTPConnector
6. For each subscriber, HTTPConnector makes a JSF Request to JSF Server with the subscriber's session-id.
7. JSF Server processes this ΔC, as it would from any JSF client, and returns a ΔS.
8. HTTPConnector sends the received ΔS to the subscriber, which will be processed by the AJAX engine.

This way, whenever a new push data is published by the Service Provider, the user will receive a ΔS from the push server. All the interactions between Push Server and JSF Server is transparent to the user, and no JavaScript programming is needed for the user to process the incoming push data. We have tested this solution successfully under different application servers including Tomcat 5.5, 6.0, Jetty 6.1.2, Weblogic 9.1 and Websphere 6.

[1] `jakarta.apache.org/httpcomponents/httpclient-3.x/`

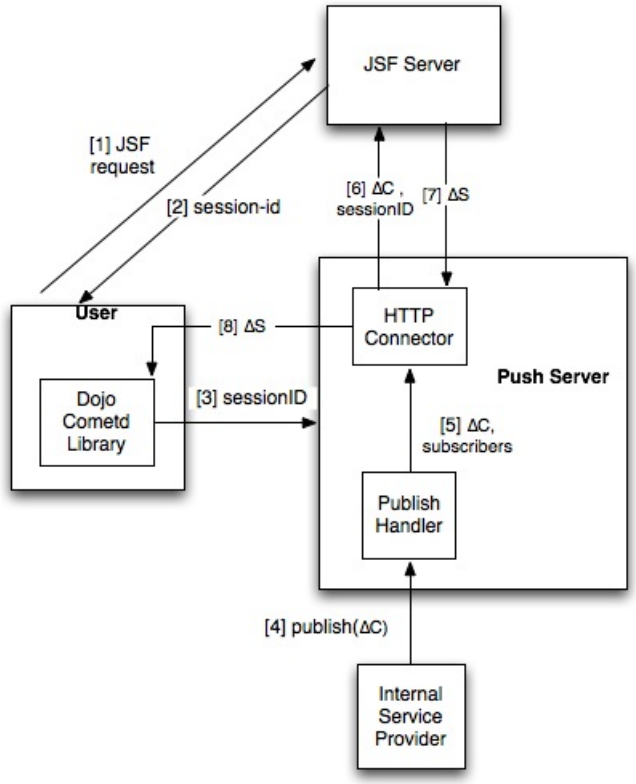


Figure 8.2: Implementation of JSF-Push Integration and HTTPConnector

8.1.2 Other extra features

As we mentioned in Section 7.2.4, there are some features missing from Jetty Cometd that should be available for a chat application. These were a list of channels, join/leave notification, and a list of subscribed users. We realized these features by the following implementations:

- **ChannelListHandler:** In an application, it is handy for a user to ask a list of channels. For example, in a chat application the user can ask a list of available chatrooms, and then join one. ChannelListHandler extends the Handler class and sends a list of channels when requested.

- **Join/Leave Notification:** In a chat application, the users should be able to be notified if somebody joins/leaves a certain channel. In order to enable that, we added an optional "notification" field into the "subscription request" message. If the user sets notification to "true" in a subscription request, the server will notify the user of user join/leaves for that particular channel. This feature required modifications in SubscribeHandler and UnsubscribeHandler.

- **UserlistHandler** It should be possible for a user to ask already subscribed users to a channel. This can be used for example a user to receive a "userlist" when

he/she joins a channel. UserlistHandler extends the Handler class and sends a list of users when requested.

8.2 Extensions on JSF Server

8.2.1 Raw data support

As shown in Figure 8.2, Service Provider (SP) sends a ΔC to Push Server. However, as discussed in Section 7.2.3, only Internal SP has access to user components, thus only Internal SP can generate ΔC. In order to allow External SP's, we need to support raw push data (jsfData) as well. However the mechanism depicted in 8.2 will not work, since in Step 6, JSF Server expects ΔC, not *jsfData*. In order to add *jsfData* support, we need an extension on the JSF Server. We define the following two classes:

- NotifyPushData: Interface that contains a single method `newData(Object data)`. Any backing bean implementing this class, will receive the new data when their *newData* method is called.
- JSFPushServlet: accepts raw data from Push Server, notifies the backing beans by calling their *newData* method. It later calculates a ΔS and returns it to Push Server.

In this scenario the following events take place (See also Figure 8.3):

1. User Browser sends previously received JSF sessionID to Push Server. This id is registered by ConnectHandler (not shown in the figure).
2. Service Provider publishes raw data to Push Server, which is delegated to PublishHandler
3. PublishHandler forwards this raw data with a list of subscribers to HTTPConnector
4. For each subscriber, HTTPConnector calls JSFPushServlet with the subscriber's session-id and the raw data.
5. JSFPushServlet, which resides inside the JSF Application, catches the data. It calls all Backing Beans (by means of *newData* method) in any scope that implement NotifyPushData interface. At this stage Backing Beans will process the incoming data, which is specified by the application programmer
6. JSFPushServlet calculates ΔS and returns it to HTTPConnector
7. HTTPConnector sends the received ΔS to the subscriber, which will be processed by the AJAX engine.

If we compare Figure 8.3 with Figure 8.2, we see that the only extra steps are Step 5 and 6, where the JSFPushServlet has to update the server components and calculate a ΔS.

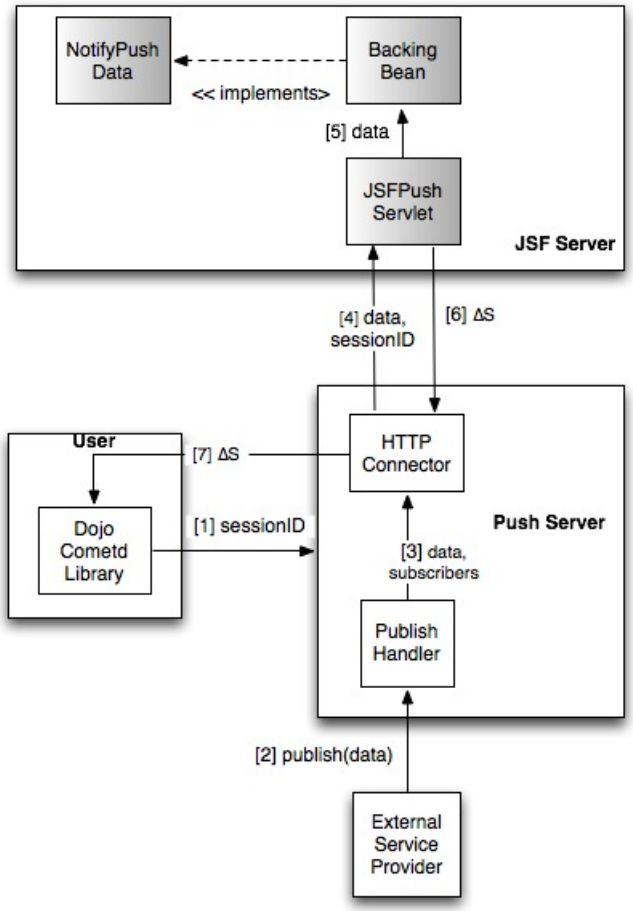


Figure 8.3: Raw data support in JSF Integration

8.3 Extensions on the Client

We use XML to wrap push data. Dojo Cometd cannot be used as a Push client library, since it only supports JSON. Backbase client team provided us a simple proof of concept JavaScript library that implements Bayeux stages in order to subscribe/publish data. The delivered library only supported client-to-client communication, which required us to extend it for JSF support. We will discuss the library's limitations in Chapter 10.

8.4 Testing and Coverage

Backbase requires us to test Push Server by developing sample applications. Nevertheless, we tested BackbaseBayeux class by using JUnit[2]. BackbaseBayeux is the kernel of the system and is responsible for administration of channel and classes. After

[2] www.junit.org

testing BackbaseBayeux, we obtained the coverage with EclEMMA[3].

The result is given in Figure 8.4. The class coverage is 93%. Some methods (such as *initialize()*) show lower coverage, since these can only be tested when a ServletContext variable is available, which is difficult to realize with JUnit.

