FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Comet.NET, Design and Implementation of a Bayeux Server for the .NET platform

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Boris Mesetovic
Matrikelnummer 0225445

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Prof. Dr. Schahram Dustdar
Mitwirkung: Univ.-Ass. Dipl-Ing Dipl-Ing Johann Oberleitner

Wien, 04.06.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Abstract

Modern Web applications increasingly rely on real-time server-initiated delivery of data to browsers. It increases the responsiveness of the application, improves the overall user experience and allows various real-time collaboration scenarios between users. This application model, in which the server delivers data to a browser via HTTP without the browser explicitly requesting it, is called Comet. Protocols and techniques defined by the Comet application model are essential for Web applications that need to asynchronously deliver events to clients.

One of the recent developments in the domain of Comet Web applications is an open-source application protocol called Bayeux. It provides means of two-way and low-latency communication between the server and the client and is typically used to deliver notifications directly to Web browsers as soon as they occur.

In this thesis, we present a native .NET implementation of a Bayeux server called Comet.NET. Comet.NET is a stand-alone, high-performance Bayeux server with support for both streaming and polling communication techniques. It offers synchronous and asynchronous application programming interfaces, is very scalable and robust and is designed to be easily embeddable in any .NET application.

Features of Comet.NET are demonstrated by presenting Teletrader HTTP Push Service, a fully functional enterprise stock market ticker application built on top of it. Furthermore, the thesis provides a detailed evaluation of the performance and scalability of the presented solution and discusses how it compares with the reference Bayeux server implementation.

Kurzfassung

Moderne Web Applikationen basieren zunehmend darauf, dass Daten in Echtzeit vom Server direkt an den Browser ausgeliefert werden. Dieser Ansatz erhöht die Reaktionsfähigkeit und die allgemeine Usability der Applikation und ermöglicht verschiedene Szenarien für eine Real-Time Kollaboration zwischen Benutzer. Das Applikationsmodell, mit dem der Server die Daten an Browser über das HTTP Protokoll ausliefert, ohne dass sie explizit seitens Client angefordert werden, heisst Comet. Die im Comet Applikationsmodell definierten Technologien und Protokolle sind für die Web Applikationen, die Daten asynchron an Clients ausliefern möchten, von wesentlicher Bedeutung.

Eine der neuen Entwicklungen im Bereich von Comet Web Applikationen ist das Open-Source Protokoll namens Bayeux. Das Protokoll ermöglicht bidirektionale Kommunikation zwischen dem Server und dem Client und wird typischerweise dazu verwendet, Benachrichtigungen mit geringer Latenz direkt an den Browser auszuliefern.

In der vorliegenden Master-Arbeit präsentieren wir Comet.NET, eine auf .NET basierende Implementierung von Bayeux-Server. Comet.NET ist ein stand-alone, hoch performanter Server, der sowohl Streaming als auch Polling unterstütz. Der Server bietet eine synchrone und eine asynchrone API, ist sehr skalierbar und robust und wurde so designed, damit er einfach in beliebige .NET Applikation eingebettet werden kann.

Die Eigenschaften von Comet.NET werden anhand von Teletrader HTTP Push Service demonstriert. Es handelt sich dabei um eine Applikation für die Auslieferung von Marktdaten im Web. Am Schluss werden die Performance und Skalierbarkeit von Comet.NET evaluiert und mit Performance und Skalierbarkeit der Bayeux Referenz-Implementierung vergliechen.

Acknowledgments

I would like to thank my parents, friends and family, whose support, encouragement and love has been a great inspiration in all my academic and professional endeavors.

In addition, special thanks are due to Johann Oberleitner for his support, creative insights and comments for the whole duration of this project.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Initially, the Web was an information system consisting mainly of static textual and graphical content that was virtually connected via hyperlinks. Over the time, the Web slowly evolved from this rather static global information system into what it is today: a very dynamic global distributed application platform.

Advantages of Web applications over desktop applications are numerous, for application developers and application users alike[1, 2, 3]. A Web application can be used anywhere; the only requirement is that the device has an Internet connection and a Web browser. There is no need to download and install the application on a computer, which makes the entry barrier for the users very low. Web applications are located on servers and have typically no client-side parts, so adding new features and fixing bugs is simplified. From the user's perspective, software updates are fully automated and are available next time the user accesses the application. From the developer's perspective, maintenance is a lot easier, because only the central server application needs to be updated. Besides that, all users always use the same application version, so the overhead for supporting multiple application versions is non-existent. The fact that the application and data is located on the server makes another important aspect of Web applications possible: they can be easily built as multiuser applications that allow data sharing and collaboration between users.

A prime example of a modern Web application is Facebook[4], a social networking service with more than 600 million users[5]. Facebook is a highly dynamic Web application that allows users to share personal information, interests, photos, status updates and communicate with friends in real-time. Other prominent examples of dynamic real-time social networking services include Twitter [6], FriendFeed[7] and reddit[8].

Web applications such as Youtube[9] and Hulu[10] allow users to stream various types of multimedia, upload their multimedia content, collaborate with others and participate actively in the community. Flickr[11] is another example of a social network that allows sharing of photos and collaboration between users.

Google Docs[12], a Web-based office suite offered by Google, includes a word processor, a spreadsheet and a presentation application. One of the main features of Google Docs is that

it allows users to create and edit rich documents directly in a Web browser while collaborating with other users in real-time.

Another example of highly dynamic Web applications with real-time requirements are financial services. These applications aim to efficiently distribute market data in the Web in real-time and offer the same level of service as traditional desktop applications. A prime example of this type of applications is Thompson Reuters Eikon[13], which brings the financial terminal into the Web environment and enhances it with social collaboration features.

Even though Web applications have many advantages over desktop applications, the underlying technologies and architecture continue to prevent the full exploitation of the possibilities of the Web. Some of the core protocols and technologies used in the Web were not originally designed for use cases that are prevalent today. One example for a technology that acts as a limiting factor in the Web today is the Hyper Text Transfer protocol (HTTP)[48].

HTTP is the communication protocol used in the Web. It is a stateless protocol that implements a request/response communication paradigm. Data exchange is always initiated by the client. This poses a significant technological challenge for Web applications that have use cases where a server should be able to send data to client browsers as reaction to certain events, without waiting for clients to explicitly request it. Server-initiated delivery of data is an important aspect of all Web applications mentioned above. It increases the responsiveness of the application, improves the overall user experience and allows various real-time collaboration scenarios between users. Some applications, such as market data delivery, are only possible with server-side push of data.

This application model, in which the server delivers data to a browser via HTTP without the browser explicitly requesting it, is called Comet[40] or Reverse Ajax[41]. Protocols and techniques defined by the Comet application model are essential for Web applications that need to asynchronously deliver events to clients. For a detailed description of the application model and its technological foundations, please refer to Section 2.1.

One of the recent developments in the domain of Comet Web applications is an open-source application protocol called Bayeux[50]. This is a protocol for transporting asynchronous messages in the Web with the main goal of overcoming the traditional, client/server nature of HTTP. It provides means of two-way and low-latency communication between the server and the client and is typically used to deliver notifications directly to Web browsers as soon as they occur.

Over the past two years, several implementations of the Bayeux protocol have emerged. Aside from the reference implementation - Jetty CometD[14] - there are several other quite mature Java-based implementations. IBM has incorporated an implementation of the protocol into WebSphere as part of Feature Pack for Web 2.0[15]. Oracle WebLogic application server also supports the Bayeux protocol[16]. The Atmosphere Framework[59], a portable Java Comet framework, ships with a module that provides support for Bayeux.

While the protocol has a widespread support in the Java world, the support in the .NET world is rather limited. Oyatel's CometD .NET[19] is an attempt to port the core of Jetty CometD to .NET, but the project was started just recently (January 2011) and is not usable. At the time of writing, the only known implementation of the Bayeux protocol in .NET is WebSync[20], a closed-source and on-demand solution built on top of Microsoft Internet Information Server (IIS). As far as we know, there is no publicly available implementation of the Bayeux protocol

in .NET that can be used in a stand-alone mode, without relying on Microsoft IIS.

A native .NET implementation of a Bayeux server has several advantages over Java implementations. The most important one is that the server can leverage the .NET technology stack which offers some technologies not available in Java. An example is the Windows HTTP Server API[21]: a high-performance Windows kernel-level library that handles HTTP traffic. Although it is a low-level library that exposes only a C interface, .NET applications can use its functionality directly using wrapper classes that are part of the .NET framework. There is currently no Java wrapper for the Windows HTTP Server API. A native .NET Bayeux server can easily provide hooks for Windows Communication Foundation (WCF) too and expose its functionality to consumers in a very flexible way.

The lack of support for the Bayeux protocol on the .NET platform also poses a significant technical problem for companies that want to expose their .NET-based backend data stores on the Web using the protocol. While this technological gap between the platforms could be solved, for example by implementing proxy/translator services on both sides, it induces a lot of overhead for implementation, maintanence and system operations. A much better solution is to avoid the gap altogether and implement a native .NET Bayeux server. The server fits much better into a .NET-based eco system and makes the integration with other backend services and applications easier, less error-prone and more natural.

TeleTrader Software AG is a company with a backend system that is primarily based on the .NET framework. It is a Vienna-based IT service provider specializing in stock market data. The company has a strong focus on the Web and offers a complete technology stack for integration of market data into Web applications. An important part of that stack is a framework for real-time delivery of market data directly to Web browsers. The existing solution is based on deprecated technologies and shall be replaced by a solution based on the Bayeux implementation resulting from this master's thesis. The solution will also be presented in this thesis, as a sample application that demonstrates features of the Bayeux server.

1.1 Goals

After presenting the problem background and the motivation for the solution, we can formulate the goals of the master's thesis.

The main goal of this thesis is: *design and implementation of a stand-alone, high-performance Bayeux server based on Microsoft .NET framework that supports both streaming and polling communication techniques*. The server will be designed as a class library that can be used by any .NET application that wishes to provide Comet functionality and deliver data in real-time to Web clients. This library is called *Comet.NET*.

The second goal of the thesis is: *performance evaluation of Comet.NET and comparison to the Jetty CometD reference implementation*. Performance and scalability are considered to be very important aspects of the presented solution. Once the solution is implemented, its performance and scalability will be evaluated in a series of load tests.

1.2 Scope

This thesis addresses the problem of server-side push in the Web and provides a .NET-based solution that allows building of efficient and scalable Comet Web applications. The thesis focuses on the following three topics:

- Functional and non-functional requirements, design and implementation details of Comet.NET. We do not only provide a description of the architecture and implementation, but also describe architectural constraints and give insight into reasons and rationals behind architectural and implementational decisions.
- Teletrader HTTP Push Service. The enterprise stock market ticker application is used as a show case that demonstrates how to build Comet applications based on Comet.NET.
- Performance. For purposes of performance testing, a series of repeatable benchmarks is defined. We present the general setup for performance testing, benchmark details and their results.

1.3 Organization

The remainder of this thesis is structured as follows:

- Chapter 2 discusses the core technologies and protocols which form the technical basis of the solution presented in the thesis. It gives an overview of the Comet application model, provides an in-depth explanation of the Bayeux protocol and briefly presents the JSON format.
- Chapter 3 describes the solution presented in the thesis. The chapter starts with the functional and non-functional requirements for Comet.NET. The design and architecture of the solution are discussed next. The chapter also provides insight into various relevant implementation details. The goal is to not only describe the requirements and the architecture, but also to provide reasons and rationals behind particular functional and architectural decisions.
- Chapter 4 presents two sample applications based on Comet.NET. The first sample application is a Chat application that serves as a simple example of a real-time interaction via Comet.NET. The second sample application is the TeleTrader HTTP Push Service, an enterprise stock market ticker application built on top of the Comet.NET library.
- Chapter 5 discusses the actual performance and scalability of the solution presented in the thesis. In addition to that, the chapter provides a performance comparison of the presented implementation with the reference Bayeux server implementation.
- Chapter 6 presents three existing open-source solutions that are based on the Bayeux protocol and provides relevant implementational details. The presented solutions are Jetty and its CometD[14] module, Grizzly[58] and its Comet module and the Atmosphere Framework[59].

- Chapter 7 presents the related work in the field of push-based systems in general and the solutions for push-based delivery of data to Web browsers in particular.
- Chapter 8 concludes the thesis with a short summary and plans for improving the solution and work to be carried out in the future.

CHAPTER 2

Technologies

In this chapter we discuss the core technologies and protocols which form the technical basis of this thesis. The first section gives an overview of the Comet application model. The second section provides an in-depth explanation of the Bayeux protocol. The chapter is closed with a brief presentation of the JSON message format.

2.1 Comet

Comet is a Web application model in which a server delivers data to a browser without the browser explicitly requesting it. Comet is an umbrella term for various techniques that try to reduce delays and deliver data asynchronously from a Web server to Web browsers. Unlike Ajax[37] and traditional Web applications, Comet applications do not pull data from a server periodically. Instead the server has an open line of communication with which it can push data to the client. The communication between the server and the client relies only on features included in browsers by default, most notably on Javascript, rather than on 3rd party plugins. As such, Comet is essential for real-time event-driven Web 2.0 applications that generate a lot of server-side events that have to be pushed frequently to clients without the dependency on technologies such as Java Applets[38] or Adobe Flash[39].

The term was first coined by A. Russel, lead developer at Dojo Foundation, in his article called “Comet: Low Latency Data for the Browser”[40]. The Dojo Foundation is the co-initiator of the CometD project which later evolved into the Bayeux protocol specification[50]. Different solutions to the problem of low-latency delivery of data to Web browsers have existed prior the definition of the term Comet, but it was hard to communicate about them. There was no clear name developers and other people involved could associate existing solutions with. Giving these approaches a digestible, sounding name and a compact description made discussion and collaboration on topics of real-time data delivery in the Web easier. It also helped overcome the complete lack of standardization via W3C and start pushing these data delivery approaches into the mainstream Web development.

It is worth noting that some authors use the term Comet interchangeably with Ajax Push and Reverse Ajax[41]. All of these terms refer to essentially the same, but are far less common.

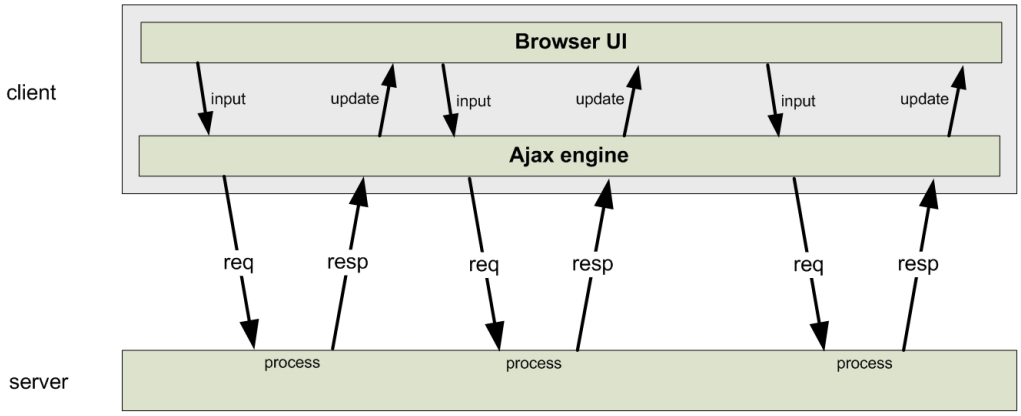

Figure 2.1: Ajax communication style

Available research papers mainly focus on comparing existing Comet solutions or implementation of new ones, but none discusses the actual term, what it accompanies and how it relates to other existing and future technologies. Russel's original article delivers a sufficient description of what Comet is and what its main goals are, but it leaves a lot of room for interpretations. For example, the relationship of Ajax and Comet is not completely clear: is Comet a subset of Ajax or just a similar architectural style with common technologies, but different approaches? It is also undefined if Comet should only be based on Web standards like HTTP and Javascript. There has been some discussion on these topics in the Comet community[42], but there are still no satisfactory and widely accepted answers. Lack of standardization structures and an almost non-existent formalization process make the standardization of Comet an almost impossible task. This technical ambiguity will probably never be completely solved, just like Ajax still leaves room for different interpretations even though it has reached a point where it can be considered a state-of-the-art approach in building Web applications.

Sometimes Comet is advocated as real-time Web, regardless of the technique used to achieve it. The focus is on the objective and on the result visible to the user. What matters is *what* is achieved, not how. Applications that update themselves without user interaction when server-side changes occur are considered Comet applications. The only requirement is that the update is not triggered by the user and that only modified parts of the page are changed. According to this definition, Comet equals to any kind of periodic refresh of a Web page.

This thesis uses a more common and more strict definition of Comet as facility for server side push of data: only Web applications in which a server asynchronously initiates sending of data to a client, as this becomes available, are actually Comet applications. Simple background refresh of a Web page is not sufficient: the update needs to be implemented in a certain way in order to be qualified as Comet application. The focus here is on *how* event data is delivered to clients and *who* initiates the delivery, and not only on *what* effect is achieved from a user

perspective.