Element	Coverage	Covered Instructions	Total Instructions
BBPushSVN	38.3 %	2558	6679
src	38.3 %	2558	6679
com.backbase.push	53.1 %	1335	2514
BackbaseBayeux.java	93.0 %	726	781
BackbaseBayeux	93.0 %	726	781
BackbaseBayeux()	100.0 %	142	142
addFilter(String, DataFilter)	84.2 %	16	19
addHandler(String, Handler)	100.0 %	7	7
addTransport(String, Class)	100.0 %	6	6
getChannel(BBChannelId)	100.0 %	5	5
getChannel(String)	100.0 %	15	15
getChannel(String, boolean)	92.1 %	35	38
getChannelId(String)	100.0 %	21	21
getChannelIdCache()	100.0 %	3	3
getClient(String)	87.0 %	20	23
getClientIDs()	100.0 %	4	4
getClientTimeoutMs()	100.0 %	3	3
getHandler(String)	100.0 %	6	6
getRandom(long)	100.0 %	15	15
getRootChannel()	100.0 %	3	3
getSecurityPolicy()	100.0 %	3	3
getTimeOnServer()	100.0 %	5	5
getTimeout()	100.0 %	3	3
getTimeStamp()	100.0 %	3	3
handle(BBClient, Transport, Eleme	100.0 %	53	53
handleError(HttpServletResponse	100.0 %	40	40
initialize(ServletContext)	52.2 %	35	67
isInitialized()	100.0 %	3	3
newClient(String)	100.0 %	15	15
newTransport(BBClient, Element)	94.1 %	95	101
publish(BBChannelId, BBClient, St	93.8 %	75	80
removeClient(String)	89.3 %	25	28
setChannelIdCache(Hashtable)	100.0 %	4	4
setClientTimeoutMs(long)	100.0 %	4	4
setSecurityPolicy(BBSecurityPolicy	100.0 %	4	4
setTimeout(long)	100.0 %	4	4

Figure 8.4: Backbase Bayeux class test coverage

8.5 Used Libraries

We have used the following external libraries in order to realize our Bayeux implementation.

- **dom4j 1.6.1**: dom4j[4] is an open source library for working with XML, XPath

[3] `www.eclemma.org/`

[4] `www.dom4j.org`

and XSLT on the Java platform using the Java Collections Framework. We used it for XML parsing and String-XML conversion.

- **jaxen 1.1**: The jaxen project[5] is a Java XPath Engine. We used jaxen with dom4j to evaluate XPath expressions and to parse XML from incoming push messages.
- **jetty-continuations 6.1.2rc2**: Jetty Continuations allow less thread usage in a Jetty webserver. We included it in our implementation as well.
- **Apache Commons HttpClient and Codec**: HTTPClient and Codec libraries were needed to implement JSF-Cometd integration. Our push server used HTTP-Client to make requests into the JSF server on behalf of the clients.

8.6 Final Class Diagram

After applying all the mentioned extensions, we obtain the class diagram in Figure 8.5. In this figure we see the following changes, comparing with Jetty's class diagram of Figure 7.1:

- All classes are modified in order to support XML processing. Therefore they received a different name, i.e. BBClient instead of Client.
- Handler interface and its implementers are not inner class anymore, which makes it easier to extend and maintain.
- Connector interface and HTTPConnector classes are added, to realize Push Server - JSF Server integration. PublishHandler is also extended to realize this integration and support different types of publish messages, which was mentioned in Section 7.2.2.
- ChannelListHandler and UserListHandler are also added to realize the features mentioned in Section 7.2.4.
- XMLTransport and XMLFilter extensions are added in order to support XML, instead of JSON.
- XMLHelper class is added to support XML parsing and XML push message creation.

For a full list of package listing See Appendix B.

8.7 Conclusion

In this chapter we first discussed the needed changes and extensions in order to realize a JSF-Push integration. We discussed difficulties such as session scope and presented our solution HTTPConnector. We mentioned several other extensions such as ChannelList, which are needed for the sample chat application. We presented new

[5] www.jaxen.org

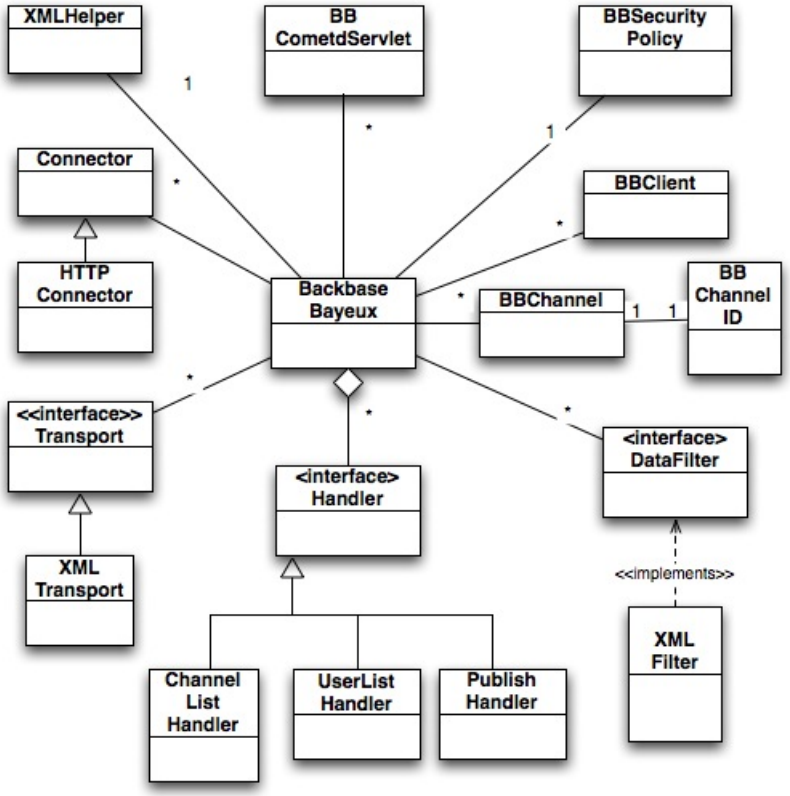


Figure 8.5: Jetty based Backbase Cometd Class Diagram

components NotifyPushData and JSFPushServlet in order to allow raw data support (jsfData) with server components. Finally we discussed the final class diagram and all the components.

Chapter 9

Sample Push Applications

This chapter will appear as a whitepaper which will be published in www.backbase.com.

In Chapter 8, we have presented the implementation of a push server. In this chapter we will present sample applications that use our Jetty-based implementation, but also DWR, which was mentioned in Chapter 5. First, we display two different use cases for client-to-client communication with different types of publishers (See Section 4.2.5): we present a chat application (use case for subscribed publisher), and a stock ticker application (use case for nonsubscribed publisher). Later, we present two examples demonstrating Push-JSF integration.

9.1 JSF Chat application

In order to demonstrate the push feature, we first need a sample JSF application. Our first use case is a chat application. In this application, there are many subscribers and many publishers. Figure 9.1 shows the architecture of our application. In this design, the user side consists of Backbase widgets and a JSP page. On the server side, we have the UI components, which are contained in User Bean. In our chat scenario, users communicate via channels, so we define two classes to implement this feature: a Channel object representing a channel with subscribers, and a ChannelManager object containing multiple channels.

The initial state of the JSF page is shown in Figure 9.2. A button with the label "Connect" is rendered. When we click the button, an ActionEvent is sent to the server as ΔC, and a ΔS is received, which results in the state shown in Figure 9.3. As we see from the figure, the button label is changed to "Disconnect" and a form is displayed for login. Clicking the "login button will result in another ActionEvent, which will result in a "logged in" state (See Figure 9.4). Now the client is logged in and a list of available channels is received. The user can add a channel to the list, by typing in the name of the channel and clicking the "plus" button. This will fire another ActionEvent, and the channel will be added to the list of ChannelManager (See Figure 9.5). If the user clicks on any of the channels, the server will create a new tab on the tabBox item (See Figure 9.6).

Push support can be added to several components of this application. For example, we can introduce push support to the channel list, so that when a user adds a channel, it

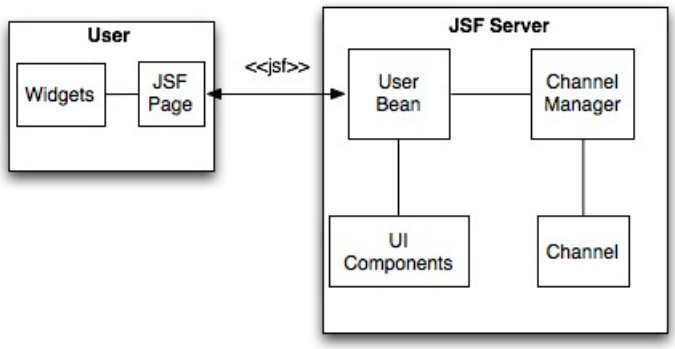


Figure 9.1: The architecture of the chat application

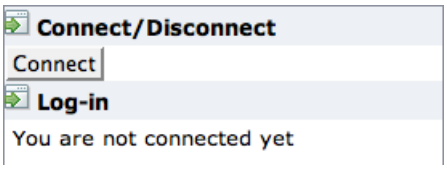


Figure 9.2: The startup state of the chat application

Figure 9.3: The state of the chat application when the client is connected

will be broadcasted to all the users and they will be notified of the availability of a new channel. However, the most suitable part for push is chat box itself (receiving/sending messages). If a user sends a message, it should be delivered to other users immediately. In our push extensions, we implemented this feature. We discuss the push extensions in the following subsections.

9.1.1 Bayeux Extension

Backbase requires a client-to-client push solution for this use case, instead of a full JSF-Push integration. This requires us to develop the architecture depicted in Figure 9.7. In order to add push support, the user needs to deploy the Push Server (WAR file) and include Push Client in the JSP page. Push Server communicates directly with the user, via the Push Client. Note that the JSF connection is not disturbed with push extension: Backbase Client Runtime can communicate with JSF Server without any

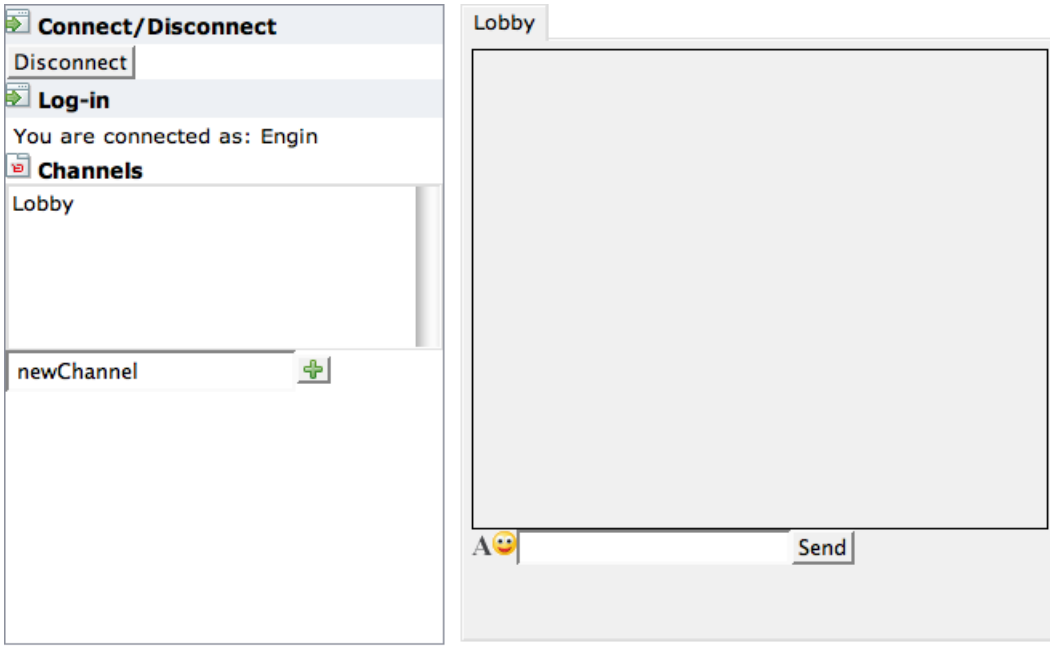


Figure 9.4: The state of the chat application after login

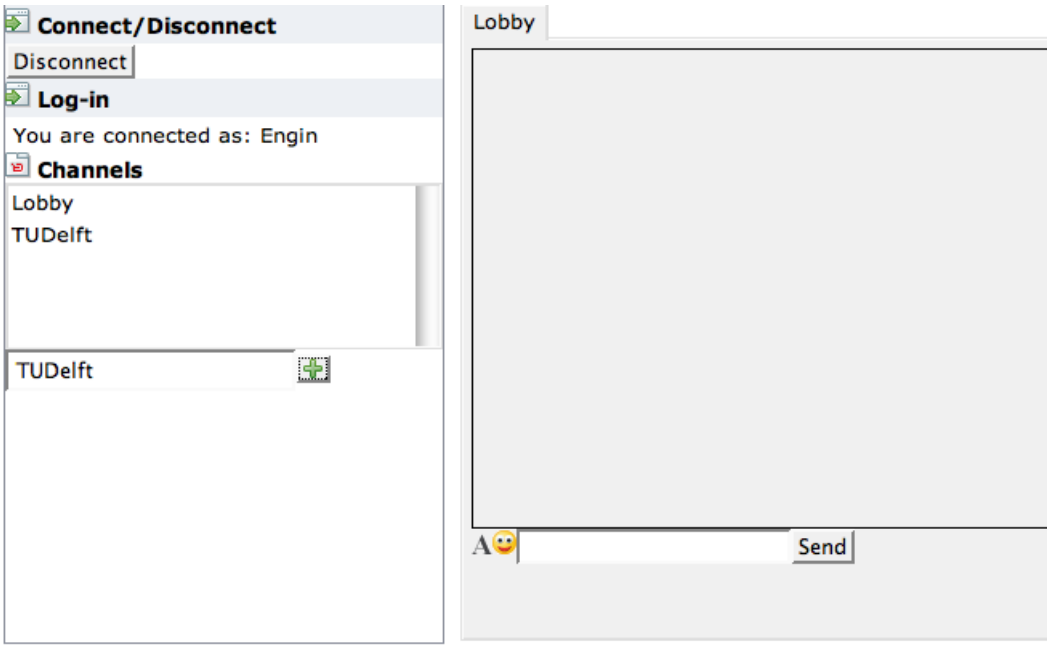


Figure 9.5: The state of the chat application after a channel creation

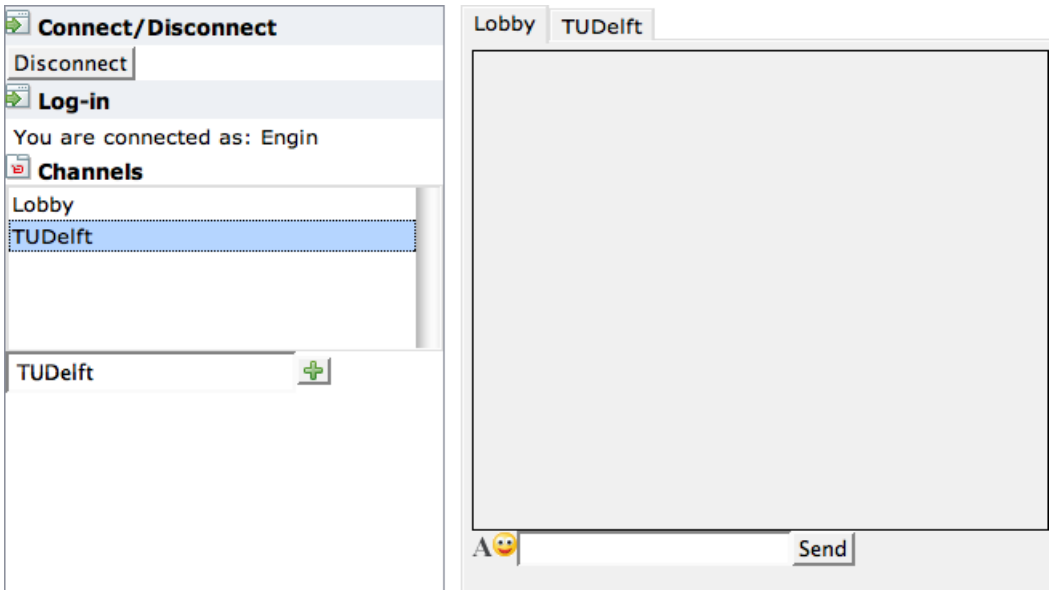


Figure 9.6: The state of the chat application after a channel join

interruption. In this scenario, all subscribers are also publishers: the arrow between the Push Client and the Push Server is two sided. The publishers send their update enclosed in“data” tag (See Section 7.2.2 for push message types).

When a push message is received, the user has to process it and update the DOM. This requires JavaScript programming on the client side. After bringing these extensions, the users can chat with push support (See Figure 9.8).

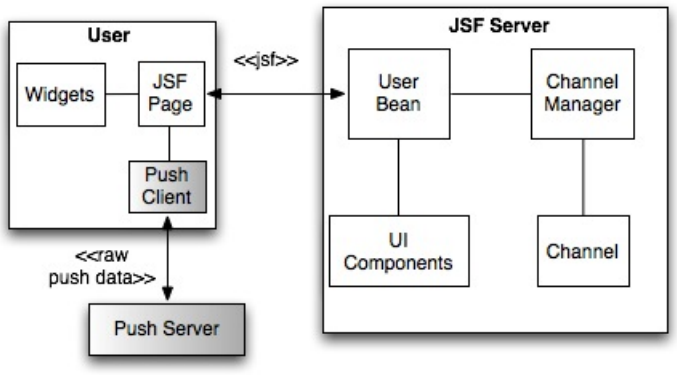


Figure 9.7: The architecture of chat application with Bayeux extension

Figure 9.8: Push support (Bayeux/DWR) in the chat application

9.1.2 DWR Extension

As mentioned in Section 5.2, in order to add DWR push support, the user has to perform some steps. We list them below:

- Library inclusions: DWR jar file has to be included in the JSF application's library directory. DWR's JavaScript engine and utility functions has to be included in the JSP page.
- Push mode setting: on <body> tag of the JSP page, `onload="dwr.engine.setActiveReverseAjax(true);"` has to be set
- A method has to be implemented in User Bean, in order to handle push data and broadcasting to others. After this method, a JavaScript file will be generated for the User Bean. This has to be included in the JSP page as well.
- The bean has to be added to DWR's configuration file.
- DWR's servlet and necessary filters have to be added to application's configuration file (i.e.: web.xml). Necessary servlet-mapping has to be performed.

After performing these steps, we obtain the architecture depicted in Figure 9.9. DWR Client (JavaScript Engine) and DWR Servlet communicate directly with each other, while Backbase Client Runtime and Backbase JSF Server still communicate via JSF. In DWR's solution, with each push update, the User Bean is updated. However, to reflect these changes on the client, either JavaScript programming or an explicit JSF

synchronization is needed. After performing the necessary steps, the users can chat with push support (See Figure 9.8).