The two most widely accepted Comet techniques are long polling and streaming. They are described in the following sections.

2.1.1 Long polling

The *long polling* Comet communication technique is a mixture of server-side push and client-side pull approaches.

When using the conventional, periodic polling, a client sends requests in a predefined interval and receives an immediate response. If there is new data since the last request, it is sent as a part of the response. If no new data is waiting to be delivered to a client, an empty response is sent. This communication style is typically used in Ajax applications and is depicted in Figure 2.1.

There are numerous problems with this approach, but the two major are: high average latency and high server load. Long polling introduces a very important optimization: no response is returned to the client until data is available or a timeout occurs. As soon as a response is received by the client, either due to the fact that events have occurred or a request timed out, the client sends a new request. The result is a significant reduction in latency because the server usually has a request pending when it is ready to return information to the client. A graphical representation of the long polling interaction between a client and a Comet server is shown in Figure 2.2.

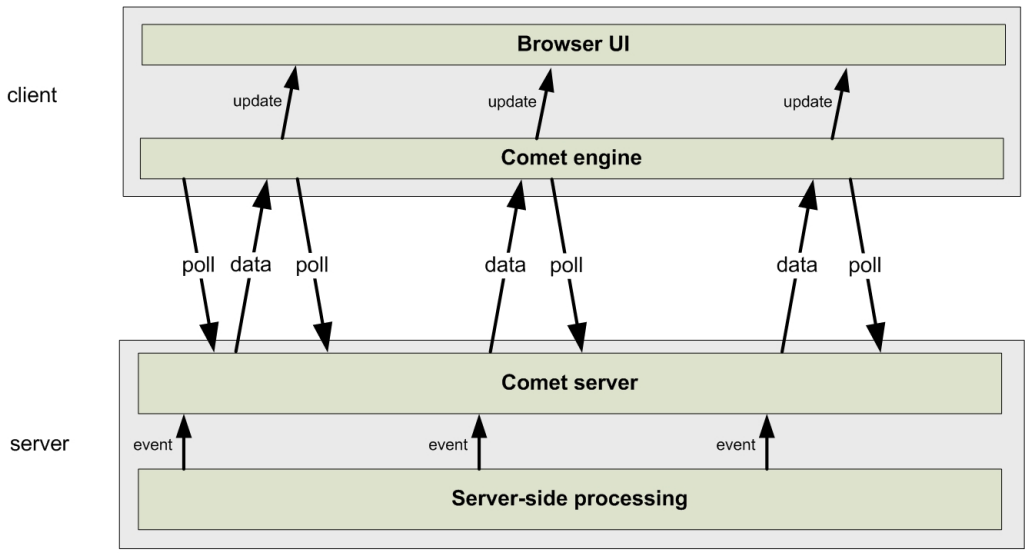

Figure 2.2: Comet communication style: long polling

One of the main characteristics of long polling is that a full HTTP request/response cycle is always executed, regardless of whether data was available or a timeout occurred. However, this does not necessarily mean that the underlying TCP/IP connection needs to be terminated as

well. Most modern implementations of long polling reduce costs of TCP/IP setup and teardown to a minimum by reusing the same connection for multiple request/response cycles. Without the overhead for setup and teardown of a connection, which is the primary cause of delays, clients are able to eliminate latency almost completely. The concept of persistent connections is defined in HTTP/1.1 and is supported by all major browsers and Web servers.

The only actual latency issue with long polling is related to the round trip time between browser and server. Because of the nature of HTTP, a full request/response cycle is needed for any kind of data exchange, so each event sent using long polling has roughly double latency compared to streaming (described in Section 2.1.2). Even though it induces some additional overhead compared to pure streaming, it has two important benefits: standard compliance and network resilience to a certain level.

Since long polling uses a request/response pair for each exchange of data, it is fully compliant to the HTTP communication model. And because it works both in keep-alive and older HTTP/1.0 interaction scenarios, it is resilient to occasional interruptions in TCP/IP connections. Applications based on Comet implementations that work with long polling do not need to know if the TCP/IP connection is interrupted, as long as requests come in - possibly from a re-opened connection.

A typical use case for long polling is Web chat[43][44]. Users that are viewing the same Website at the same time can exchange messages just by typing them into a field directly in the browser. Messages are then delivered to other users via a Comet server. This scenario is often seen on social networking sites or sites that offer live technical support. Even if group chat is enabled, the number of users and messages that they send to each other is usually not very high. The frequency of the messages is usually quite low and does not exceed several messages per second. In most cases, messages do not contain critical data and latency induced by the request/response cycle of long polling is permissible.

2.1.2 Streaming

Streaming via HTTP is an old technique first introduced in 1992 by Netscape under the name "dynamic document" [45]. Their flagship product and most popular browser at the time, Navigator 1.1, introduced the ability to perform a server push via an HTTP content type of *multipart/x-mixed-replace*[46]. A long running process, usually a CGI process, would maintain an open TCP connection to the browser and send content at arbitrary intervals, without client requests. The technique was ahead of its time and was rarely used. Support for this content type was never added to Internet Explorer, so when Netscape lost the browser war, HTTP streaming with multipart/x-mixed-replace was mostly abandoned. This old idea was revived in Comet applications, but its implementation differs slightly.

The basic form of HTTP streaming is *forever-response*. It is a simple and straightforward streaming via HTTP by keeping the response open indefinitely and sending data as soon as it is available. The client sends a request, to which the server sends an immediate response which is never terminated. The server uses chunked transfer encoding and persistent connections to deliver events to the client as they occur. Chunked transfer encoding is a data transfer mechanism in HTTP/1.1 that allows data to be reliably delivered between a server and a client without

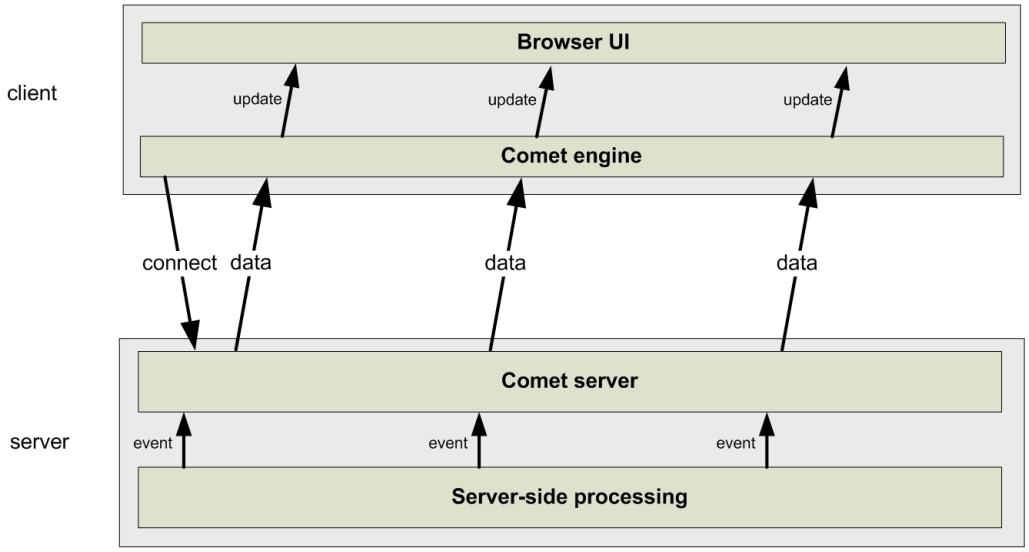

Figure 2.3: Comet communication style: streaming

knowing the size of the entire response body in advance of the transmission. The client is notified upon receiving of each chunk of data and is able to process it.

Because of numerous security policies such as the same origin policy [47] and limited content types allowed by browsers, the pure *forever-response* streaming cannot be used in Comet Web applications which are per definition based on Javascript. It has to be augmented with mechanisms for circumventing browser security mechanisms and tunneling of data chunks to the Javascript client.

Streaming via a *hidden iframe* is the most common streaming technique used in Comet applications. The technique is based on a hidden iframe and a never-ending HTTP response. The client opens a hidden iframe and issues a request to the Comet server. The server sends data in chunks and encloses each chunk in a HTML script tag with a Javascript callback function. The browser renders script elements as they are received and invokes the callback with data chunk as parameter. The incremental execution of script elements received in the iframe creates the effect of streaming: the callback function is provided with data as soon as it is received and can react on it, i.e. by updating GUI elements. If properly configured, the technique works in all browsers, which is the main reason it is the most commonly used streaming technique.

Another popular streaming technique is *interactive XHr streaming*. Streaming via interactive *XmlHttpRequest* is very similar to the streaming vie a hidden iframe, but is based on the *XmlHttpRequest* Javascript object instead of the iframe. *XmlHttpRequest* is an object that exists in all browsers and allows background execution of HTTP requests. Some browsers allow access to the content of the response being received via *XmlHttpRequest* before the server terminates the response. This can be used to achieve streaming to Web browser: the client reads response chunks as they come in and provides them to a dedicated callback function. If application-level

messages in the response are properly delimited they can be easily extracted from received data chunks and used in Comet applications.

The described streaming techniques all have the same pattern of interaction between a client and a server, which is depicted in Figure 2.3. They cause virtually no latency because a single TCP/IP connection and a single request/response cycle is used. The server always has a pending request and an open connection, so it can immediately send data as it gets available. However, the streaming techniques do not fully comply to the HTTP standard[48]. A strict interpretation of HTTP leaves no room for streaming. While the HTTP streaming works in most browsers without problems, it does not always work with proxies and firewalls.

For example, some firewalls are configured to allow only certain protocols on certain ports and actively inspect communication taking place on those ports. If the interaction between server and client does not seem to be HTTP conform, it is usually blocked. Another problem most streaming implementations are facing are intermediary proxies that cache data, which breaks the streaming.

A typical use case for streaming is market data distribution in the finance sector. Stock exchanges produce a very high volume of real-time data that characterizes the state of a financial security (or instrument). This state is represented by a number of fields with values. Some of these fields change only rarely, but some of them are updated at a very high pace, even several hundred times per second. In addition to high frequency updates there is also the requirement that clients need to receive updates virtually immediately after they occur. So, Comet applications that distribute financial data to clients have to push a large amount of data with very low latency. Streaming is the only technique that offers acceptable latencies and also scales well in such a scenario.

2.2 Bayeux protocol

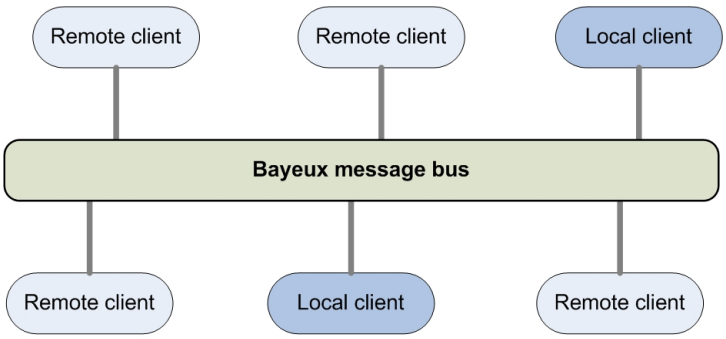

Figure 2.4: A logical communication schema defined by the Bayeux protocol

As a response to the lack of standardization for Comet applications, the Dojo Foundation released a protocol draft called “Bayeux”[50]. Bayeux can be defined as a protocol for transporting asynchronous messages in the Web. It is a JSON-based protocol designed to overcome

the traditional client/server nature of the Internet in general and HTTP in particular. The protocol provides means of two-way, low-latency communication between all participants by defining an additional communication abstraction on top of the transport protocol.

Unlike other Comet approaches, the Bayeux protocol is based on a publish/subscribe model [51, 52]. It essentially defines semantics for multi-point messaging over a point-to-point transport. While the publish/subscribe approach is rarely used in traditional Web applications, it offers a vastly better abstraction for Ajax applications with regards to flexibility, scalability and resource sharing. The model used in Bayeux, shown in Figure 2.4, is very simple and natural: clients act as subscribers, publishers or both, and the server acts as message bus and is responsible for routing messages between them. By subscribing to a channel, a client expresses its interest in a topic represented by the channel. Whenever a message is published to the channel, either by a remote or local server-side client, it is delivered by the server to all clients subscribed to that channel. The relationship between channels, subscribers and publishers is shown in Figure 2.5. This model achieves very high degree of logical decoupling of clients, which in turn allows for greater scalability and more dynamic and flexible network topology.

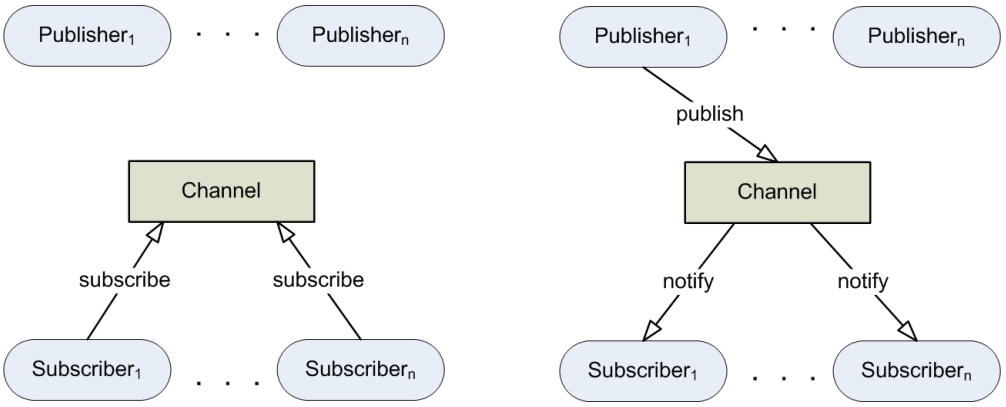

Figure 2.5: The relationship between publishers, subscribers and channels

Even though the publish/subscribe approach has its advantages, it is not suited for all applications. Some applications require simple request/response communication between client and server. For use cases where public broadcasting of messages is not an option, the Bayeux protocol defines special channels for private, point-to-point communication. These channels can be used to simulate request/response behavior without any additional costs.

Messaging semantics defined by Bayeux is independent of the underlying transport protocol, although the only transport protocol actually used throughout the specification is HTTP. This certainly makes Bayeux not as flexible as it could be, at least in its current version, but in scenarios where it is primarily used - Comet Web applications - this does not make much difference. Bayeux was designed for the Web, so it makes sense to focus it on the standard communication protocol used in the Web. But regardless of its focus on HTTP, Bayeux can be used on top of any protocol that supports the request/response paradigm.

In order to achieve bi-directional communication, the Bayeux protocol defines that clients should use two connections to the server. This way, messaging in both ways (server to client

and client to server) can occur simultaneously. However, regardless of the transport type used, at most one connection can be long-lived and in idle state, waiting for events to occur. The other connection is used to send requests to which the server immediately replies and is terminated immediately. This is important because of the *two connections per host limitation* posed by the HTTP protocol: the section 8.1.4 of the HTTP protocol specification[48] states that "a single-user client should not maintain more than 2 connections with any server or proxy" and most browsers adhere to this limitation.

2.2.1 A sample client/server interaction

Before further technical details of the protocol are discussed in the following sections, this section gives an overview of the protocol by presenting a sample interaction between a client and a server. For demonstration purposes, we shall describe a scenario with one remote client that acts as a subscriber and one local, server-side client that acts as a publisher. The remote client runs in a browser and the local client is an event source located on the server.

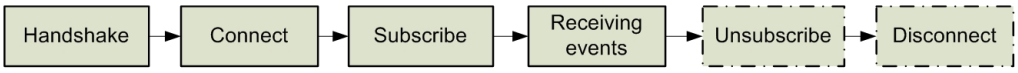

Figure 2.6: The interaction between the server and the remote client

A typical interaction between a client and a server consists of several process blocks shown in Figure 2.6. The client initiates communication by contacting the server and starting the handshake procedure. After communication details have been negotiated during the handshake, the client establishes a logical connection with the server by sending a *connect* request. Once the connection is confirmed by the server, the client will typically subscribe a set of channels it is interested in.

As with connect, subscription requests have to be confirmed by the server. Once confirmed, the client is subscribed and will receive messages published to these channels. Every time an event is published to a channel that the remote client subscribed to, the server will route the event message to the remote client. In this sample scenario, events are published by the server-side event source. The client receives messages published to subscribed channels and processes them as long as it is connected to the server. When it no longer wants to receive messages from a channel, the client unsubscribes by sending an unsubscription request. The same effect is achieved if the client disconnects from the server.

Figure 2.7 shows message exchange between a client and a server during a session presented in this section.

2.2.2 Transport types

As already stated, the Bayeux protocol separates the messaging semantics from the communication details. These details are encapsulated within *transport types*.