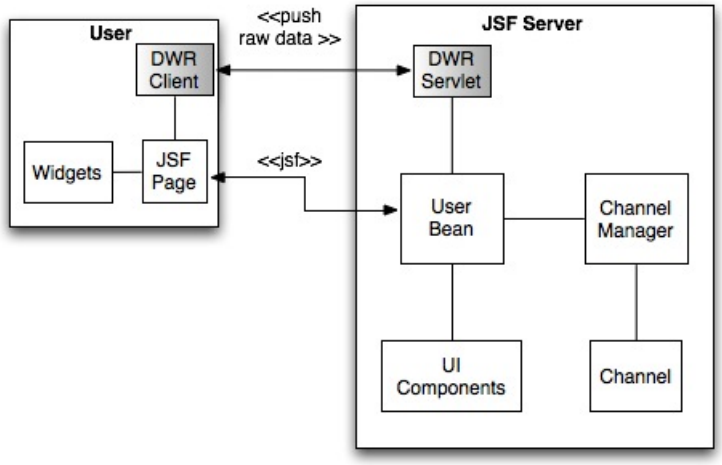


Figure 9.9: The architecture of the chat application with DWR extension

9.2 JSF Stock Ticker application

Stock ticker is an application where latest stock updates are displayed (near) real-time. It is a good use case for nonsubscribed publisher scenario (See Section 4.2.5). Figure 9.10 shows the architecture of our application. The client side is formed of a JSP page and Backbase widgets. The server side is rather simple comparing to the architecture of the Chat application; it only contains UI components, contained in User Bean. Figure 9.11a shows the JSP page after the initial load. It displays a table with 3 companies, the current stock status and the change given as percentage.

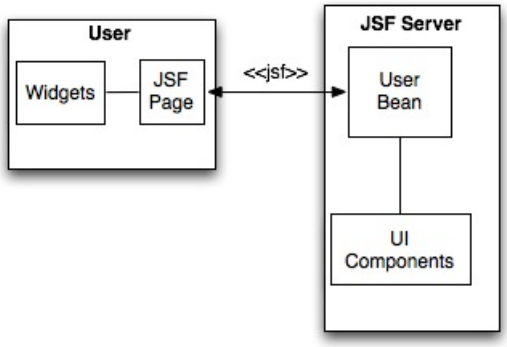


Figure 9.10: The architecture of the stock ticker application

Company	Status	Change
IBM:	15	0%
Sun:	15	0%
Backbase:	15	0%

(a) Initial state

Company	Status	Change
IBM:	29	163%
Sun:	12	9%
Backbase:	10	-33%

(b) After some push updates

Figure 9.11: Stock Ticker application

Push support in such an application is essential. As we have concluded in Section 6.5, if the pull interval is higher than the publish interval, some data miss will occur. If it is lower, network performance will suffer. Pull is useful only in situations where the data is published frequently according to some pattern, in stock updates this is rarely the case. The following subsections specify how push support can be introduced to the application.

9.2.1 Bayeux Extension

Backbase again requires a client-to-client push solution for this use case, instead of a full JSF-Push integration. This results in the architecture depicted in Figure 9.12. The required changes are identical to those mentioned in Section 9.1.1. Push Client has to be included in the JSP page and the the Push Server (WAR file) should be deployed. In this scenario, there is only one publisher and many subscribers, therefore the arrow between Push Server and Push Client is one sided. Service Provider is the class that generates push message with “data” tag. When the data is received, Push Server will send it directly to the User via Push Client. JSF connection is still not disturbed with the push extension: Backbase Client Runtime can communicate with JSF Server without any interruption. When Service Provider begins to publish data, the DOM will be updated (See Figure 9.11).

9.2.2 DWR Extension

Apart from the steps mentioned in Section 9.1.2, the user has to define a Publish Servlet (See Figure 9.13). This servlet is responsible for:

- processing the incoming push data
- accessing the bean by using JSF’s FacesContext object
- updating the bean
- broadcasting the data to all the subscribers

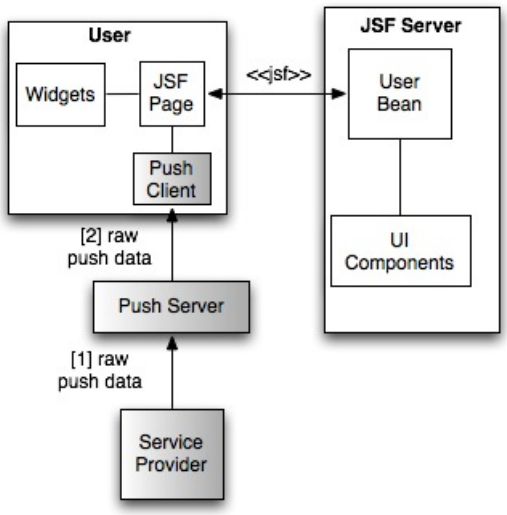


Figure 9.12: The architecture of Stock Ticker with Bayeux extension

DWR Client (JavaScript Engine) and DWR Servlet communicate directly with each other, while Backbase Client Runtime and Backbase JSF Server still communicate via JSF. Just as in the chat application, to reflect the changes on the bean to the client, either JavaScript programming or an explicit JSF synchronization is needed. See Figure 9.11 to see the new state after the DOM update.

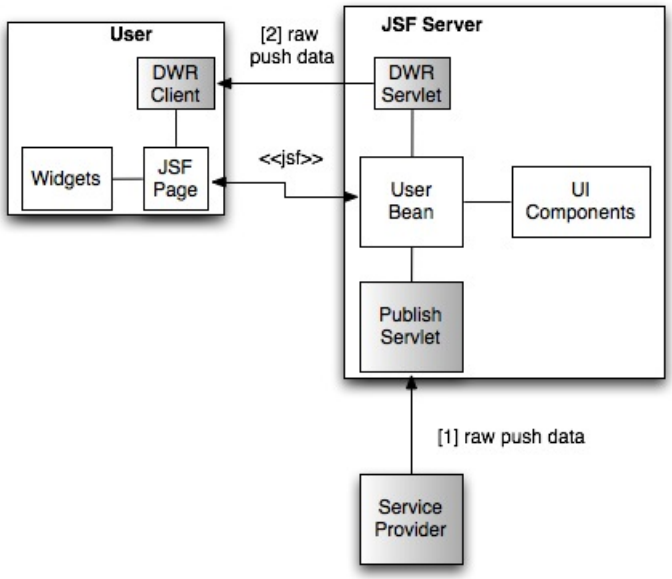


Figure 9.13: The architecture of Stock Ticker with DWR extension

9.3 Blank Application

In the previous sections, we showed sample applications that use "data" tag, which is designed for client-to-client push communication. As mentioned in Section 7.2.2, Push Server should also support *clientDelta* and *jsfData* tags as well, in order to support JSF-Push integration. To demonstrate this integration, we chose the simplest JSF application that comes with Backbase JSF 4.01 framework: blank application.

9.3.1 The Application

Blank app. (See Figure 9.14a) consists of a "commandButton" with the label "Backbase BV". If the button is clicked, an ActionEvent will be sent in the form of ΔC. Upon the receipt of this ΔC, server side Backing Bean will change the label of the button to "TU Delft" and send a ΔS. Backbase Client Runtime will update the button with the incoming ΔS (See Figure 9.14b). The architecture of this application is the same as Stock Ticker (Figure 9.10).

(a) Initial state (b) After a click on the button

Figure 9.14: Backbase JSF 4.01 blank application

In the following subsections we present sample applications that extend this JSF application with a JSF-Push integration. We first show the version with an internal Service Provider (SP). Later, we demonstrate it with an external SP.

9.3.2 JSF-Push Integration with an Internal SP

As stated in Section 7.2.3, an Internal SP is a data publisher that has access to the application's components and therefore can send ΔC. If we add this component to our current design, we obtain the architecture depicted in Figure 9.15. In this architecture, the following events take place:

1. Internal SP publishes a new message in ΔC form. In our example we submit `[evt=form:jsfButton | event | submit]`.
   This is what the browser is actually sending when the button is pressed by the user.

2. Push Server calls JSF Server with this ΔC.

3. JSF Server Returns a ΔS to Push Server

4. Push Server sends this ΔS. Backbase Client Runtime will process ΔS and update the DOM.

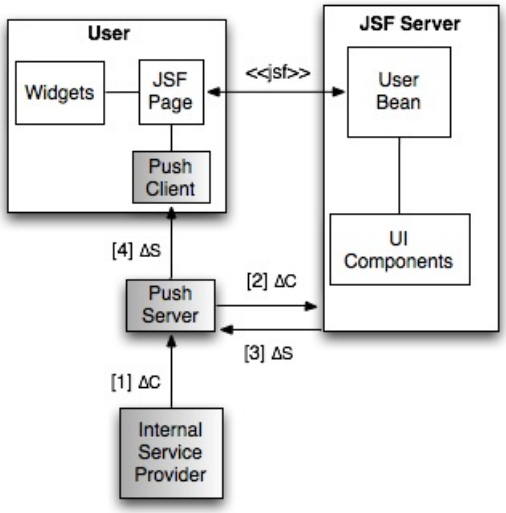


Figure 9.15: JSF-Push Integration with Internal SP

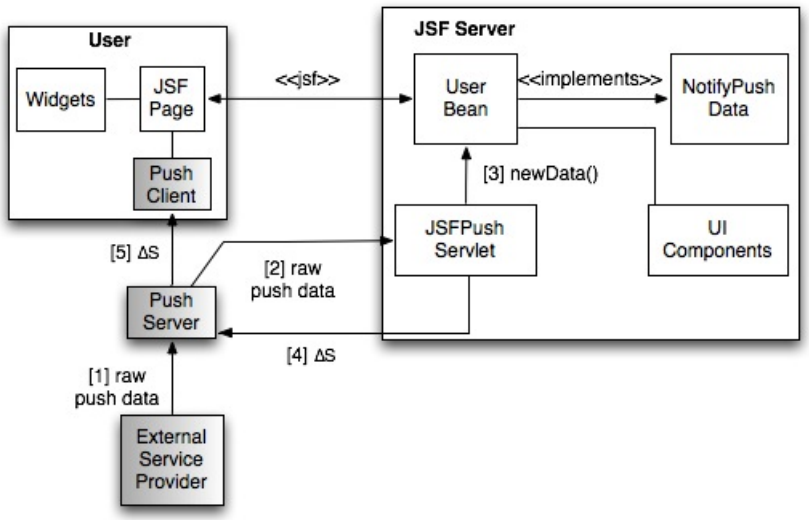


Figure 9.16: JSF-Push Integration with External SP

Figure 9.17a shows the initial state of the blank app with JSF-Push integration. We see the button "Backbase BV" above. Below, we have a form that simulates an Internal SP. Here, the publisher can specify the ΔC, address of the JSF Server and the channel. Figure 9.17b shows the state of the application after data is published. We see that the button is changed to TU Delft, and ΔS (Server Delta) is displayed below.

9.3.3 JSF-Push Integration with an External SP

An External SP is a data publisher that has no access to the application's components and therefore can only send raw data. In our design this type of data that needs to be sent to JSF Server is called *jsfData*. If we add External SP to our current design, we obtain the architecture depicted in Figure 9.16. In this architecture, the following events take place:

1. External SP publishes *jsfData* in String form. In our example we submit "hello world!".
2. Push Server calls JSFPushServlet with this data.
3. JSFPushServlet gives this new data to all classes in JSF context that implement NotifyPushData interface. In our case this is User Bean. It is now up to the bean how to process the data. In the example we only change the label of the button with the data.
4. JSFPushServlet calculates a ΔS after User Bean is done with processing the data, and transmits it back to Push Server
5. Push Server sends this ΔS. Backbase Client Runtime will process ΔS and update the DOM.

Figure 9.18a shows the initial state of the blank app with the integration. We again see the button "Backbase BV" above. Below, we see the same form as in Figure 9.17a, however, with one difference. Now the user has to specify the "pushDir". This is the servlet-mapping for the JSFPushServlet. Since now *jsfData* will be sent, JSFPush-Servlet has to be mapped somewhere else to process this data. Figure 9.17b shows the state of the application after data is published. We see that the button is changed to "hello world!", and ΔS (Server Delta) is displayed below.

9.4 Other Possible Use Cases Using JSF-Push Integration

Using JSF-Push integration, many other sample applications can be developed. Some examples are listed below:

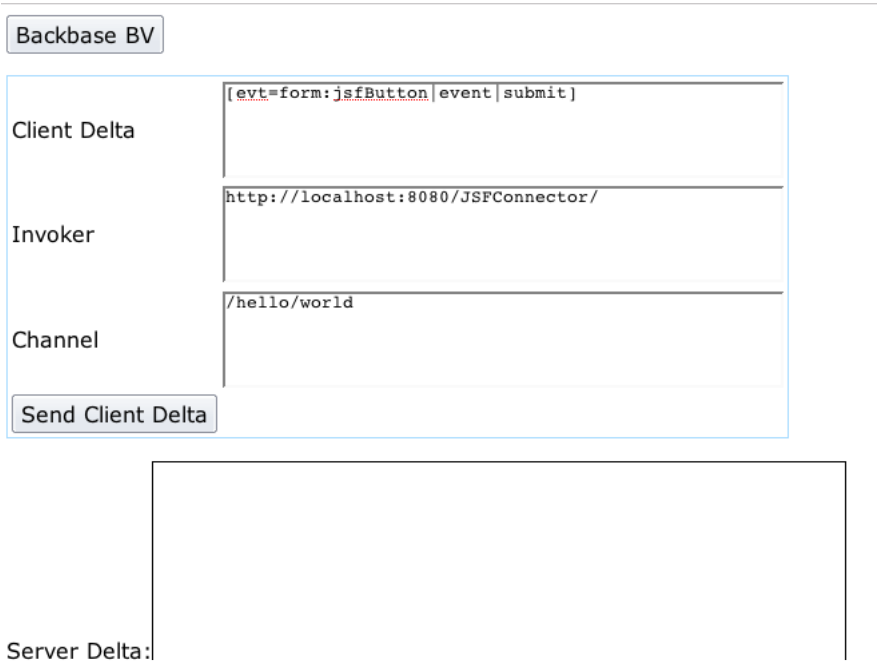
- Web presentations: A teacher can change the slide when she wants to go to the next one, and the current slide will be updated in all student users. Slides can be represented by AJAX widgets that have server components. When a slide is changed, all students will receive a ΔS, and their widgets will get updated with the new slide.

- Online collaboration applications: Applications such as Google Docs[1] might also use push support. Usually, in such complex applications server components are used, which are good use cases for JSF-Push integration.
- Any other application that uses server components: Many applications make use of server components. Introducing push support to such applications is very easy with this concept. Without any JavaScript programming on the client side, the existing widgets can be updated real-time by deploying the push server as a WAR file. This way existing applications can have push support and JSF elements such as *graphicImage* or *dataTable* can be updated real-time, without requiring an extra trip to the JSF server.

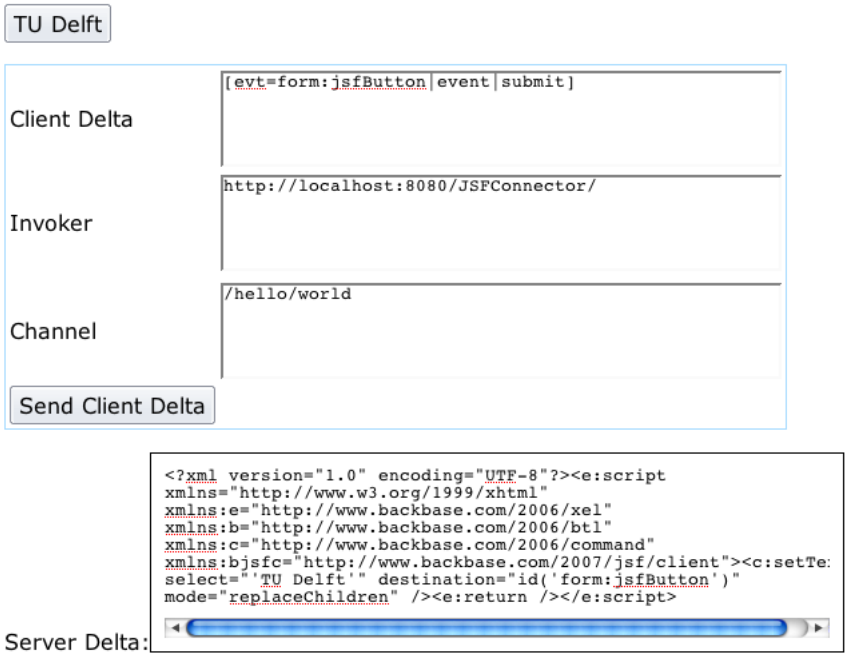
9.5 Conclusion

In this chapter we demonstrated several sample applications that represent certain use cases. We showed two examples that show client-to-client scenario, with subscribed publishers (chat) and nonsubscribed publishers (stock ticker). Our example showing full JSF-Push integration was rather basic, this is because we did not have enough time to develop a more advanced application. However, our examples demonstrate that a JSF-Push integration is indeed possible, and widgets of a user can be modified real-time by another user/service provider. This provides the opportunity to develop many different types of applications with server components support.