Transport types define the sequence and content of connections initiated by clients and how messages are wrapped up for delivery over the transport protocol. These may seem like im-

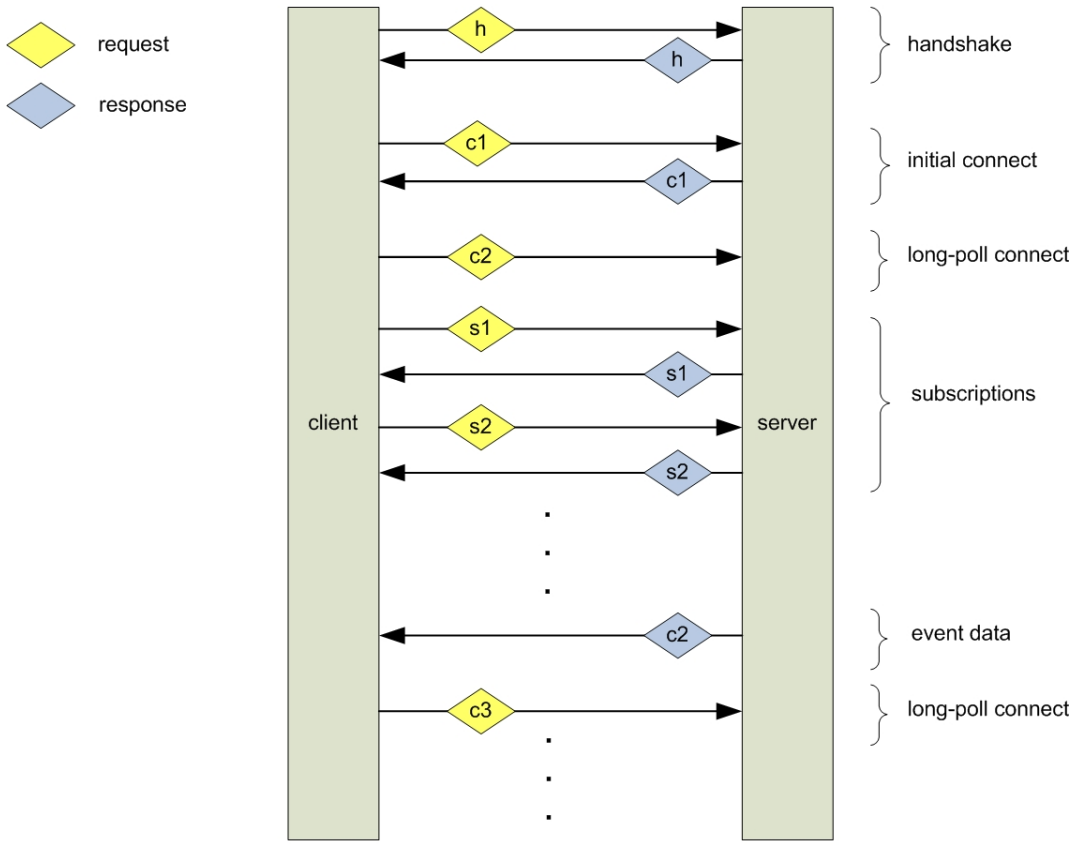

Figure 2.7: Message exchange between the client and the server

plementation concerns that should not be covered in a protocol specification, but it would be impossible to create interoperable Bayeux server and client implementations without fully specifying transport details. Which transport type is used is negotiated between server and client in the handshake phase, on connection initialization.

Bayeux transport types can be seen as more detailed and concrete specifications of the aforementioned communication styles long polling (Section 2.1.1) and streaming (Section 2.1.2) and their variations.

The protocol defines two transport types based on long polling communication style: one intended for same-domain and another for cross-domain data transfer. In order to be fully Bayeux compatible, server and client implementations have to support at least these two transport types, but are free to implement additional types.

Defining required transport types helps establish interoperability between different server and client implementations. Allowing additional, custom transport types greatly improves flexibility.

2.2.3 Channels

Like all publish/subscribe protocols, Bayeux uses channels as addressing mechanism. Channels represent topics or classes of interest, to which messages are published. They are identified by an absolute URI without parameters (i.e. "/chat/programming/" or "/stocks/GOOG/") and can be hierarchical.

For situations where clients want to subscribe to multiple channels that have the same parent segment, the Bayeux protocol defines *channel globbing*. Instead of sending one subscription request for each channel, clients can send only one request and use trailing wildcards in the channel name to specify that they wish to subscribe to all child channels. A single wildcard ("*") matches single segment while a double wildcard ("**") matches multiple segments. By using channel globbing, a client's bandwidth is preserved, but far more important is the reduction of overhead for subscriptions induced on the server.

The protocol defines two special cases of channels: meta and service channels. Meta channels begin with "/meta/" and are reserved for use by the Bayeux protocol itself. Service channels are located under "/service/" and are designed to assist request/response communication. Clients are not allowed to subscribe any of these channels.

2.2.4 Messages

Bayeux messages are JSON[53] encoded objects that contain an unordered sequence of name/-value pairs representing fields and their values. The protocol defines a list of valid fields with their value types and messages where they can be used. The field set is fixed, but there is a special field called "ext" that serves as extension point and can be used to transfer any kind of data.

The protocol defines only seven types of messages. There are six meta message types used for communication between a server and a client, and there is one separate message type used to deliver event data to clients. All message types are briefly described in this section.

Handshake These messages are used for negotiation of connection details. The client initiates connection negotiation by sending a handshake request to the */meta/handshake* channel. The request message tells the server what connection parameters the client supports (i.e. protocol version and a list of supported transport types). On successful handshake, a handshake response is sent back to the client with a unique client id, the used connection type and a flag stating that the handshake was successful. If the parameters in the handshake request cannot be met by the server, it sends an unsuccessful handshake response back to the client. Complex connection negotiations may require multiple handshake request/response pairs to be exchanged until the server and the client agree on all communication parameters.

Connect After successful handshake exchange between client and server, the client can establish a connection to the server by sending a Connect request to the */meta/connect* channel. Depending on the transport type used, the server can respond immediately or wait until there is data to be delivered to the client.

Disconnect When a connected client wishes to disconnect from a server, it sends a disconnect request message to */meta/disconnect*. Usually, no additional data is contained in this request and the server responds with an empty disconnect response. This message type is used to signal graceful termination of the client and initiate cleaning of resources allocated on the server for the client.

Subscribe Clients express interest in topics by subscribing to appropriate channels. In order to subscribe to a channel, a connected client needs to send a subscribe request containing names of channels it wants to subscribe to */meta/subscribe* channel. The server responds with a subscribe response which always contains all channel names specified by the client, a flag stating if the subscription was successful or not. There is also a range of optional fields such as "timestamp" or "ext".

Unsubscribe Connected clients can cancel their interest in topics by unsubscribing from channels. Unsubscribe response and request messages follow the same logic and contain the same dataset as subscribe messages, except that they are sent to */meta/unsubscribe* channel.

Publish Clients can publish messages by sending a publish request to the server. The request has to include the channel name to which the client wishes to publish and the data it wishes to publish. The client does not need to be connected, although the server may decide to reject publish requests from unconnected clients. In any case, the server responds with a publish response, which contains a flag stating if the publish operation was successful or not.

2.2.5 Client state handling

Bayeux clients are always in one of three states defined by the protocol: disconnected, connecting and connected.

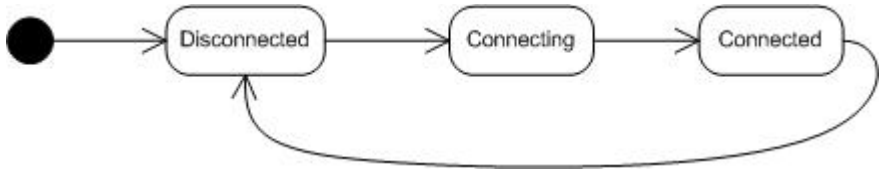

Figure 2.8: Bayeux client state transitions

The client is in disconnected state prior to the handshake or after it has sent a disconnect request. Generally, disconnected clients cannot communicate with the server, but there is one exception: they can send publish messages. A server can, but does not have to accept publish messages from disconnected clients. Publish messages from remote disconnected clients will be usually ignored, but local clients will be usually permitted to publish messages without a need to connect.

Connecting is an intermediary state that client has only during the handshake phase. After successful handshake and after it receives a connect response with a "successful" flag, a client

is considered connected. This does not necessarily mean that there is a permanent transport protocol connection to the server. Even if the underlying connection is broken for a short period of time, a client remains in the connected state. It changes to *Disconnected* state only if a timeout occurs or the client explicitly requests to be disconnected by sending an disconnect request. In the *Connected* state, clients can subscribe to channels, unsubscribe from them, receive and publish messages.

2.2.6 Security

The protocol does not provide any details on authentication or authorization. It only vaguely covers two ways to achieve authentication: container supplied authentication and Bayeux extension authentication. Container supplied authentication means that the application relies on authentication services provided by the hosting container (i.e. servlet container). One popular mechanism is session-based authentication with cookies. Bayeux extension authentication is defined as authentication mechanism that exchanges credentials and tokens within Bayeux messages ext fields. However, the protocol only states that the ext field may be used to exchange authentication challenges, credentials and tokens, without specifying details such as data structures or algorithms for generating or validating security credentials.

2.3 JSON

The Bayeux protocol uses JSON (Java Script Object Notation)[53] as message exchange format. JSON is a lightweight data-interchange format based on a subset of the Javascript Programming Language. It is completely language independent but uses conventions that are familiar to programmers of the C-family of languages (such as C++, Java, C#, Javascript, Perl, etc.). JSON is mainly used in Javascript applications, but it can be used to represent data structures from virtually any programming language. The notation uses only two constructs to represent data:

- A collection of name/value pairs. This represents an object, record, struct or a dictionary.
- An ordered list of values. This represents an array or a sequence.

JSON is easy to parse, quite simple and at the same time quite versatile. These properties and the availability of libraries for a number of programming languages make JSON an ideal data-interchange language.

Compared to XML, it offers several advantages for Ajax and Comet applications. It has much simpler syntax than XML and much smaller data encapsulation overhead. This in turn significantly reduces the amount of data being transfered between server and client. Additionally, JSON objects are syntactically legal Javascript objects and can be directly interpreted in Javascript.

CHAPTER 3

Comet.NET

This chapter briefly presents functional and non-functional requirements for Comet.NET, discusses the architecture of the system and provides insight into various implementation details. The goal was to not only describe the requirements and the architecture, but also to provide reasons and rationals behind particular functional and architectural decisions.

3.1 Requirements

This section presents functional and non-functional requirements posed on Comet.NET. The process of designing the architecture of the system was guided by the goals and principles described in this section.

3.1.1 Functional requirements

Full Bayeux protocol compliance The main functional requirement is the *unconditional compliance* to the Bayeux protocol. An implementation is considered unconditionally compliant if *it satisfies all the must or required level and all the should level requirements of the protocol*[50]. The unconditional compliance will ensure interoperability of Comet.NET with other Bayeux-compliant client-side applications. Besides that, it will allow functional and performance comparisons between Comet.NET and other Bayeux-compliant server implementations.

Streaming communication The Bayeux protocol requires implementations to support the long polling communication style, but does not pose any requirements regarding streaming. However, we consider streaming an equally important Comet communication style and define that Comet.NET has to support both long polling and streaming. Furthermore, we pose a requirement that Comet.NET must support the three streaming techniques: forever-response streaming, streaming via hidden iframe and interactive xHr streaming. These streaming techniques are described in Section 2.1.2.

Synchronous and asynchronous programming model The application-level API exposed by Comet.NET has to allow client applications to invoke I/O-bound operations both in the synchronous and the asynchronous manner. The synchronous programming model is appropriate if the application wants to block while waiting for the I/O-bound operation to complete. If however the application wants I/O-bound operations to be executed in a separate thread, the asynchronous model has to be used. Comet.NET shall use the Event-based Asynchronous Pattern [70] to provide asynchronous API methods.

Efficient handling of HTTP connections Comet applications have a radically different traffic profile than traditional Web applications. A request in a standard Web application means that the server has to perform a task and return the results as soon as the task is finished. Traditional Web servers are optimized for this kind of short-lived requests that are always associated with task execution. However, in a Comet application, a request does not cause the server to execute a task. Instead, it is parked on the server most of the time and used to deliver event data as soon as the event occurs. An efficient implementation of this key aspect of a Comet server - handling of long-lived HTTP connections - is a very important functional requirement of Comet.NET.

Configurability Comet.NET has to offer a wide range of configurable properties so that its behavior can be easily adjusted at runtime. The configuration has to be implemented by standard means of the underlying framework. This makes it easy for applications to use standard application configuration files for persisting Bayeux configuration settings. The configurability is only required prior instantiation of the server. Changing of configuration settings on-the-fly, after the server has been started, is not required.

3.1.2 Non-functional requirements

Extensibility The system must offer a high level of extensibility. It must be possible to add new capabilities or change the behavior of existing components without significant changes to the underlying architecture. This can be achieved by using well-defined interfaces for communication between components in the system and defining various observation and interception points. These points can be used to dynamically attach custom features or override the default behavior. However, the extensibility of the system must not significantly reduce its usability. Drawing a line between extensibility and usability is not an easy task, because these two features contradict each other. A certain trade-off between simplicity of the Comet.NET public API and increased extensibility is expected, but it must not lead to a cumbersome and obtrusive system.

Embeddability In addition to the stand-alone mode, the server has to be easily embeddable into arbitrary .NET applications. For example, it must be possible to plug Comet.NET into an enterprise application and make data from the application's backend store *pushable.* Also, libraries that wish to integrate Comet functionality should be able to easily base relevant parts on Comet.NET. In order to achieve this goal, the server will be developed as a library and can as such be referenced in any application based on the .NET framework. One of the key aspects here

is the public API of the library, that is, interfaces and classes that will be used by applications to interact with Comet.NET.

Robustness and fault-tolerance Great care needs to be taken to make the server resilient to invalid or unexpected client behavior as well as to suboptimal conditions in the runtime environment. The server should handle communication with clients according to the robustness principle for Internet protocols, also known as Postel's Law: *be conservative in what you send, liberal in what you accept*[60]. In case of component failures or resource shortages (i.e. high memory or CPU consumption), the service efficiency and speed can gradually be decreased, but the server has to continue to operate and serve clients. This property is also known as *graceful degradation*.

Scalability This is one of the top requirements and at the same time one of the top challenges. The server has to be able to handle a growing number of concurrent users and messages in a graceful manner. Since Comet.NET is a stateful server and needs to keep session state data for each connected user, this is not a trivial issue. Another important aspect is that the scalability of the system does not only depend on Comet.NET, but rather on application and services on top of it. The key goal here is to design Comet.NET in a way that it allows load balancing of multiple instances of applications based on it with minimal architectural constraints. It should even be possible to employ Comet.NET in a "shared nothing" distributed system[61] if the application layer supports it.

3.2 High level overview

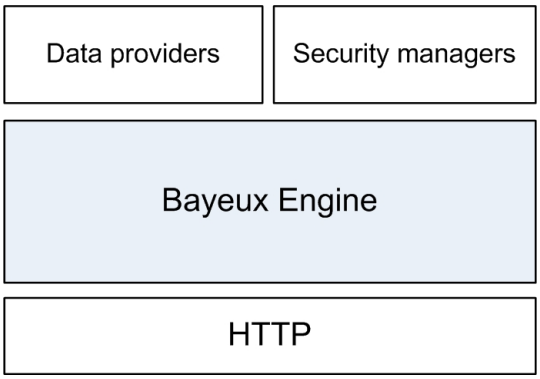

Figure 3.1: View from bird's perpective

Comet.NET is best described as a bus for routing events between remote clients using the publish/subscribe communication paradigm. It resembles a typical messaging middleware, with the main difference that it is focused on the Web. The high-level components of a system based on Comet.NET are shown in Figure 3.1.

The main part of the system is the Bayeux engine, which sits between the HTTP handling module in the bottom and data providers and security managers on the top, and handles the routing of messages between clients via named channels.

When viewing the system from a bird's perspective, it can be divided into three distinctive parts:

- HTTP: the transport of messages.
- Bayeux engine: responsible for handling of Bayeux specific details such as messaging, JSON marshalling, subscriptions and event delivery.
- Data providers and security managers: application-level components that provide services on top of the Bayeux engine.

The actual handling of HTTP traffic is not part of Comet.NET. The bottom layer is quite thin and serves only as wrapper for an actual HTTP handling library. The gray part, the Bayeux engine, is the server core, where most of the code and logic resides. The topmost layer is supplied by the application that provides services based on Comet.NET.

3.3 Static structure

After the high level overview of the system presented in the previous section, we continue with the decomposition of the architecture in a top-down manner. Section 3.3.1 presents the overall system packages and Section 3.3.2 discusses the system layers.

3.3.1 Packages

On a very abstract level, the architecture of Comet.NET consists of the six packages shown in Figure 3.2. These packages represent a compile-time logical architecture of the system. Each of them consists of a number of related classes that implement one or more related features.

The package `Transport and communication` contains classes that implement mechanisms for the communication with the outer world. Typical responsibilities of classes from this package consist of dealing with HTTP request and responses, handling the incoming and outgoing data chunks and performing their basic sanitation.

The `Messaging and marshalling` package consists of classes that represent messaging constructs defined in the Bayeux protocol and offer marshalling and unmarshalling services and hooks.

Security-related classes are contained in the package `Security`. Since the Bayeux protocol does not define explicit security mechanisms, this package consists mainly of interfaces and hooks intended for injection of custom security mechanisms provided by applications based on Comet.NET.

The `Collections` package contains various collections that are custom-tailored to fulfill requirements posed by other parts of the system.

The package `Bayeux handlers` groups together classes and interfaces that implement the business logic defined in the protocol.

The package `Bayeux domain` offers an object-oriented abstraction of the constructs defined by the Bayeux protocol.

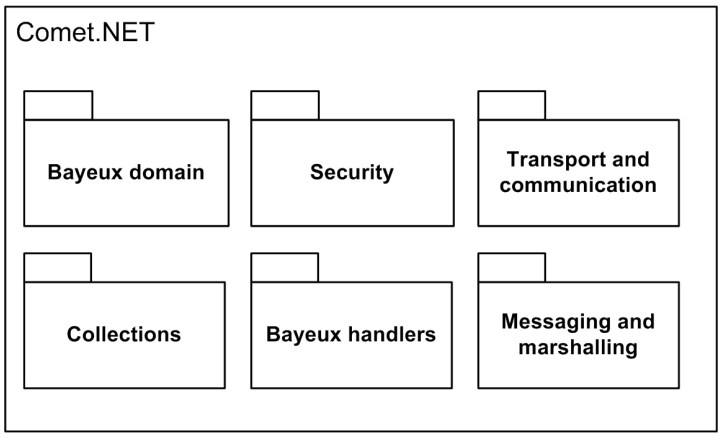

Figure 3.2: Comet.NET package overview

3.3.2 Layers

Comet.NET has a layered architecture[62] and constitutes of four layers. Each layer has well-defined roles and responsibilities. Layers are as self-contained as possible and communicate with each other only via interfaces, which makes them agnostic of consumers. The communication is one-way, top-down and each layer uses only functions from the layer directly beneath it. Coupling between them is kept low, while at the same time cohesion within each layer is high. Ultimately, the goal was to make layers replaceable without too much effort.

The Figure 3.3 shows the decomposition of the system into layers. The rest of this section contains brief descriptions of each layer, along with their responsibilities.

The physical layer at the bottom of the system is responsible for communication with remote clients. It receives and sends data by using a transport protocol with support for the request/response paradigm. The specification of the Bayeux protocol is heavily based on HTTP as transport protocol and although it states that any request/response based protocol can be used as transport, there are currently no other protocols actually used. Bayeux is intended for use on the Web and HTTP is the standard communication protocol in this environment, so it can be argued that the decision to tightly couple Bayeux and HTTP is a reasonable one. Following this reasoning, the physical layer in Comet.NET was designed to work with HTTP and provides methods for accepting HTTP requests and sending HTTP responses. Tight coupling of the physical layer to HTTP removes the need for an additional abstraction of the communication protocol and makes the implementation of the layer as well as its interface to the adjacent layer much simpler. However, in spite of tight coupling to HTTP, introduction of another request-based communication protocol would require only local changes. Since layers are allowed to communicate only with adjacent layers, addition of another transport mechanism would only require changes to

several parts of the second layer (message marshalling). Layers above the message marshalling layer would not be affected.

The next layer is the message marshalling layer. It is responsible for converting raw input data received from the physical layer into Bayeux messages and vice versa. Data is transported in form of JSON objects, so this layer serializes and deserializes higher level objects that represent messages to and from JSON strings. All layers above operate only on high level representations of Bayeux messages (`IMessage` objects) and have no contact with JSON. The Bayeux protocol defines several combinations of Web methods (`GET`, `POST`) and encodings that can be used for communication between client and server. This layer encapsulates logic for proper handling of various input/output combinations and shields upper layers from dealing with these details.

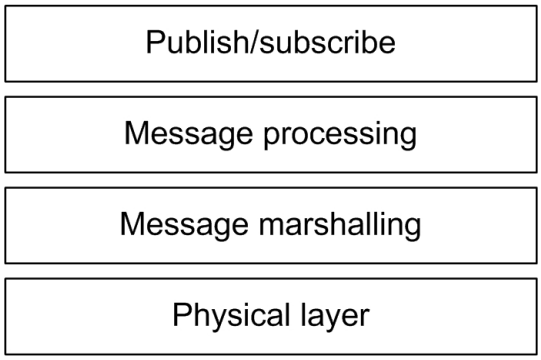

Figure 3.3: Comet.NET layers overview

The business logic is located in the message processing layer. Components of this layer give meaning to messages and process their content. Messages are validated and processed by one of several message handlers. Each message handler encapsulates logic for processing one type of Bayeux requests (details are discussed in Section 3.4.5). This allows clean enforcement of separation of concerns in the layer. Another responsibility of this layer is creation and removal of high level business objects that represent a large part of the public API of Comet.NET: channels, clients and subscriptions.

The top layer is the representation of the publish/subscribe paradigm of the Bayeux protocol. This is the interface that applications based on Comet.NET use to interact with remote clients. The layer contains business level objects already mentioned in the previous paragraph: channels, clients and subscriptions. Channels represent Bayeux topics, clients represent connected remote clients and subscriptions are expression of interest of a client for a channel. The central point of the server, the class `BayeuxServer`, is also part of the layer. In addition to providing means of interaction with clients, this layer is also responsible for various notifications towards the client application. An example notification would be a creation of a new channel or a new subscription. These notifications can be used by applications based on Comet.NET to trigger appropriate actions when certain events occur.

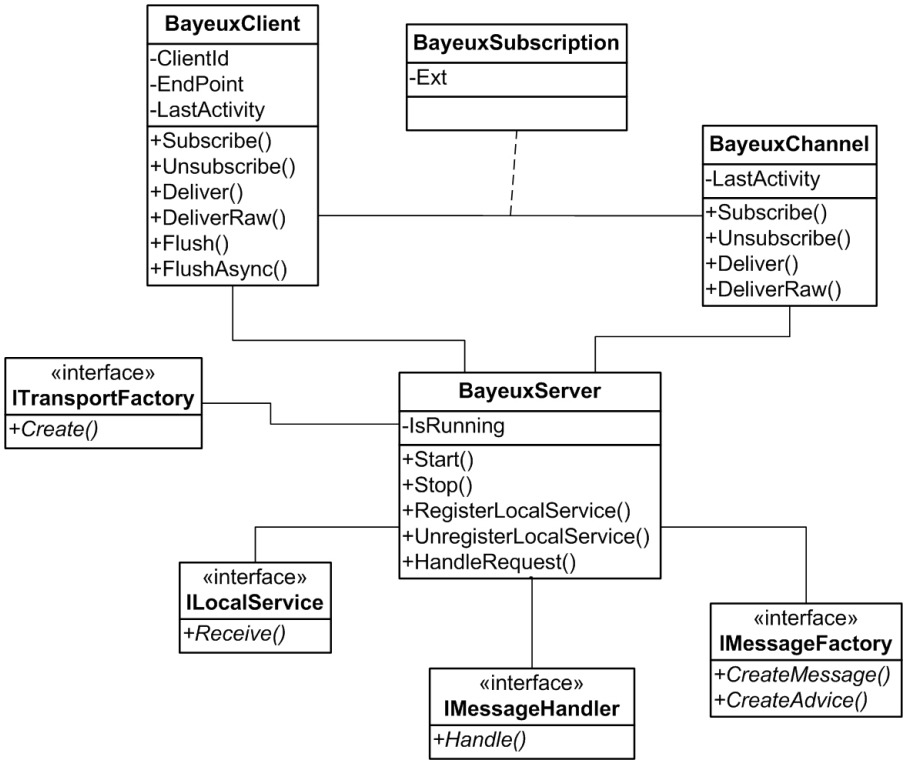

Figure 3.4: Dependencies of the `BayeuxServer` class

3.4 Domain model

This section represents a more detailed overview of Comet.NET's architecture. It presents important classes from the domain model and shows their static relationships in form of UML diagrams[63]. The section also provides insight into some relevant implementation details.

The domain model is divided into coherent logical parts - subdomains or modules - that contain tightly coupled classes with similar responsibilities. Each section describes one of the parts. Section 3.4.1 describes the coordination within Comet.NET. Section 3.4.2 explains how HTTP requests are handled. Section 3.4.3 explains how messages and related facilities are modelled. The design of the transport concept is presented in Section 3.4.4. Classes involved in processing Bayeux requests are presented in Section 3.4.5. Section 3.4.6 describes the classes that implement the publish/subscribe model in Comet.NET.

3.4.1 Coordination

`BayeuxServer` is the main class in the package. It acts as mediator and orchestrates other parts of the library such as `HttpListener`, message handlers, clients or channels. The relationship between `BayeuxServer` and other classes in the system is shown in Figure 3.4.

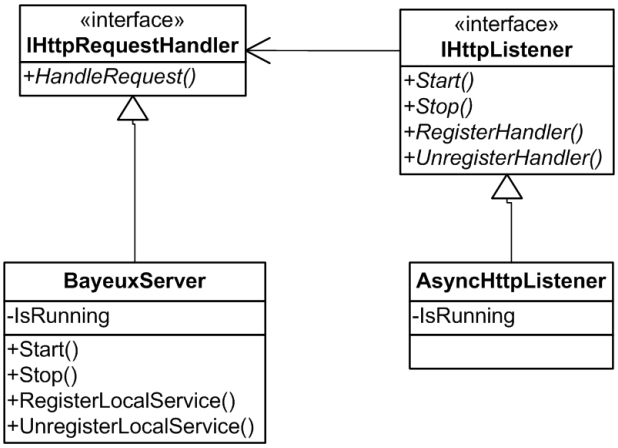

Figure 3.5: Relationship between `IHttpListener` and `IHttpRequestHandler`

An instance of this class represents a fully functional Bayeux end point. Although the class holds a lot of references to other classes and seemingly has several responsibilities, it is not a *God class*[64]. It is quite coherent and has only one true responsibility: coordinate the processing of incoming requests by delegating work to appropriate objects. Client applications use `BayeuxServer` as the entry point for interaction with Comet.NET. It must be noted however that it is not the sole interface between the client applications and the library. For the actual interaction with remote clients, the classes `BayeuxClient`, `BayeuxChannel` and `BayeuxSubscription` have to be used. The main reason for not having a facade[67] is that the number of public classes is reasonably small and their relationships are very clear.

3.4.2 Request handling

The relationship between interfaces presented in this section is shown in Figure 3.5.

HTTP requests are provided by an implementation of the `IHttpListener` interface. The listener is responsible for accepting incoming HTTP requests and handling them over to a registered `IHttpRequestHandler`. The only available `IHttpRequestHandler` implementation is `BayeuxServer`, but using interfaces for interaction between these two parts greatly reduces coupling. `IJsonMarshaller` is responsible for deserialization of the content of HTTP requests. It creates one or more high-level representations of Bayeux requests in form of `IMessage` objects.

3.4.3 Messaging

`IMessage` objects are common entity objects that contain only data and methods for data validation. They are the only objects allowed to traverse layer boundaries and can even be used by applications on top of Comet.NET.

They are created with factories that implement the `IMessageFactory` interface[67]. By providing custom implementations of `IMessageFactory` and `IMessage`, client appli-

cations can easily supply their own message classes that will be used in Comet.NET. Additionally, serialization and deserialization of messages can be customized by providing another implementation of `IJsonMarshaller`. The triple `IMessage`, `IMessageFactory` and `IJsonMarshaller` offers a great degree of flexibility, while keeping the interface clean and simple.

The relationship between the interfaces presented in this section are shown in Figure 3.6.

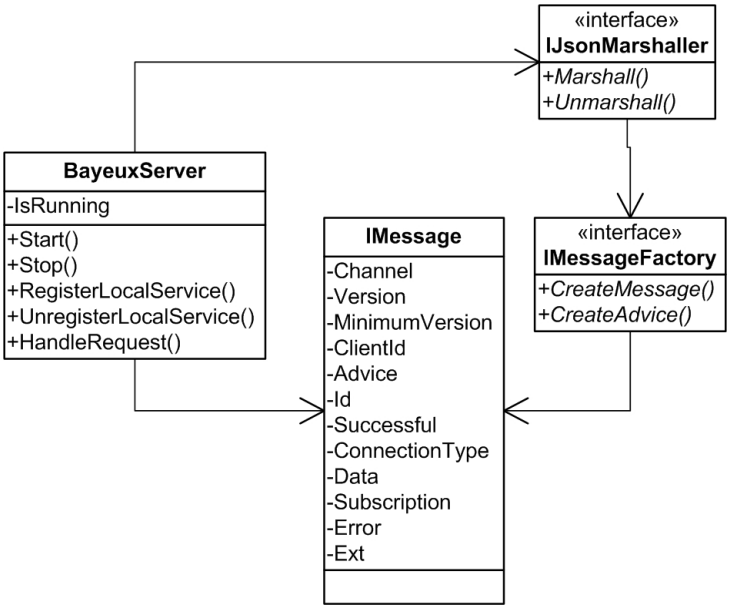

Figure 3.6: Relationships between message-related interfaces and `BayeuxServer`

As defined in the protocol, messages are maps of key-value pairs. Each message type has a set of required and optional fields. A simple approach would be to define message as a key-value store similar to associative hash tables and several convenience methods for accessing common fields such as *channelId* or *clientId*. Although very flexible and easily extensible, this approach has several downsides. The main problem is the weakly typed nature of message objects: the client is responsible for using proper field names, as defined in the protocol, and providing their values as objects of appropriate types. For example, the field "success" can only have type boolean, but the field "supportedConnectionTypes" is an array of strings. This cannot be checked at compile time with messages as loosely typed maps of keys and values. Another downside of this approach is that messages are not self-documenting and developers that write applications based on Comet.NET would need to consult the documentation more often.

The opposite, strongly typed approach would be to create a class hierarchy with a subtype for each message type defined in the protocol. Each subclass would have only properties defined in the protocol and it would not even be possible to create invalid messages (other than leaving required fields empty on purpose). However, such a hierarchy introduces overhead in both design and implementation, which is not justified by the gains from it. The main problem is determining the concrete type of the message after it has been received and unmarshalled. The best approach

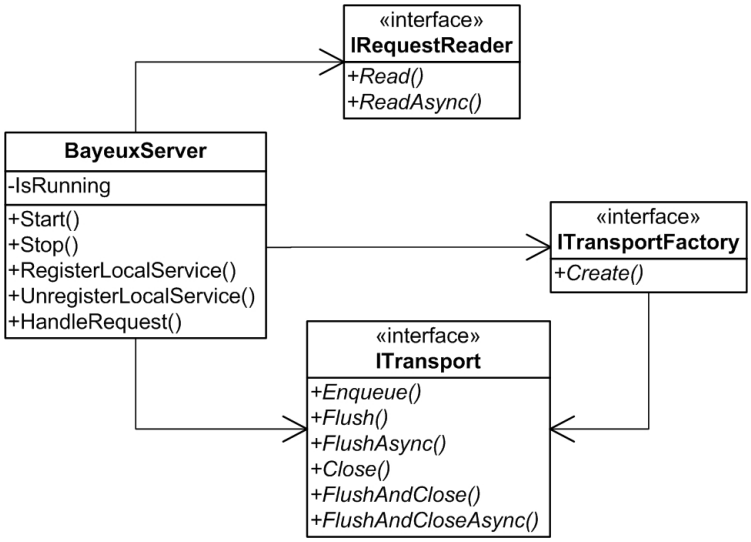

Figure 3.7: Relationships between transport-related interfaces

would be to write methods for handling each message type and let the runtime determine the dynamic type of the message and invoke the appropriate method. However, the .NET Runtime 4.0[65] does not support multiple dispatching[66], so we cannot use this approach. Switch statements would be required to determine the type of the message, which must then be casted to its most specific type and processed accordingly. A better solution for the problem is provided by the visitor pattern[67], but the visitor pattern has it own set of limitations: it reduces the ability to introduce new subclasses and hence reduces the extensibility.

After carefully reviewing possibilities, a hybrid approach was chosen. There is one interface - `IMessage` - that exposes all message fields defined by the protocol as properties. The name of the property represents the name of the field in the Bayeux message and the type of the property defines the field type. This allows compile-time type safety of messages. However, since there is only one interface for all types of Bayeux messages, clients still have to take care which properties have to be set for a particular message type. A set of validators provides run-time message validation and helper methods allow clients to easily construct valid messages.

3.4.4 Transport

Relationships between classes and interfaces described in this section are shown in Figure 3.7.