[1] `http://docs.google.com`

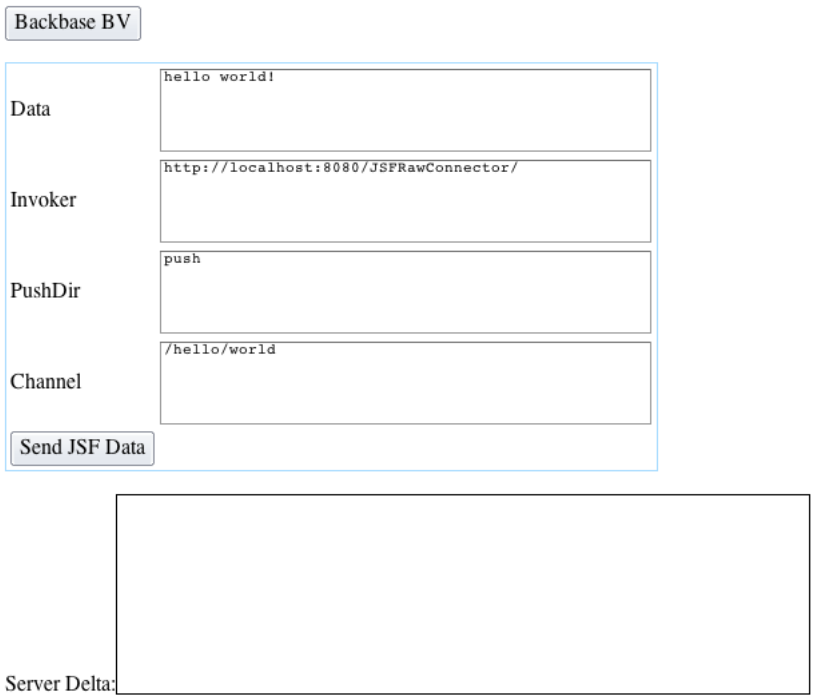


(a) Initial state



(b) After a ΔC update

Figure 9.17: The state of the blank app with JSF-Push integration using clientDelta



(a) Initial state

(b) After a raw data update

Figure 9.18: The state of the blank app with JSF-Push integration using jsfData

Chapter 10

Discussion

In the previous chapters we introduced the design and implementation of our COMET prototype. However, the prototype has still some limitations. As we have shown in Chapter 6, using a push application can cause scalability problems in certain application servers, pointing out a need for load balancing. In this chapter we first mention the limitations of our implementation, and later we discuss load balancing options.

10.1 Known Limitations

10.1.1 Limitations of the server

Our push server satisfies the requirements given in Section 4.2. However it needs the following additions:

- **Research for the scalability of HTTPConnector:** In order to integrate JSF and Push servers, we created an HTTPConnector class that uses HTTPClient library[1]. When Service Provider sends an update, Push Server has to make a JSF request for each subscriber using the HTTPClient library. According to Stevens [36], if an HTTP call is local, network layer of the operating system will detect this and it will transfer it to a memory call. This means that if Push Server and JSF Server are on the same machine, the call will be a local one, thus the solution will not cause huge network traffic. However, it still has an overhead and the solution is not tested with big number of users. We are dependent on the implementation of HTTPClient library and this might become a bottleneck. More research here is needed.

- **Optimization on the integration:** In our integration, JSF Server makes a call for each subscriber. However in certain cases, session scope is not needed. If an application scope is used, making a single call to JSF Server is enough. This optimization needs to be implemented before a stable release.

- **Full streaming support** Our push implementation is based on Jetty Cometd, therefore it only supports *long polling*. Full-streaming support should be added and its performance should be compared with long polling.

[1] `http://jakarta.apache.org/httpcomponents/httpclient-3.x/`

- **Adaptable pull/push:** As we have shown in Chapter 6, pull solution is more scalable, but its data coherency is low. Push on the other hand provides high data coherency, but is less scalable. Not all clients need high data coherency. The server should be able to switch those clients to "pull mode" under high load. An adaptable algorithm that uses both push and pull will optimize server performance.

- **Concurrent messages:** In certain use cases, only the latest push data matters. For example, in a Stock Ticker application, the client is only interested in the latest stock update. However, in our implementation, all updates are queued for all clients, and all are delivered upon a reconnect. For such scenarios, only the latest message should be saved and delivered.

- **Other push scenarios:** In Chapter 6, we performed an empirical study in order to see how the system behaves under high load. However, in this study we only considered one use case: 1 channel, a single publisher and many users. In this scenario, all subscribers need to be notified in case of every update. There are other use cases, such as multiple channels, subscribers divided unequally over these channel and multiple publishers. It is necessary to test these use cases as well.

- **Advices:** The optional *advice* field in BAYEUX protocol provides a way for the push server to inform its clients of its preferred mode of client operation so that in conjunction with server-enforced limits, BAYEUX implementations can prevent resource exhaustion and inelegant failure modes [35]. For example, if a server is under high load, it can tell some of its clients to connect to another server, or wait before reconnecting. Advice mechanism was still in draft and not yet supported when we began to extend Jetty Cometd. Such a feature can optimize server performance, along with adaptable pull/push approach.

10.1.2 Limitations of the Push Client library

As mentioned in Section 8.3, Backbase client team created a simple proof-of-concept JavaScript library that implements Bayeux stages in order to subscribe/publish XML data. The library performs well for satisfactory use, however it contains some limitations and needs to implement the following features:

- **JSF-Push Integration:** Backbase push client library only supports client-to-client push communication. We have extended this to support JSF-Push integration. However, the programmer still has to call the engine with the ΔS in order to update the DOM. These operations should be performed by the Backbase Client Runtime automatically.

- **Push Tag:** Currently, in order to establish a connection to Push Server, the application programmer has to call connect() function of the client library with the Push Server's URL. The client also has to process the incoming data. An elegant solution here is to create a custom JSF tag. JSF tags are the basic building blocks for JSF user interfaces and require no JavaScript knowledge [21]. See

Figure 10.1 how the user has to process incoming ΔS now. Figure 10.2 shows our tag suggestion.

```
 // handshake and connect to server
client.connect("http://localhost:8080/Push");
// subscribe to the channel
client.subscribe("/stocks/ibm/");
// process incoming push data and call the engine with the data
client.ondataavailable  = function(channel, data) {
        var dataString = ""+arguments[2].firstChild.data
        var xml = new DOMParser().parseFromString(dataString, 'text/xml');
        bb.construct(xml);
}
```

Figure 10.1: Adding Push support to a JSF application by JavaScript calls

```
<bjsf:push pushUrl="http://localhost:8080/Push"
channel="/stocks/ibm"/>
```

Figure 10.2: Adding Push support to a JSF application by JSF tags

- **Unimplemented features:** We extended Jetty Cometd with several features such as channel list, user list and join/leave notification (See Section 8.1.2). However, the client does not support them.

- **Multiple Batch Support:** Multiple push messages result in multiple trips to the server. The code in Figure 10.3 will result in 3 trips to the server. It should be possible to send multiple operations in 1 message.

```
client.publish("/stocks/ibm","5");
client.publish("/stocks/sun","6");
client.publish("/stocks/backbase","7");
```

Figure 10.3: Multiple sequential publish messages

- **Browser Support:** Currently the client library only supports Firefox 2.0 and Internet Explorer 7.0. It does not work with Safari and Opera, and it is not tested under Internet Explorer 6.0.

- **Proxy:** As a security precaution, XMLHttpRequest object does not allow requests to different hosts/ports. So, it is not possible to make a JSF request to *http://host1* and make push request to *http://host2*. If the push server and the JSF server are located on different hosts/ports, a proxy solution is needed to overcome this limitation.

10.2 Load Balancing

Load balancing is a technique (usually performed by load balancers) to spread work between many computers, processes, hard disks or other resources in order to get optimal resource utilization and decrease computing time. According to our results in Chapter 6, under a big number of COMET clients, a load balancing solution is needed. This is because the server will begin to saturate as the number of users increases.

Before deciding on a particular load-balancing solution, we need to introduce an important requirement: persistence [27]. Within the REST model [18], a request has no relationship with any other request from the Web site's perspective; that is, there is no need to maintain information, termed "state", on each user throughout his session. This stateless approach was suitable for Web sites that simply served up information in response to browser requests. But sophisticated web-based applications require the maintenance of information on a user and his session between individual user/server exchanges (transactions). This information (state) cannot be lost and is vital for proper functioning of such applications. We therefore define persistence as the continued existence of client state. Persistence is sometimes necessary in applications where client state is maintained on the server, and in other cases it can simply provide better performance [38] (as one server may have data cached related to a particular user while other servers may not).

Persistence is vital for push. Below we list and discuss available load balancing solutions and discuss how they can support persistence. They are taken from [27, 33, 38].

10.2.1 Round Robin

In this scheme, requests are sent to servers in a sequential and circular pattern: first message to server 1, second message to server 2, and so on. This scheme is suitable for traditional web applications, but it will cause problems for the push server, since it does not allow persistence by default. Push server keeps information about each user such as list of subscribed channels and messages. The user cannot be simply forwarded to a random server without sharing this data, otherwise she will be interrupted in the middle of a session. State must be stored somewhere (i.e., on a disk) and must be shared among all the servers. This will result in high availability, however also in high overhead. Storing session state in a database requires careful design consideration. For example, in one system, 70 percent of the total volume of SQL was related to handling state, and this was the area where the majority of the initial performance problems lay [27].