`ITransport` encapsulates a HTTP response and specific details such as encoding and behavior (for example, streaming vs. polling). The interface hides low-level communication and serialization details and offers methods for enqueuing and flushing (sending) of `IMessages` to remote clients. Just like messages, instances of transport classes are created with factory classes that implement the `ITransportFactory` interface. However, unlike messages, the creation of transport objects is dependent on the HTTP request. The factory creates an appropriate transport instance based on the content of the HTTP request. This is needed because the

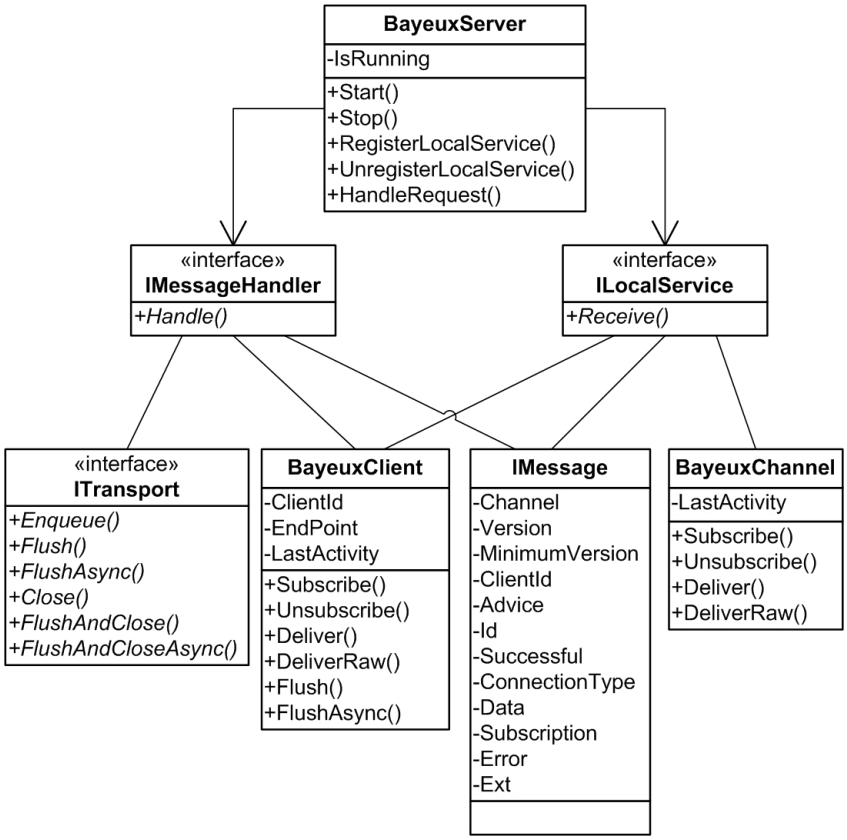

Figure 3.8: Static structure of system parts that deal with message processing

Bayeux protocol defines several possible combinations for communication between client and server and properties of the response depend on properties of request.

The transport can be seen as a sink where various parts of the system can enqueue messages, without knowing when or how they will be actually transported to the remote client.

3.4.5 Processing

The actual processing of requests is the responsibility of implementations of the `IMessageHandler` interface. There is a message handler for every type of Bayeux message. The decision which handler will process which message is made by `BayeuxServer`. Handlers differ greatly in complexity and dependencies on other classes, but all of them have very narrow and well-defined responsibilities. The relationship between interfaces and classes described in this section are shown in Figure 3.8. and concrete implementations of depicted interfaces are described in the following paragraphs.

`HandshakeHandler` is responsible for processing of handshake request messages and serves as entry point for the authentication process. It uses the associated implementation of `IBayeuxSecurityManager` to determine if the client is allowed to connect to the server

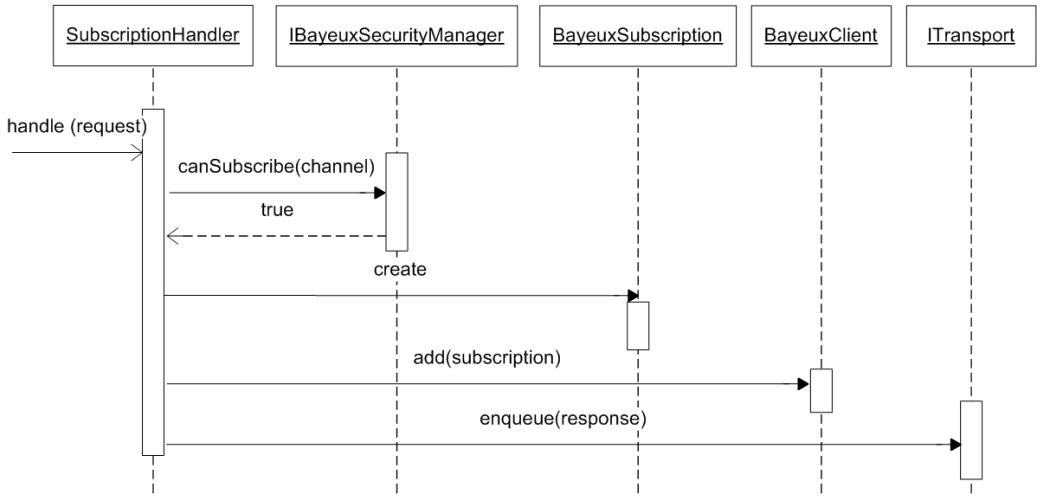

Figure 3.9: The handling of a subscription request

and if so, proceeds to select compatible connection types depending on client properties such as user agent (browser), create a client representation on the server in form of an instance of the `BayeuxClient` class and send the appropriate response message via the associated implementation of `ITransport`.

`ConnectHandler` handles connect request messages. These messages are mainly used to indicate a long-lived request that should be used for event delivery, so there is not much logic involved in their processing. The handler sets the implementation of `ITransport` into the state in which it can be used for asynchronous sending of events and creates a connect response message. ConnectHandler's counterpart - `DisconnectHandler` - is responsible for processing of disconnect request messages. It removes the client state and frees used resources such as event queues.

`SubscribeHandler` and `UnsubscribeHandler` are responsible for handling of subscribe and unsubscribe request messages, respectively. The sequence diagram in Figure 3.9 shows the collaboration between components during the processing of a subscription request. For each channel contained in the subscribe request, the handler uses `IBayeuxSecurityManager` to check if the client is authorized to subscribe to the channel. For subscriptions that are allowed, the handler establishes a connection between the `BayeuxChannel` and `BayeuxClient` by creating a `BayeuxSubscription` object and creates the response message with the "successful" flag. For disallowed subscriptions, the response with "successful" flag set to false and appropriate error message is sent. Handling of unsubscribe requests is considerably simpler: apart from consistency checks, the handler only has to remove the logical connection between channel and client and send an appropriate response.

`PublishHandler` processes publish messages. The publishing of messages is plugged into the security process the same way handshake and subscription are: `IBayeuxSecurity-Manager` is used to determine if a client can publish the provided message to the specified channel. If allowed, the handler has to schedule the distribution of the message to all clients

subscribed to the channel and also send a confirmation message to the publishing client. In case the publishing was not allowed, an error message can, but does not have to be sent to the client. The decision if an error message should be sent to the client is made in the implementation of the security manager.

`ILocalService` is a special case of a message handler. Client applications can attach custom implementations of this interface to `BayeuxServer` and subscribe to one or more channels. Every time a message is received on one or more specified channels, it is forwarded to `ILocalService` with context objects such as `BayeuxClient` and `BayeuxChannel`. The service can then react on a message according to its content as well as internal logic of the service. This interface is best suited for request/response types of services, where a remote client communicates only with the server and consumes a service. Clock synchronization and checking of e-mail inbox are typical use cases that rely on such communication style. The Bayeux protocol defines special channels for this kind of communication, so called service channels (/service/). Local services are not limited to request/response communication only: they can listen on multiple channels and also publish on multiple channels. For example, a simple chat service can be implemented on top of Comet.NET by implementing `ILocalService`. This is demonstrated in section 4.1.

3.4.6 Publish / Subscribe

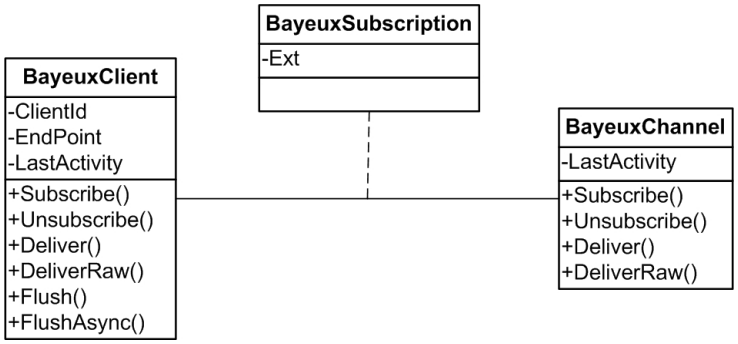

Figure 3.10: Static structure of publish / subscribe classes

The publish/subscribe paradigm of the Bayeux protocol is represented by the classes `BayeuxChannel`, `BayeuxClient` and `BayeuxSubscription`. The classes are presented in Figure 3.10.

Channels are the addressing mechanism of Comet.NET. They have unique names, can be hierarchically structured and represent topics or classes of interest. The class `BayeuxClient` is the server-side representation of a remote client. It holds session data such as a list of subscribed channels and a queue of messages that are waiting to be pushed to the client. This class also has an optional reference to an implementation of `IDataFilter` that is used for filtering outgoing messages on each push operation. `BayeuxSubscription` is an association class, a logical link between a client and a channel that expresses that a client wants to receive messages pub-

lished on a particular channel. Unlike client and channel, it is very simple and contains almost no logic.

Instances of these three classes are used by applications on top of Comet.NET for interaction with the library and indirectly with remote clients. Applications can publish messages to a channel, which in turn sends provided messages to all clients subscribed to that channel, or they can send messages directly to clients. They can use events raised by `BayeuxClient` and `BayeuxChannel` to react on situations such as subscription of a new channel or removal of a client. And finally, they can use various properties of these two classes to inspect their state (i.e. last activity, connection type, id, etc).

3.5 Message flow

The previous section gave an overview of the important server components, including their static structure and some implementation details. This section presents a message flow through Comet.NET and describes how these components interact with each other. The sequence diagram in Figure 3.11 shows a sample message flow in case where incoming requests are successfully processed by an `IMessageHandler`. The sequence diagram in Figure 3.12 shows a sample message flow in case that incoming requests cannot be unmarshalled.

Incoming HTTP requests are received by an implementation of `IHttpListener`. On each request, an instance of `HttpContext` is created with request data and an response object that can be used to send response to the client. This context object, containing all relevant data of the request/response pair, is then passed to `BayeuxServer` for actual processing. It is worth mentioning here that `IHttpListener` and `BayeuxServer` are not tightly coupled: `IHttpListener` knows statically only `IHttpRequestHandler`, which defines a single method for processing HTTP requests. Implementations of `IHttpListener` have one or more registered implementations of `IHttpRequestHandler`, to which they forward incoming requests. `BayeuxServer` implements the interface `IHttpRequestHandler` and registers itself as the request handler on initialization.

The main responsibility of the `BayeuxServer` is to coordinate the handling of incoming HTTP requests. First, it uses an implementation of `IRequestReader` to read the request and get its content in string form. As specified by the protocol, the content of the request is one or more Bayeux messages, so the second step is unmarshalling messages from the request content. An implementation of `IJsonMarshaller` is used for this purpose: it is provided with a string representation of the request content and returns one or more `IMessage` objects. All classes involved in processing of messages write their output, if there is any, to `ITransport`. The server instructs `ITransportFactory` to create an appropriate instance of `ITransport` using the request and response objects, as well as messages deserialized by `IJsonMarshaller` in the previous step. After this step is finished, `BayeuxServer` is ready to start the actual inspection and processing of each `IMessage` from the request.

Only basic rules for message validity, such as that every message must have a channel, are enforced in the `BayeuxServer` itself. If a message fails to meet these basic validity rules, a generic error message is sent and the ITransport object associated with the request is closed. Otherwise, messages are forwarded to appropriate implementations of `IMessageHandler`

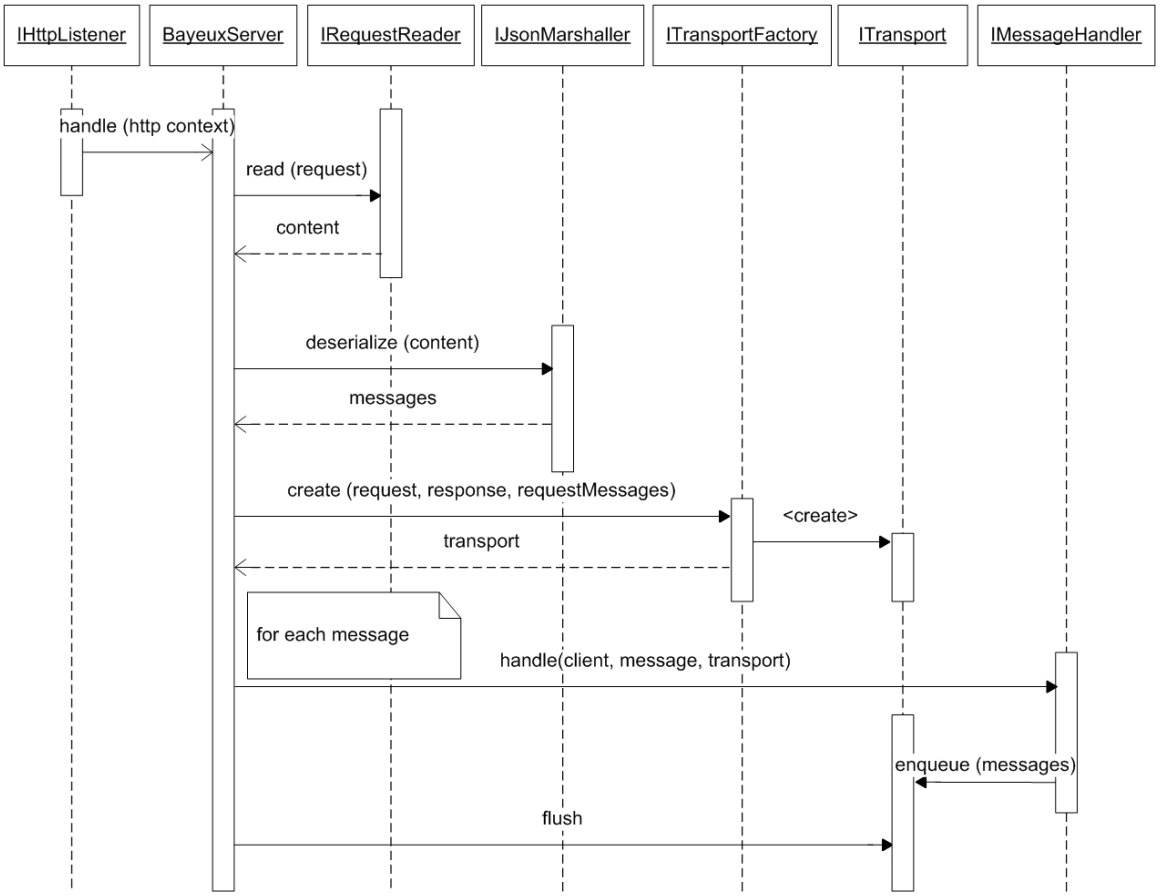

Figure 3.11: A sample message flow in Comet.NET (normal case)

for actual processing. `BayeuxServer` uses the channel name to determine the type of the message, which in turn determines which handler has to be used for its processing.

Since `IMessageHandler` expects not only `IMessage` and `ITransport` objects, but also an instance of `BayeuxClient`, BayeuxServer has to do one more lookup before messages can be forwarded to actual processing. It uses `ClientStore` to get the client associated to the client ID contained in the message (omitted in Figure 3.11). Handshake messages are handled differently, because they are sent by an unconnected client and therefore do not have a client ID. If the server encounters a handshake message, a new instance of `BayeuxClient` is created and added to the store.

After instances of `IMessage`, `ITransport` and `BayeuxClient` are available, `BayeuxServer` can invoke appropriate implementations of `IMessageHandler` or `ILocalService`. The decision for the invocation can be described as follows:

1. If a message is sent to a meta channel, find the implementation of `IMessageHandler` responsible for this meta channel and invoke its `Handle` method

2. If a message is sent to a channel that was registered by an implementation of `ILocalService` instead of `IMessageHandler`, invoke its `Handle` method

3. Otherwise, invoke the special handler for publishing of the message (`PublishHandler`)

The instance that handles an incoming message can enqueue one or more response messages to `ITransport` that was provided as parameter of the invocation. Per convention, classes that process messages do not flush or close the transport. This is important, because there is a one-to-many relationship between incoming messages and transport: there can be many messages that were received in the same HTTP request and therefore share one transport. It is the responsibility of `BayeuxServer` to decide if and when a transport should be flushed.