10.2.2 Sticky Sessions

In this scenario the load balancer (master) gets involved at the beginning of a session (e.g., user log-in) allocating a specified slave to a user. That slave is subsequently used for the life of the session and the master is bypassed. If the master fails, one of the slaves assume the master role. In some approaches, there is no master and packets are broadcasted to each of the target servers. There are several ways to achieve sticky sessions:

Source address switching

There are two types of source address switching [27, 33]:

- **Memory lookup:** The load balancer identifies the user by its IP address and creates a record in memory that records which server that remote IP address was sent to. It might employ sophisticated algorithms to assign a user to a specific server. Future requests from the same IP address will be sent to the same server. A timeout is usually specified so these entries do not need to be stored forever.
- **IP Address hashing**: This approach uses a hash function to assign an IP address to a specific server. This way users will be forwarded to a specific server in a deterministic way. This solution uses no memory, however it cannot use advanced algorithms to make an optimized user-server assignment.

The problem with these methods is that some users access the Internet through a cluster of proxy servers which means their request address may change from request to request. In corporate applications, many users might come from a few source addresses (i.e. internal NAT addresses). This way huge groups of users may come from one IP address and be sent to the same server, even if it is overloaded.

Cookie switching

The load balancer directs HTTP requests to a server group based on information embedded in a cookie in the HTTP header, which is known as session id [27, 33]. If SSL is used, the load balancer has no access to the cookie content. In such cases, SSL Session ID switching [6] can be used. This solution has the following limitations:

- **Cookie lifetime:** The load balancer has finite memory, so it might saturate, and the only solution to avoid this is to limit the cookie lifetime in the table. This implies that if a push user comes back after the cookie expiration, he will be directed to a different server. The new server will assign a new session id to the user, and the user will lose its previous messages.
- **Load balancer failure:** If the load balancer dies and its backup takes over, it will not know any association and will again direct users to a different server.

A workaround for these drawbacks is to use IP Address hashing. This way, should the cookie be lost on the load balancer, at least the users who have a fixed IP address will be kept on their server.

Cookie Insertion

In this method, the load balancer assigns a server to the user based on its cookie and inserts the address of the server into the user's machine as a new cookie [38]. This way, the user-server association is saved on the user side, instead of the server. This solves the problems mentioned in the previous subsection: cookie lifetime and load balancer failure. However, it requires more effort from the load balancer, which must open the stream to insert data to the user. This is not easily done, especially on hardware-based load balancers which have very limited knowledge of TCP and HTTP. It also requires

that the user agent accepts multiple cookies. This is not always the case on small mobile terminals for instance, which are sometimes limited to only one cookie.

URL switching

This load balancing solution is performed at layer 7 (Application). The requests are divided among servers based on the URL name or part of the name. In this scheme, users of the same application will be directed to the same server, which might cause in overloading [27, 38].

10.2.3 Other solutions

Other known load balancing solutions are as follows [27]:

- **Best Performing Server:** In this solution, the load balancer continually checks the performance of the servers, and assigns new clients to the best performing server.
- **Weighted Percentages:** With this approach, a bias can be given to specified servers so that they receive more or less traffic. This is mostly used where the target servers have different hardware specifications; for example one server may be twice as fast as the others and it should therefore be weighted to receive twice the traffic
- **Least Connections:** In this scheme, the server with the least number of connections processes the request.

All these solutions, just like Round Robin, require state synchronization among servers, which will have a negative effect on performance.

10.2.4 Applications in the industry

Load balancers in the software industry usually combine couple of mentioned solutions above:

- **Tomcat 5.0 :** Tomcat 5.0.15[2] and later ships with a simple load balancer. It is a Java based extensible solution. By default, it comes only with *URL switching*.
- **Apache 2 mod_proxy_balancer:** Apache's mod_proxy_balancer module provides load balancing support for HTTP, FTP and AJP13[3] protocols. It supports *sticky sessions*, *weighted percentages*, and *round robin* .
- **JBoss:** JBoss Application server[4] supports *cookie switching* and *session sharing*. It can replicate session bean state among servers. It also provides JMS communication between the nodes.

[2] http://tomcat.apache.org/tomcat-5.0-doc/balancer-howto.html
[3] http://tomcat.apache.org/connectors-doc-archive/jk2/common/AJPv13.html
[4] http://labs.jboss.com/jbossclustering/

- **Open Terracotta:** Open Terracotta[5] is an open-source, JVM-level clustering solution. By using JVM-level clustering, it enables applications to be deployed on multiple JVM's, yet interact with each other as if they were running on the same JVM. Terracotta extends the Java Memory Model of a single JVM to include a cluster of virtual machines such that threads on one virtual machine can interact with threads on another virtual machine as if they were all on the same virtual machine with an unlimited amount of heap. The programming model of applications clustered using Open Terracotta is the same or similar to that of an application written for a single application. When clustered by Terracotta, web application session becomes highly available without being replicated to every app server in the cluster. This allows persistence without session sharing by databases, but also high availability, since all servers can process the user's request. Jetty and Terracotta developers are currently collaborating in order to find a clustering solution for COMET[6]

10.3 Conclusion

Even though our prototype fulfills the requirements and displays an integration of JSF and COMET, some limitations such as scalability and concurrent messages still need to be addressed before a possible release. Full streaming support and adaptable pull/push are also features that will improve the overall performance. Load-balancing is also an option in order to deal with scalability. Several load-balancing solutions are available, however each comes with a limitation, and these should be considered before making a decision.

[5] `www.terracotta.org`

[6] `http://docs.codehaus.org/display/JETTY/Session+Clustering+with+Terracotta`

Chapter 11

Conclusion and Future Work

There are many use cases where it is important to update the client-side interface in response to server-side changes. AJAX itself does not address stock tickers, chatrooms or auction sites. Libraries such as Cometd or DWR bring real-time event notification support to existing AJAX applications, without the need of downloading any plugins, unlike a TCP socket based solution, such as Flash sockets or Java Applets. However, there has been no empirical study conducted to find out the actual trade-offs of using these libraries on browser-based or AJAX applications and comparing it with a pull approach. The integration of the libraries with existing server components (such as JSF components) is also not trivial, and requires further study.

In this thesis we have compared pull and push solutions for achieving web-based real time event notification. Our experiment shows that if we want high data coherence and high network performance, we should choose the push approach. However, push brings some scalability issues; in our test server, the application's CPU usage is 7 times higher as in pull. According to our results, the server starts to saturate at 350-500 users. For larger number of users, load balancing and server clustering techniques are unavoidable.

With the pull approach, achieving total data coherence with high network performance is very difficult. If the pull interval is higher than the publish interval, some data miss will occur. If it is lower, network performance will suffer. Pull performs well only if the pull interval equals to publish interval. However, in order to achieve that, we need to know the exact publish interval beforehand. However, the publish interval is rarely static and predictable. This makes pull useful only in situations where the data is published frequently according to some pattern.

These results allow engineers to make rational decisions concerning key parameters such as pull and push intervals, in relation to, e.g., the anticipated number of clients. Furthermore, the experimental design allows them to repeat similar measurements for their own (existing or to be developed) applications on different application servers.

In this thesis, we also showed how push libraries Cometd and DWR can be integrated with an AJAX JSF framework. We provided several extensions on Cometd that will allow a full integration between the JSF Application and Cometd. We demonstrated our integration for these two libraries by providing several examples. Even though our examples were rather basic, they prove that a JSF-Push integration al-

lows current JSF applications to have push support without requiring any JavaScript knowledge from the developer. By using the integration the push connection will be transparent to the user and the Push Server can serve multiple JSF apps, running on different app servers.

The contributions of this thesis include the experimental design, a reusable implementation of a sample application in push and pull style as well as a measurement framework, and the experimental results. A prototype COMET server with Cometd extensions (including HTTPConnector and several extra features), JSF extensions (JSF-PushServlet and NotifyPushData) and Push client extensions form our other contributions. Our prototype fulfills the requirements of Backbase, which were mentioned in Section 4.2.

Our future work includes the following:

- As we have mentioned in Chapter 10, the prototype contains several limitations that need to be addressed. These include, among others, a scalability research for the HTTPConnector, optimization on the JSF-Cometd integration and the client library.
- A hybrid approach that combines pull and push techniques for AJAX applications will increase performance and have a positive effect on scalability, since it will combine the benefits of both approaches.
- We intend to extend our testing experiments to different versions of Jetty and alternative push server implementations, for example ones that are based on holding a permanent connection (e.g., Lightstreamer[1] or DWR) as opposed to the long polling approach which was used during the tests. Additional experiments with a variety of pull intervals and a multiple number of channels are also desired.

[1] `http://www.lightstreamer.com`

Bibliography

[1] *Backbase JSF Edition 4.0.1 Application Development Guide*. Backbase B.V., 2007.

[2] Swarup Acharya, Michael Franklin, and Stanley Zdonik. Balancing push and pull for data broadcast. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 183–194. ACM Press, 1997.

[3] M. Ammar, K. Almeroth, R. Clark, and Z. Fei. Multicast delivery of web pages or how to make web servers pushy. Workshop on Internet Server Performance, 1998.

[4] Jean-Francois Arcand. Grizzly: An HTTP listener using java technology nio. `http://weblogs.java.net/blog/jfarcand/archive/2005/06/index.html`, 2005.

[5] Backbase enterprise Ajax framework. `http://www.backbase.com/`, 2007.