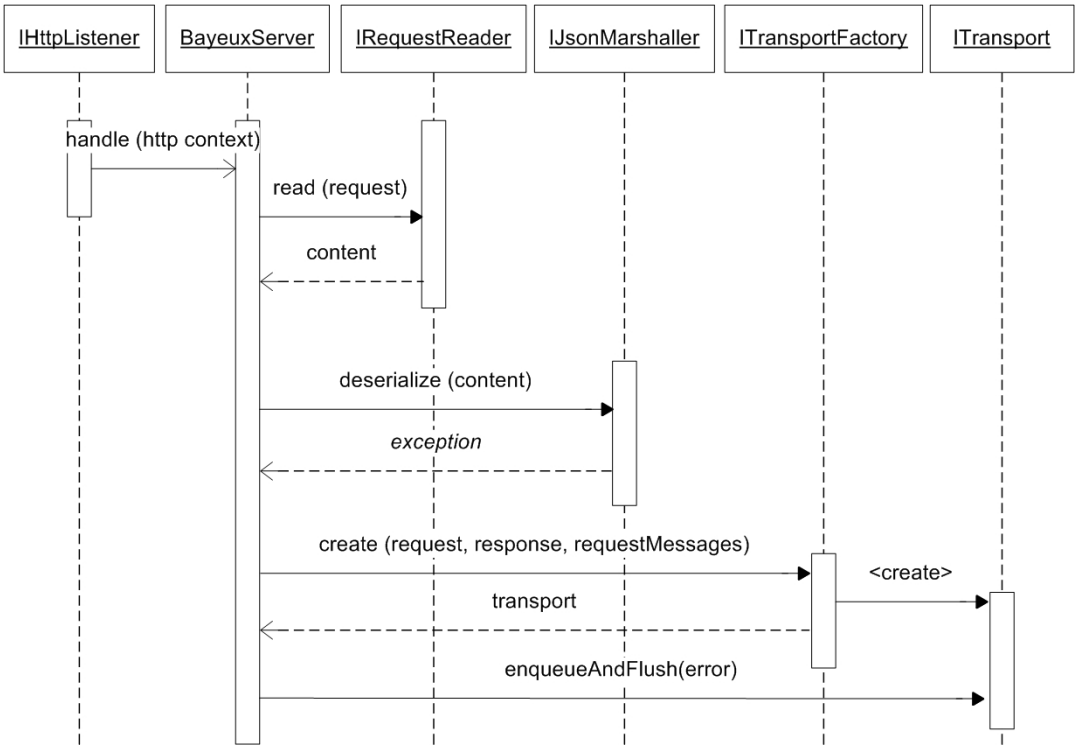

Figure 3.12: A sample message flow in Comet.NET (error case)

3.6 Transport types

The following two sections discuss transport types supported by Comet.NET and present various details on their implementation.

3.6.1 Supported transport types

Comet.NET supports the two widely accepted Comet communication styles: long polling and streaming. These styles define communication details such as sequence of connections and the content that is transported over them, but leave a lot of implementation-level details open. For example, long polling defines that the server should attempt to keep each request open until events arrive or, otherwise, a timeout occurs, but does not specify details on the format of the request and response. Similarly, streaming defines that the response should be kept open and used for delivering of events as they occur, but leaves even more implementation details open than long polling.

As defined in the Bayeux protocol, there are two transport types for long polling: one for communication in the same domain and one for communication between different domains (cross-domain). Same-domain long polling is called "long-polling" in the protocol specification, whereas cross-domain long polling is called "callback-polling". The semantics of these two transport types is essentially the same, but they differ in the HTTP method (POST vs. GET), encoding of messages and the way a client handles responses. Comet.NET supports both regular and cross-domain long polling transport types.

A streaming transport type was part of early versions of the Bayeux protocol, but was later removed and is not contained in the final version of the specification. However, the protocol allows server and client implementations to define custom transport types. Comet.NET uses this possibility and defines three streaming transport types: "hidden-iframe", "xhr-streaming" and "forever-response". These transport types represent the streaming techniques presented in Section 2.1.2.

3.6.2 Implementation details

The concept of *transport* was introduced and briefly described in 3.4.4. This section gives more insight into implementation details of transport-related interfaces and classes.

The interface `ITransport` defines the contract for enqueuing and flushing of `IMessage` objects and represents the main point of interaction with the underlying communication protocol (in the current version only HTTP). ITransport is an abstraction that other parts of the system can use to send messages without dealing with low-level communication details. The interface is designed to support both the polling and the streaming communication type. It also supports both synchronous and asynchronous flushing of data to remote clients.

A careful analysis of transport types Comet.NET should support was done prior to and during the early stages of development. The analysis included not only properties of transport types but also typical interaction scenarios between other parts of the library and transports. It turned out that there are only two differences between transports: the initialization and the way data is encapsulated into the underlying response. This led to the class hierarchy presented in Figure 3.13.

There is an abstract base class `BaseTransport` that contains most of the logic needed by all supported transport types. The class implements all methods and properties defined by `ITransport` and has only one abstract method: `GetResponseBytes`. The method accepts a list of `IMessage` objects and returns a byte array that can be written directly to the underlying

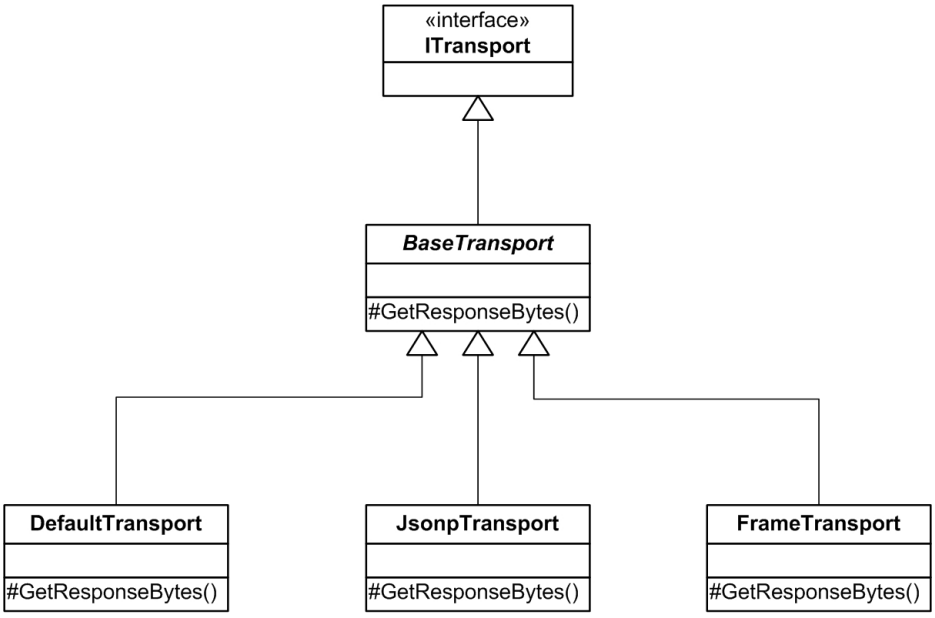

Figure 3.13: Transport implementations in Comet.NET

response. By leaving the implementation of the method open we allow subclasses to hook themselves easily into the process of serialization and sending of messages, while still keeping the logic encapsulated in one place and enforcing the single responsibility principle[68].

`DefaultTransport` has no initialization other than standard initialization sequence provided by `BaseTransport` and encodes messages with UTF-8[69]. This implementation is used for "long-polling" and "forever-response" transport types. `JsonpTransport` also does not have any additional initialization and encodes messages with UTF-8. But unlike `DefaultTransport`, it encapsulates serialized data into a Javascript callback code, as required by "callback-polling" transport type. `FrameTransport` is the implementation used for "hidden-iframe" transport type. It has a rather complex initialization, because it has to prepare the iframe opening HTML content and send it along with the first message. As with other transport implementations, messages are encoded as UTF-8, but in this case they have to be embedded in script tags with appropriate Javascript callback. `XHRTransport` is very similar to `FrameTransport`, but instead of wrapping messages into script tags, it injects a delimiter between them so that the client can identify them during parsing. Messages are encoded with UTF-8 encoding.

As specified in the requirement in Section 3.1.1, transport implementations in Comet.NET offer both models, the synchronous and the asynchronous model, for sending data to remote clients. The synchronous model is appropriate if the application should block while waiting for data to be serialized and sent. This is usually not the case in high-performance applications. In the more complex asynchronous model, the serialization and sending of the data is executed in a separate thread and the application does not block. This model is intended for high-performance

scenarios.

BaseTransport uses the Event-based Asynchronous Pattern[70] to provide asynchronous versions of methods for flushing (`FlushAsync`) and flushing and closing (`FlushAndCloseAsync`). The underlying HTTP library, Windows HTTP.SYS, uses a highly optimized mechanism for asynchronous I/O operations called Windows I/O completion ports[71, 72].

3.7 Thread management

The Section 3.7.1 discusses the traffic profile of Comet applications and how it affects the thread management in a Comet server. The Section 3.7.2 presents how Comet.NET deals with the thread management requirements posed by the Comet application model and provides insight into implementation details.

3.7.1 Comet application model

As noted in Section 3.1.1, one of the key aspects of a scalable Comet infrastructure is the efficient handling of HTTP connections.

The most commonly used request processing model in traditional multi-threaded Web servers is the thread-per-request model[73, 74]. There are numerous variations, but the concept is the same: each incoming request is associated with a thread. This thread is used to perform all the necessary work to service the request and send the response back to the client. After the thread is done serving the request, it is assigned to another incoming request or returned to the thread pool if there are no pending requests. By using multiple threads, each bound to one request, the server can process many concurrent requests simultaneously, resulting in high throughput and minimizing the number of requests pending at any given time. This approach works great for serving a large number of rather short-lived requests - rendering dynamic Web pages, for instance - and is commonly used with Web application frameworks such as ASP.NET, PHP and Ruby on Rails.

Comet applications have a radically different traffic profile than traditional Web applications. In a standard Web application, a request means that a client wishes to retrieve a resource or invoke a service that will perform some task. The server performs work, processes the request and sends back the response as quickly as possible. However, a long polling or a streaming request from a Comet client does not cause the server to perform work. It spends most of its life-cycle in an idle state, waiting for an event to occur.

The Comet application model is based on creating and maintaining long-lived HTTP connections with outstanding requests. These requests are used by the server to send events to the client as they occur. Typically, each connected client will have an outstanding request most of the time, so a Comet server needs to deal with as many concurrent requests as it has clients. The traditional thread-per-request model degrades to a thread-per-client model, which does not perform well with an increasing number of concurrent clients. Since threads consume considerable amount of resources, the thread-per-request model is generally unable to scale to a large number of Comet clients.

In order to efficiently implement the key aspect of a Comet server - handling of long-living HTTP connections - a non-standard, asynchronous and non-blocking approach for processing of HTTP traffic is needed [75]. The basic premise of this approach is the separation between threads and requests. In this model, a thread is not bound to a single request during the whole life-cycle of the request, but can be assigned to another request if the one that is being processed enters the "waiting" state. Requests in the waiting state are the ones that are "parked" on the server and used for sending events back to the client. Once a request enters the waiting state, the thread that was bound to it is released and waiting is done completely asynchronously, without any resources being held. When an event occurs that should be sent using the request, it can be done from any available thread.

3.7.2 Implementation details

Unlike Java servlet containers, Comet.NET does not have the burden of the Java servlet model, which has an inherent thread-per-request design[55]. The underlying HTTP engine, Windows HTTP.SYS, does not implicitly bind threads to requests, so there were no architectural limitations when implementing the asynchronous and non-blocking processing of HTTP traffic in Comet.NET.

Incoming requests are received by an implementation of `IHttpListener` and are delegated to `BayeuxServer`. At this point, each request is associated with a thread. The server coordinates creation of appropriate `ITransport`, deserialization and inspection of `IMessages` as well as their delegation to appropriate instances of `IMessageHandler` or `ILocalService`. This work is done by the thread associated with the request. After the processing of the request is completed, the server needs to decide if it should send the response immediately or if the request should be parked and used for sending events to a client at some later point in time. This is done by inspecting the transport associated to the request: if it contains useful response data, the response is sent to the client and the request-response cycle is terminated; otherwise the transport is kept open and attached to `BayeuxClient` so that it can be used for dispatching events. In either case, the thread is freed after this last step and can be used for processing another incoming request.

Typically, requests that contain only *connect* Bayeux messages are kept open and are used for asynchronous sending of events to the client. Other meta messages defined in the protocol are used to control the state of the client on the server and require that the response is sent immediately. Control messages include subscribe, unsubscribe, disconnect and publish messages. If a request contains one of the control messages, it can not be used for asynchronously sending events.

Once the waiting request, encapsulated by an instance of `ITransport`, is attached to `BayeuxClient`, it is the responsibility of the application on top of Comet.NET to enqueue and flush events. How and when events are dispatched to remote clients depends largely on the type of application and its event source(s). The library hides low-level details such as ITransport and offers a rich public API for sending events to remote clients, but leaves the actual logic to be implemented in the application layer.

3.8 Security

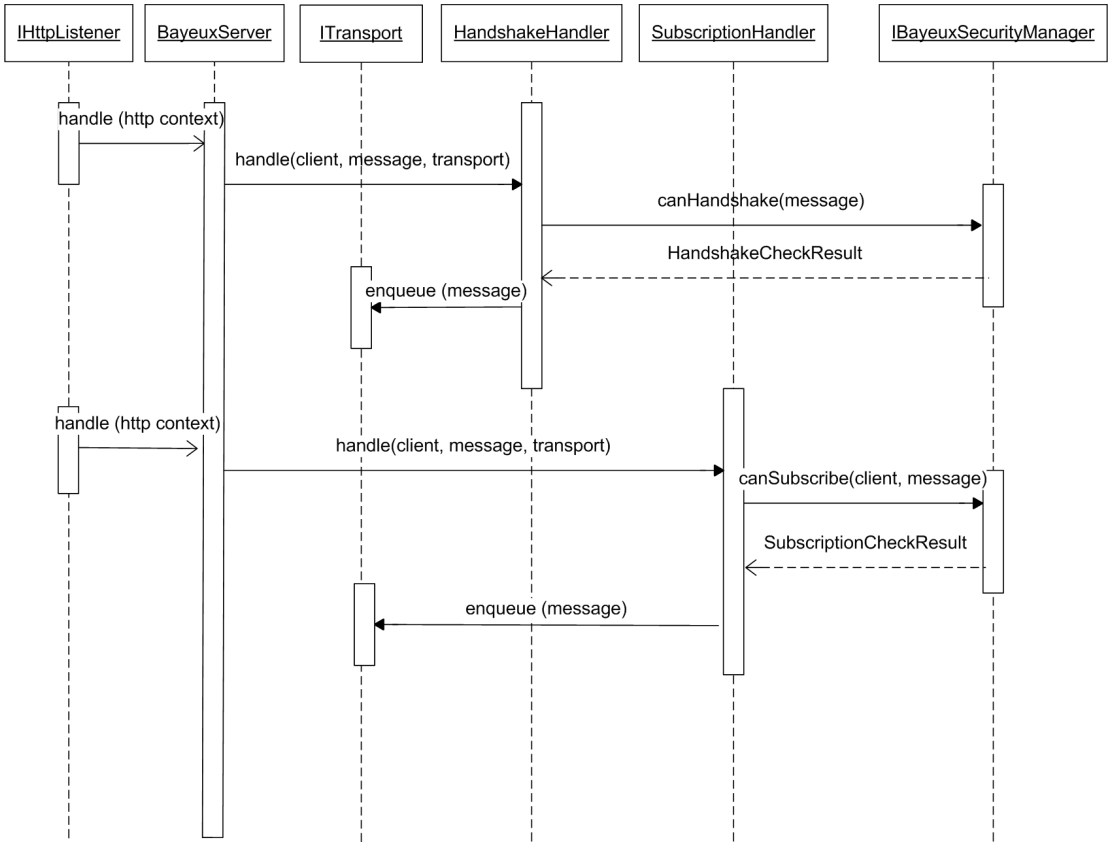

Figure 3.14: Sample security checks in Comet.NET

Since the Bayeux protocol does not define mechanisms for authentication or authorization, it was not possible to provide an interoperable implementation of these security aspects in Comet.NET. Instead, the library defines three interception points that can be used by application-level security managers to implement custom authentication and authorization on top of the Bayeux protocol. These are the available interception points:

- OnHandshake: executed when a handshake message is received from a remote client
- OnSubscribe: executed when a subscribe message is received from a remote client
- OnPublish: executed when a publish message is received from a remote client

Interception points are represented as methods in an interface, `IBayeuxSecurityManager`, whose implementation can be injected into `BayeuxServer` upon creation. A sample interaction between the involved components is shown in Figure 3.14.

Handshake When a client issues a handshake request, the security manager is provided with the Bayeux message as it was received from the client. At this point, there is no client state at the server, so an instance of `BayeuxClient` cannot be provided. The security manager can inspect the message and decide if the client should be allowed to establish a connection with the server. If the connection is rejected, the security manager can provide an error code and human readable error description, which are sent in the "error" field of the response message.

Subscription On each subscription request from a client, the security manager is provided with the server-side representation of the client - an instance of the `BayeuxClient` class - and the Bayeux message as it was received from the client. The message contains the name of the channel the client wants to subscribe and may also contain extended data in the "ext" field. With provided data, the security manager can execute a custom authorization check and decide if the client should be allowed to subscribe to the channel. Like with handshake, in case of rejection, an error code and description can be returned.

Publish Usually, the application wants to restrict publishing of events from remote clients in some way. This interception point is executed every time a publish request is received from a client and enables the security manager to control who can publish to channels and what can be published. The security manager is provided with the `BayeuxClient` and `IMessage` as it was received from the client and can use this data to decide if the message should be published. It can even change the message, for example remove unauthorized parts from the "data" field. In case publishing was disallowed, an error code and description can be returned and will be sent to a remote client.

CHAPTER 4

Sample applications

This chapter presents two sample applications based on Comet.NET. The first section presents a chat application. The second section provides a detailed overview of the TeleTrader HTTP Push Service, an enterprise stock market ticker application built on top of the Comet.NET library.

4.1 Chat

We present a simple chat application as a “Hello world” example of a real-time interaction via a Comet server. A typical Web-based chat application[43][44] features many users that communicate with each other by exchanging messages in chat rooms. Users publish messages to chat rooms and receive messages from other members of those chat rooms. Private communication between two users is also a standard feature of such chat applications. In this case, messages are only exchanged between the two users involved in private conversation.

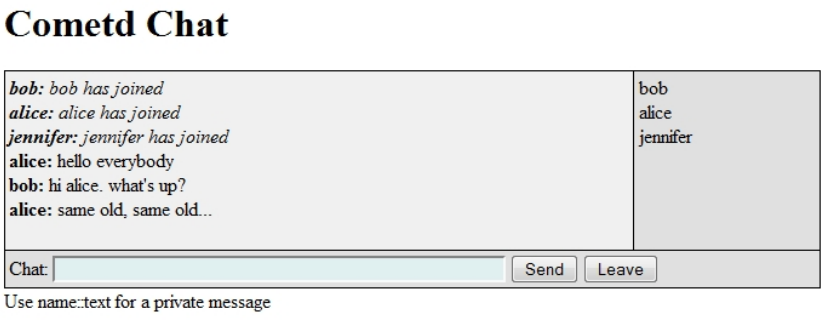

Figure 4.1: CometD chat sample application

A simple messaging application will be used to demonstrate the capabilities of Comet.NET. We will use the existing chat application delivered as a sample with CometD Jetty as basis and adapt it to work with Comet.NET. Since the library is fully Bayeux-compliant, there is no need

for changes in the client part of the application. The sample chat website, including client-side logic contained in the supporting Javascript code, will be reused "as is". The server part will be completely rewritten, but will retain the same input/output behavior.

The following sections contain a list of requirements posed on the application, a short overview of implementation specifics and a subsequent comparison of the implementation to the reference implementation included with Jetty.

4.1.1 Requirements

The chat application presented in this chapter is a simple "Hello world" example and as such has only basic requirements. The purpose of this example is to demonstrate how two-way communication between remote clients in a typical Comet scenario can be achieved with Comet.NET.

The sample application fulfills the following requirements:

- Single chat room

  There is only one public chat room and every connected user automatically joins the room.

- Public conversation

  Every connected user can publish messages to the public chat room. There is no authorization. Also, every connected user receives every message that is published in the public chat room.

- Private conversation

  Two users can engage in a private conversation, in which case messages are routed only between them.

- No authentication

  Everyone can use the chat application. No credentials are required.

4.1.2 Design

The sample chat application is modelled as a client/server, event-based application that produces events and reacts on them. The general architecture of the application is shown on Figure 4.2. The client-side part of the application is a mixture of static HTML, images and style sheets and a dynamic Javascript code that encapsulates the logic. The static content is served by a Web server using the standard HTTP requests. The dynamic part is built on top of the CometD Javascript client and communicates with the Comet server using the Bayeux protocol. The Comet server essentially consists of Comet.NET and the server-side application implementation that is encapsulated in the *ChatService* class.

All events in the system are produced by users (client-side). There are no server-side generated events. Events originating from clients are sent to the server where they are processed and forwarded to other users. The events are: a user joining or leaving the chat, sending of a message to the public chat and sending of a private message to another user.

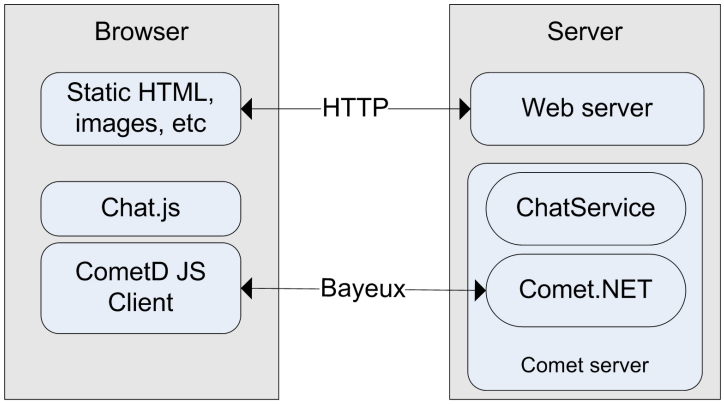

Figure 4.2: Architecture of the chat application

When a user joins or leaves the chat, his client publishes a message on */chat/demo* with the user name and a flag stating if he joins or leaves. This is a broadcast message that needs to be routed to all active clients and is used to display appropriate messages on the screen. In addition to that, the client expects the full list of names of active users every time somebody joins or leaves and uses the list to refresh the GUI. This implies that the server side of the application needs to keep the list of active users and their properties. Instead of sending the full list of user names on each change, an incremental approach would be more efficient, but it would also make both client and server implementations more complicated. Since we aim for full compatibility with the existing client implementation and for the sake of simplicity, no optimizations are introduced.

When a user writes something into the public chat room, his client publishes a message on */chat/demo* with the text that was entered by the user. Again, this is a broadcast message that needs to be routed to all active clients. For private messaging, the channel */service/privatechat* is used. When a user sends a private message to another user, the client publishes a message on */service/privatechat* with the following data: name of the sender, name of the receiver and the text entered by the sender. This message must not be broadcasted to all clients. It is only sent to the client that was addressed as receiver in the message.

4.1.3 Implementation

Local service In order to implement the server-side part of the chat application, we need to write an implementation of `ILocalService` that contains the logic for coordination and notification of chat clients. The implementation resides in the class named `ChatService`. The application uses two channels - */chat/demo* for public and */service/privatechat* for private communication - so the service will have to receive and publish messages on these two channels. This is easily achieved in Comet.NET by registering the service with `BayeuxServer` for the two channels. This instructs the `BayeuxServer` to forward all messages received on specified channels to the provided local service. And since relevant context objects are provided with each

forwarded message, the service can easily send messages to remote clients. The following code snippet demonstrates how to initialize the server-side part of the application:

```
BayeuxServer bayeuxServer = new BayeuxServer(settings, httpListener,
    jsonMarshaller);
ChatService chatService = new ChatService(bayeuxServer);
bayeuxServer.RegisterLocalService("/chat/demo", chatService);
bayeuxServer.RegisterLocalService("/service/privatechat", chatService);
bayeuxServer.Start();
```

Event types and deserialization For transporting event data, the Bayeux protocol provides the "data" field within a message[50]. The chat application uses a range of simple JSON objects to represent different events and their data. For example, the event of a new user joining the chat is represented by a JSON object with the field "chat", which contains a simple message to be displayed on the screen, and the flag "join" set to true. A user leaving the chat is represented by a similar message that has the same field "chat" and a flag "leave" set to true. There are four distinct event types and therefore four JSON objects that represent them. However, on the server side, we define only one entity class named Data that defines all possible fields and is used as representation of all events. The main reason for this design decision is the simplicity of the implementation.

The application needs to provide an implementation of `IDataDeserializer` that can create an object from JSON content in the "data" field. The actual deserialization is delegated to the JSON.NET library[76], so the data deserialization logic for the chat application consists of two lines of code that invoke appropriate JSON.NET functions:

```
public class DataSerializer : IDataDeserializer
{
  private Newtonsoft.Json.JsonSerializer serializer =
                                        new Newtonsoft.Json.JsonSerializer();

  public object Deserialize(string json)
  {
          StringReader stringReader = new StringReader(json);
          return serializer.Deserialize(stringReader, typeof (Data));
  }
}
```

Event processing Routing of broadcast messages published by chat clients is done by the Comet.NET engine itself, so the ChatService has only two responsibilities: delivering a user list on each change (join and leave) and handling of private communication.

In order to be able to deliver the user list to clients, the ChatService has to keep a local list of active users. It reacts on each message received on the channel beginning with "/chat/" and adds or removes the user if the message contains the flag "join" or "leave", respectively. Even though the application currently supports only one chat room, the client and its event data are designed to support multiple chat rooms in the future, so on the server side, we opted for a data

structure that takes multiple chat rooms into account. The data structure is a nested dictionary that maintains key-value pairs and allows keeping the list of mappings between client ID and user name for every room (Figure 4.3). When a user joins a room (currently only “demo” chat room is supported), its client ID and user name are added as a key-value pair into the dictionary for that room. When a user leaves the room, it is removed from the dictionary for that room. On each of these two events, the service delivers the full list of user names as an array of strings to all active clients.

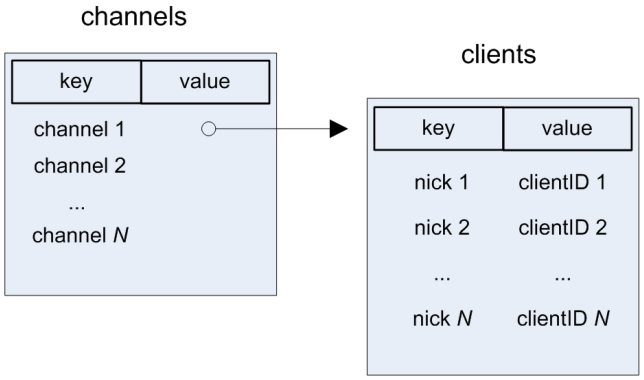

Figure 4.3: Chat service data structure

Handling of private communication requires the ChatService to keep a list of all clients along with their user names. When a user sends a private message to another user, the service needs to find the appropriate instance of `BayeuxClient` and deliver the message. For this purpose, a mapping between user names as keys and `BayeuxClient` instances as values is sufficient. The mapping is updated every time a new client is connected or an existing client is disconnected. This is done by handling `ClientAdded` and `ClientRemoved` events of `BayeuxServer`. The following code snipet shows how private messages are sent:

```
// Data data = received from the sending client
// BayeuxClient receiver = the receiving client
// BayeuxClient source = the sending client

Data chatMsgData = new Data();
chatMsgData.Chat = data.Chat;
chatMsgData.User = data.User;
chatMsgData.Scope = "private";
receiver.Deliver(channel.ChannelId, chatMsgData);
receiver.Flush();

if (receiver != source)
{
        source.Deliver(channel.ChannelId, chatMsgData);
        source.Flush();
}
```

4.1.4 Discussion

By re-implementing the server-side of the CometD sample chat application with Comet.NET, we demonstrated the basic capabilities of the library and showed how Bayeux services can be built on top of it. While this application is too simple to be used as basis for functional comparison between Comet.NET and the Jetty CometD reference implementation, it does show some commonalities and differences between the two.

The main difference is the way services are defined and attached to the Bayeux server. With Jetty, Bayeux services are essentially servlets that are instantiated and initialized by the container. They are self-contained and have little knowledge of the outside world. Most of the instantiation and configuration work is handled by the container. On initialization, Bayeux services are injected with an instance of *Bayeux* - the representation of the Bayeux engine in Jetty - and can use it to register themselves for channels, query clients, channels and subscriptions and interact with the outer world.

On the other hand, Comet.NET is designed as a hosted Comet engine that has to be instantiated and configured by the host application. This poses a certain overhead when compared to Jetty, but also offers more flexibility. Once the engine is configured and started, it can be used by the application in two ways: either by registering local services that are bound to one or more channels or by handling events raised by BayeuxServer and interacting with clients, channels and subscriptions.

We can conclude that Comet.NET poses more configuration overhead and requires more boilerplate code than Jetty. In case of simple services like the one presented in this chapter, this is very noticeable and doubles the amount of code needed for Comet.NET. However, if we compare the services themselves, we notice that there is little difference. Both services follow the same logical layout and are implemented in a very similar way. There is no substantial difference even in terms of lines of code: Jetty service has 90 lines of code [1] and Comet.NET service has 115 lines of code.

4.2 TeleTrader HTTP Push Service

To illustrate features of Comet.NET, we present the *TeleTrader HTTP Push Service*, an enterprise stock market ticker application built on top of the Comet.NET library. Teletrader HTTP Push Service[77][78] is a streaming engine designed for real-time data delivery over HTTP connections. It targets primarily Web clients and is optimized for streaming market data directly to Web browsers, but can be used to push data to virtually any client that supports HTTP.

The service is a sophisticated router between TeleTrader Market Data Server (MDS)[79] that acts as data source and remote clients that have the role of data consumers. It is a data-centric, asymmetric Comet application with one server-side component responsible for generation of events and many remote clients that do not generate events themselves, but only consume them.

[1] CometD Java samples: ChatService. `http://svn.cometd.com/trunk/cometd-java/cometd-java-examples/src/main/java/org/cometd/examples/ChatService.java`, 02.01.11

This setup is standard in financial industry where the most widely adopted application of Comet is distribution of market data via the Web.

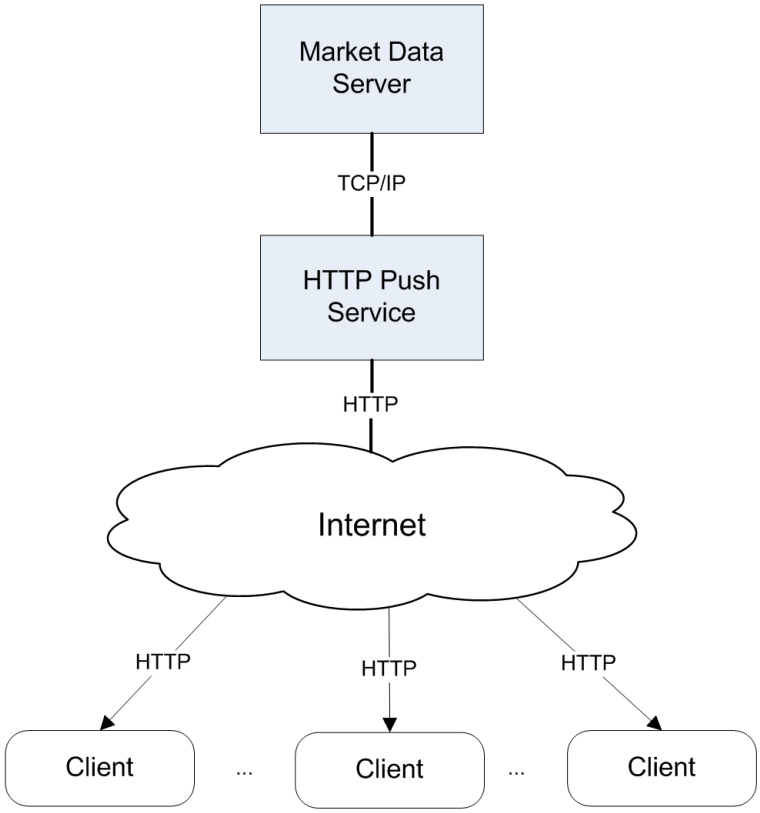

Figure 4.4: General setup of the HTTP Push Service

4.2.1 TeleTrader Market Data Server

TeleTrader Software AG is both a data re-vendor and an IT service provider specializing in stock market data. In order to be able to deliver financial data to different customers, ranging from private investors using desktop trading terminals to banks and other financial institutions that use TeleTrader solutions for both their Web presence as well as their back office needs, the company has developed a complex IT infrastructure that is able to process and offer millions of trades and quotes ('tick data') from a vast universe of market data to its customers, along with master data about the securities (investment instruments such as stocks, bonds, futures, etc.).

The Market Data Server (MDS)[79] makes up the core of the entire TeleTrader service infrastructure. It is a proprietary system developed by TeleTrader for the specific purpose of storing and serving millions of 'ticks' (intraday and interday quote data) for millions of securities while being fast, reliable, and scalable. The system is centered around the notion of a symbol, which represents one security. Symbols have two different kinds of data: master and quote data.

Master data is kept in a relational database whereas quote data is kept in several proprietary file structures optimized for this purpose. From the client perspective, both master and quote data for a symbol are represented by a range of strongly-typed fields with values.

In addition to market data, MDS stores and serves financial news articles published by a variety of news feeds. As with symbols, news data is represented as a range of strongly-typed fields with their respective values.

Almost all quote data served via any of TeleTrader's products or interfaces originates from MDS. There are client APIs for several platforms, most notably C++ and .NET/C#, that can be used for retrieval of static and dynamic data. Both the pull and the push paradigms are supported. Dynamic data, such as symbol quote data or news articles, can be retrieved in pull and push manner. Static data, such as master data of a symbol, can be retrieved only by requesting it (pulling).

4.2.2 General setup

The HTTP Push Service can be described as a sophisticated router between MDS that acts as data source and remote clients that have the role of data consumers. In essence, it forwards updates from MDS to remote clients in real-time. The relationship between the HTTP Push Service, MDS and clients is depicted on Figure 4.4.