[6] BEA. Using load balancers and web proxy servers - SSL persistence. `http://edocs.bea.com/platform/docs81/deploy/loadbal.html`, 2007.

[7] T. Bernes-Lee, L. Masinter, and M. McCahill. Uniform resource locators (URL). Internet Request for Comment RFC 1738, Internet Engineering Task Force, December 1994.

[8] Manish Bhide, Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant Shenoy. Adaptive push-pull: Disseminating dynamic web data. *IEEE Trans. Comput.*, 51(6):652–668, 2002.

[9] E. Bozdag, A. Mesbah, and A. van Deursen. A comparison of push and pull techniques for Ajax. In Shihong Uang and Massimilano Di Penta, editors, *Proceedings of the 9th IEEE International Symposium on Web Site Evolution (WSE)*, pages 15–22. IEEE Computer Society, 2007.

[10] Engin Bozdag. Push solutions for Ajax technology - research report. Master's thesis, Delft University of Technology and Backbase, 2007.

[11] Brian Chess, Yekaterina Tsipenyuk O'Neil, and Jacob West. Javascript hijacking. www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf, 2007. Fortify Software.

[12] Mortbay Consulting. Jetty 6 architecture. http://docs.codehaus.org/display/JETTY/Architecture, 2006.

[13] Mortbay Consulting. Jetty webserver documentation - continuations. http://docs.codehaus.org/display/JETTY/Continuations, 2006.

[14] Direct Web Remoting. Reverse Ajax documentation. http://getahead.org/dwr/reverse-ajax/configuration, 2007.

[15] DWR discussion forum. DWR in a clustered environment: Clusterscriptsessionmanager? http://www.nabble.com/dwr-in-a-clustered-environment:-ClusterScriptSessionManager--t2582194.html, 2007.

[16] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[17] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2068, Internet Engineering Task Force, January 1997.

[18] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Inter. Tech.*, 2(2):115–150, 2002.

[19] Michael Franklin and Stan Zdonik. Data in your face: push technology in perspective. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 516–519. ACM Press, 1998.

[20] Jesse Garrett. Ajax: A new approach to web applications. Adaptive Path, 2005.

[21] David M. Geary and Cay S. Horstmann. *Core JavaServer Faces*. Prentice Hall PTR, 2004.

[22] Manfred Hauswirth and Mehdi Jazayeri. A component and communication model for push systems. In Oscar Nierstrasz and Michel Lemoine, editors, *ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 20–38. Springer-Verlag / ACM Press, 1999.

[23] IBM websphere application server. http://www.ibm.com/software/webservers/appserv/was/, 2007.

[24] Kanaka Juvva and Raj Rajkumar. A real-time push-pull communications model for distributed real-time and multimedia systems. Technical Report CMU-CS-99-107, School of Computer Science, Carnegie Mellon University, January 1999.

[25] Rohit Khare. Beyond Ajax: Accelerating web applications with real-time event notification. http://www.knownow.com/products/docs/whitepapers/KN-Beyond-AJAX.pdf, 2005.

[26] Rohit Khare and Richard N. Taylor. Extending the representational state transfer (rest) architectural style for decentralized systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 428–437. IEEE Computer Society, 2004.

[27] Brian King. *Performance Assurance for IT Systems*. Auerbach Publications, Boston, MA, USA, 2004.

[28] Jean-Philippe Martin-Flatin. The push model in web-based network management. `http://www.labunix.uqam.ca/~jpmf/papers/index.html`.

[29] A. Mesbah and A. van Deursen. An architectural style for Ajax. In *Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*, pages 44–53. IEEE Computer Society, 2007.

[30] Ali Mesbah, Kees Broenink, and Arie van Deursen. Spiar: An architectural style for single page internet applications. Technical report, Centrum voor Wiskunde en Informatica (CWI), 2006.

[31] Ali Mesbah and Arie van Deursen. An architectural style for Ajax. In *WICSA '07: Proceedings of the 6th Working IEEE/IFIP Conference on Software Architecture*, pages 44–53. IEEE Computer Society, 2007.

[32] Netscape. An exploration of dynamic documents. `http://wp.netscape.com/assist/net_sites/pushpull.html`, 1996.

[33] Foundry Networks White Paper. Server load balancing in today's web-enabled enterprises. `http://www.foundrynet.com/pdf/wp-server-load-bal-web-enterprise.pdf`, 2002.

[34] Linda Dailey Paulson. Building rich web applications with Ajax. *Computer*, 38(10):14–17, 2005.

[35] Alex Russell, Greg Wilkins, and David Davis. Bayeux - a json protocol for publish/subscribe event delivery protocol 0.1draft3. `http://cometd.com/bayeux/`, 2007.

[36] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols, Chapter 2.7*. Addison-Wesley Longman Publishing Co., Inc., 1993.

[37] Sun microsystems - remote method invocation home. `http://java.sun.com/javase/technologies/core/basic/rmi/`.

[38] Willy Tarreau. Making applications scalable with load balancing. `http://1wt.eu/articles/2006_lb/`, November 2006.

[39] Alex Russell Dojo Toolkit. Comet: Low latency data for the browser. `http://alex.dojotoolkit.org/?p=545`.

[40] Vittorio Trecordi and Giacomo Verticale. An architecture for effective push/pull web surfing. In *2000 IEEE International Conference on Communications*, volume 2, pages 1159–1163, 2000.

[41] W3C. Chunked transfer coding. http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html#sec3.6.1.

[42] W3C. The xmlhttprequest object. www.w3.org/TR/XMLHttpRequest, 2006.

[43] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, 35(5):230–243, 2001.

[44] Matt Welsh and David E. Culler. Adaptive overload control for busy internet servers. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

AJAX: Asynchronous Javascript And XML

BCR: Backbase Client Runtime

BTL: Backbase Tag Library

CSS: Cascading Style Sheets

DHTML: Dynamic Hyper Text Markup Language

DOM: Document Object Model

DWR: Direct Web Remoting

FSM: Finite State Machine

JEE: Java Enterprise Edition

JRE: Java Runtime Environment

JSF: Java Server Faces

JVM: Java Virtual Machine

MPI: Multi Page Interface

NIO: (Java) New Input/Output

OS: Open Source

PaP: Push and Pull

PoP: Push or Pull

REST: Representational State Transfer

RIA: Rich Internet Applications

RMI: Remote Method Invocation

SEDA: Staged Event Driven Architecture

SMIL: Synchronized Multimedia Integration Language

SP: Service Provider

SPI: Single Page Interface

TDL: Backbase Tag Definition Language

TTR: Time to Refresh

URL: Uniform Resource Locator

WAR: Web Application Archive

W3C: The World Wide Web Consortium

XEL: Backbase XML Execution Language

Appendix B

Backbase Push Full Package Listing

Push server consists of the following packages (See also Figure B.1):

- *com.backbase.push*: contains the core classes such as Bayeux, Channel, CometdServlet, etc.
- *com.backbase.push.connectors* contains Connector interface and HTTPConnector class, which are needed for JSF - Cometd integration
- *com.backbase.push.filter*: contains classes for data filtering
- *com.backbase.push.handlers*: contains different handlers responsible of several Bayeux stages (See Section 5.1.1)
- *com.backbase.push.utils*: contains XMLHelper utility, which is responsible of parsing XML and generating XML based push messages.

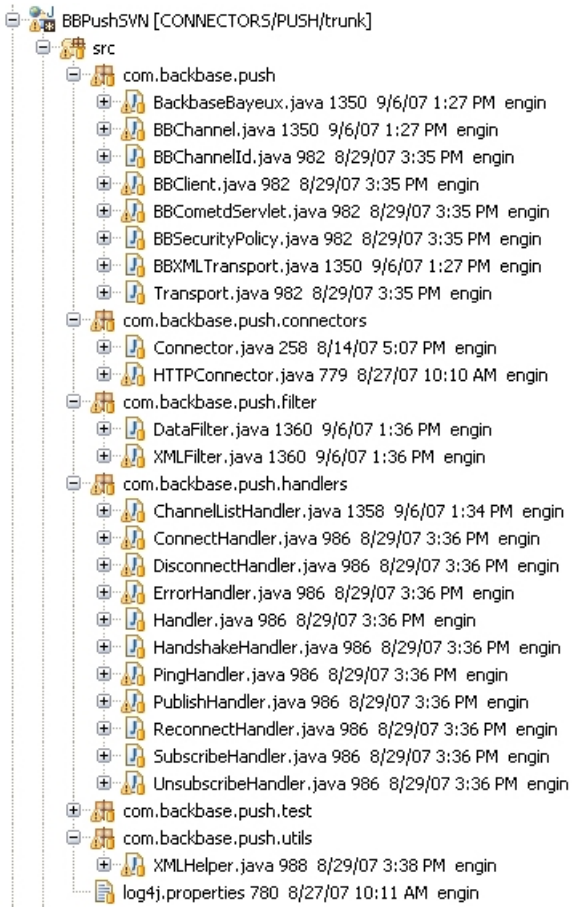


Figure B.1: Backbase Bayeux Push Package