The Bayeux protocol is used as transport mechanism, on top of which a simple application level protocol is defined. This protocol specifies how clients can issue requests to the server and the format of messages exchanged by the server and the client. The application level protocol used by the HTTP Push Service is described in section 4.2.4.

The service supports delivery of two distinct types of data: symbol quote data and news articles. In both cases, clients need to express their interest in a resource (symbol or news source) by subscribing it. With each subscription, a list of fields can be supplied. This allows very fine-grained filtering of data that is delivered to the client. After successful subscription, the client is notified every time a change of one or more fields of the subscribed resource occurs. Every subscription can be deactivated, which causes the server to stop sending updates of the resource to the client.

In order to be able to distribute changes in the state of a resource, the service itself needs to be notified about these changes by MDS. The service uses a subscription-based streaming API to communicate with MDS and receive updates for resources of interest in real-time. Resources are subscribed on-demand, so only data from resources that were actually requested by clients is received and processed. When a client subscribes a resource that is currently not subscribed by any other client, the service activates the same subscription on MDS and starts receiving updates for the resource. Subsequent subscriptions for the same resource by other clients cause no action towards MDS, because everything is already in place. When the last client cancels the subscription for a resource, the subscription for the resource on MDS is deactivated and changes are no longer received. This on-demand approach is applied not only to resources, but also to their fields. The service always subscribes the superset of fields needed by clients interested in a specific resource. When a client subscribes or unsubscribes, the set of required fields is re-evaluated and the MDS subscription is updated accordingly. This approach makes sure that no unnecessary data is ever received from MDS, thus conserving bandwidth and CPU usage.

The described data flow from MDS through the HTTP Push Service to remote clients is shown on Figure 4.5. The figure also gives a high-level overview of the system architecture.

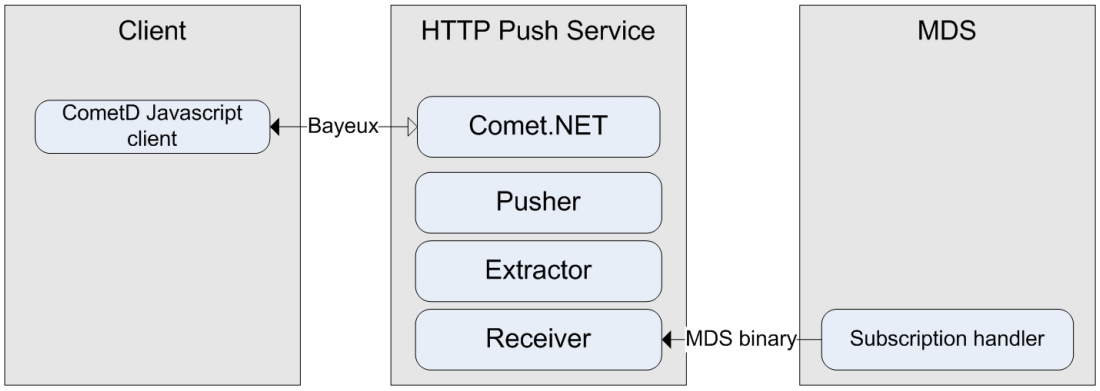

Figure 4.5: High-level architecture of the HTTP Push Service

With exception of control requests such as subscribe and unsubscribe requests, the communication between the server and the clients is one-way only: from server to the clients. Clients are passive consumers that do not publish events themselves, but only consume event messages received from the server. This is a typical example of an asymmetrical publish/subscribe architecture, where a very small number of publishers (in this case, there is only one) produce events that are dispatched to a rather large number of subscribers.

Since the HTTP Push Service is a commercial product and the business model behind it is based on selling market data in different qualities, one important requirement was to ensure that only authenticated clients are able to connect to the server and receive only market data they are authorized for. Bayeux is a plain-text protocol and it does not specify any security mechanisms, so the protocol alone cannot be used to enforce security policies needed by the HTTP Push Service. Instead, the concept of *security provider* was introduced. The security provider is essentially a separate component that allows remote clients to authenticate. It also communicates with the HTTP Push Service behind the scenes, offering authentication and authorization services. The security concept of the HTTP Push Service is described in more detail in section 4.2.6.

4.2.3 Channels

Since the Bayeux protocol uses channels as addressing mechanism, all resources have to be mapped to channels. Both resource types, symbols and news sources, have a unique numerical identification, so their mapping to channels was a trivial task. Symbol ID (tts-id) is used for identification of symbols and News Source ID is used for identification of news sources. This leads to the following format of the channels:

- Symbol channels: /teletrader/symbols/<*tts-id*>
- News channels: /teletrader/news/<*source-id*>

In addition to resource channels, the HTTP Push Service defines several service channels. Service channels allow direct, point-to-point communication between the server and the client and are used for delivery of application-level notifications. Market data is never delivered via these channels.

One example of service channel is the disconnect channel (/service/disconnect). The "disconnect" channel is used for sending a notification prior to server-initiated disconnect of a client. Clients can use messages delivered on this channel to react on pending disconnect and start cleanup procedures or display an appropriate message to end-users.

4.2.4 Message format

The Bayeux protocol defines the "ext" field that can be used for application-specific extensions of the protocol. This field is used by HTTP Push Service for transporting additional data on handshake and subscription requests. This section describes the format of extension fields for these two operations and the format of event messages sent by the service.

Handshake During the handshake, a client needs to provide a valid authentication token, otherwise the connection is refused by the server. The client can also provide a list of default symbol and news fields that should be used for subscriptions when no fields are explicitly set. This is a convenient way to define the default field sets for the lifetime of a client. The following table contains the defined fields along with short description for each of them.

Field	Required	Description
AuthToken	Yes	The authentication token provided by the security provider (TTWS)
SymbolFIDs	No	A comma-separated list of field IDs that will be used for subscriptions of symbols if no fields are supplied.
NewsFIDs	No	A comma-separated list of field IDs that will be used for subscription of news if no fields are supplied.

Table 4.1: Handshake extension fields

Field	Required	Description
FIDs	No	A comma-separated list of field IDs. Only updates to these fields will be sent to client.
PushType	No	Indicates how updates will be transmitted to the client. Valid values: [*Snapshot* | *Everything*]

Table 4.2: Subscription extension fields

Subscription When subscribing to a channel, clients can specify what data they wish to receive by providing filters for the subscription. For both symbol and news subscriptions, a

comma-separated list of valid fields can be sent with the request, restricting the subscription to only specified fields and causing the server to send update notifications only when one of these fields changes. Clients also have the possibility to specify how they would like to receive updates from the server: only the latest change per field or all changes. This feature is called PushType and is further described in section 4.2.5.

Event notification As defined by the protocol, event messages have a "data" field that contains the payload - the event data. In case of the HTTP Push Service, this field is a collection of key-value pairs, where each key-value pair represents field name and its value. The value can be simple or complex, depending on the field. The format of "data" field is generally the same for both symbol and news. The only difference is in fields that can be contained in messages.

Each symbol update message contains a special "symbolId" key with Teletrader Symbol ID (tts-id) of the symbol as value. This field can be used to correlate messages with symbols on the client. All other fields are optional and depend on subscription filters, customer-specific permissions and data available in MDS.

Each news update message contains data for one article. Unlike symbol updates, this type of message does not have any special fields: its content is determined only by the field list provided on subscription and client's permissions. It is however recommended to subscribe "sourceId" and "articleId" fields which can be used for client-side correlation.

Figure 4.6 shows a sample event message with symbol quote data in JSON notation.

```
{
        "channel" : "teletrader/symbols/12345678",
        "id" : 55,
        "data" : {
                        "symbolId" : "12345678",
                        "last" : 14.45,
                        "dateTime" : "12.01.2010 11:18:33",
                        "volume" : 4450,
                        "turnoverValue" : 653703,
                        "change" : 0.32,
                        "numberTrades" : 69
                }
}
```

Figure 4.6: Sample event message with symbol quote data

4.2.5 Design and implementation

There are two primary goals of the HTTP Push Service: managing remote Bayeux clients and their subscriptions and broadcasting data received from MDS according to those subscriptions.

Domain model As previously stated, the HTTP Push Service defines an application-level protocol on top of Bayeux that allows clients to provide additional information such as authentication tokens or subscription filters. Besides these protocol extensions, there is state information

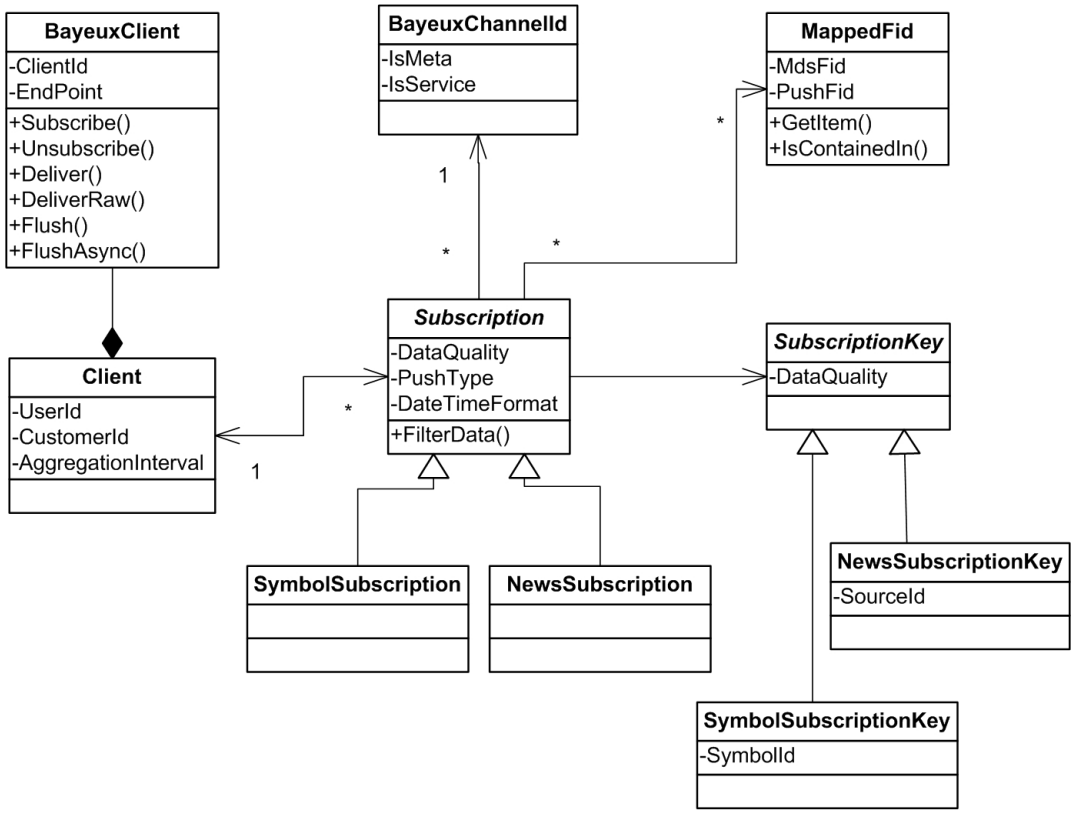

Figure 4.7: HTTP Push Service domain model

beyond the default Bayeux state that needs to be kept during the lifetime of a client. This is a typical problem that all but the most basic applications based on Comet.NET have: how to store and handle application-specific state data. The solution we opted for in HTTP Push Service was to create new classes that represent the domain model of the application. The model itself is very similar to the one found in Comet.NET and consists of classes representing clients and subscriptions, but adds application-specific properties to them.

Remote clients are represented by instances of the class `Client`. Each client has a reference to a corresponding instance of `BayeuxClient`, configuration and security-related data such as the customer the client belongs to, as well as the list of active subscriptions. Clients are created each time `BayeuxServer` notifies the application about a newly connected client and are then added to the local client list. Similarly, when the server fires an event stating that a client was disconnected, the corresponding `Client` instance is removed from the local list.

Subscriptions are represented by instances of classes `SymbolSubscription` and `NewsSubscription`. Because quote and news data is offered in different data qualities, a subscription is not a simple mapping to a resource (represented by a channel) but rather a combination of resource and data quality. This combination is represented by `SubscriptionKey` and its concrete subclasses - `SymbolSubscriptionKey` and `NewsSubscriptionKey`. All sub-

scriptions to the same resource in the same data quality have the same key. A subscription is uniquely identified by its key and by the client it belongs to.

Another important property of subscription is the list of fields that specify what kind of data clients wish to receive. The field list is the basis for fine-grained filtering of data prior to delivering it to the client. It is used to reduce full update messages received from the backend to only those parts that were requested by the client.

Fine-grained filtering is one of the main reasons for introduction of subscription keys as means of partial subscription identification. When an update is received from MDS, the application has to find all subscriptions for the updated resource in the quality in which the data was received from MDS. Found subscriptions are then used to filter the received message so that only relevant parts are published to clients. This lookup is executed each time an update is received from the backend, so high performance is the primary requirement.

Such high-performance access is provided by `SubscriptionStore`, a custom data structure designed for managing subscription objects during their lifetime. The data structure provides operations for retrieval of both single subscription objects and lists of related subscriptions (subscriptions with the same key), as well as standard add and remove operations. The focus when designing and implementing SubscriptionStore was on fast lookup operations by subscription keys, because they are expected to be executed far more often then add and remove operations. This was achieved by using a custom concurrent dictionary with `SubscriptionKey` objects as keys and custom concurrent lists of `Subscription` objects as values.

As with clients, subscriptions are created when `BayeuxServer` notifies the application about a new subscription and are then added to `SubscriptionStore`. When the application is notified about unsubscription, the corresponding Subscription object is removed from the store. “Dangling” subscriptions are removed in case a client disconnects without unsubscribing, which guarantees that the `SubscriptionStore` is always in a consistent state.

Data processing and forwarding The core functionality of the HTTP Push Service can simply be described as forwarding updates from MDS to remote Bayeux clients. The process consists of receiving messages from MDS, extracting data from them, converting it into the format expected by remote clients and finally publishing event messages to clients according to parameters specified in their subscriptions. There are three components involved in the process, defined by the following interfaces: `IDataReceiver`, `IDataExtractor` and `IDataPusher`. `Subscription` objects are used for final data filtering and the actual delivery to remote clients is delegated to Comet.NET through instances of `BayeuxClient`. Figure 4.8 shows the relationships between involved components.

`DataReceiver` is responsible for communication with MDS. It activates, updates and deactivates subscriptions towards MDS as needed by the application and receives messages with updates. The receiver is associated with exactly one connection to MDS and receives data in one quality. Multiple data quality levels are achieved by having multiple instances of this class. In the current setup there are only two data quality levels, but this approach allows adding new levels with no architectural changes.

The MDS API provides a client interface, represented by `IMdsAsyncClient` in Figure 4.8, that allows applications to asynchronously exchange messages with MDS. Most low-

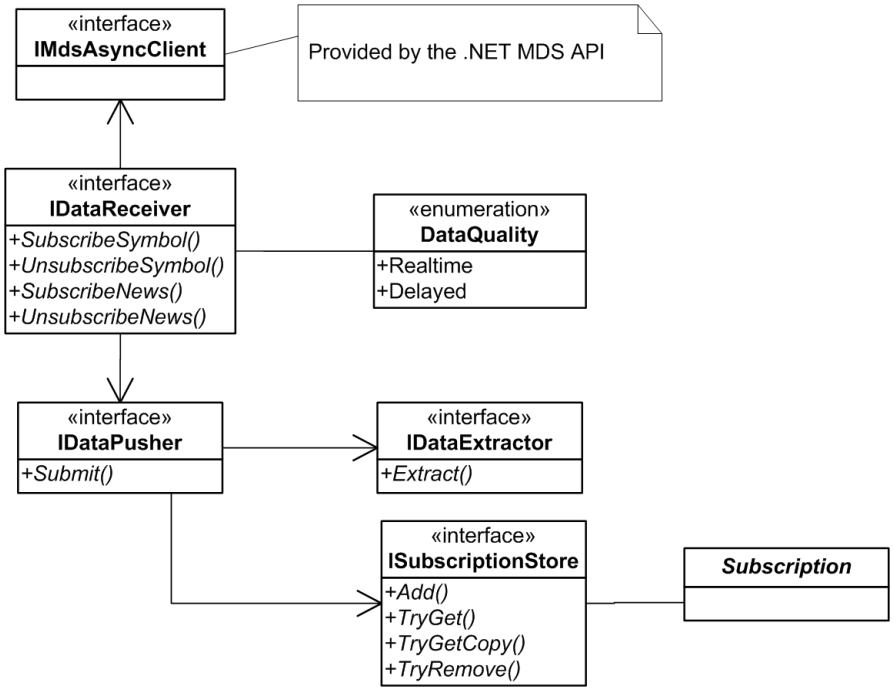

Figure 4.8: Components involved in data forwarding

level communication details are hidden inside the API implementation, so `DataReceiver` only needs to issue proper subscription and unsubscription requests and react on new messages. Messages received from MDS are represented by instances of the class `MdsMessage`, whose structure is of little relevance in this context. On each new message, the data receiver does some internal processing after which the message is submitted to `DataPusher`, along with the data quality associated with the receiver.

`DataPusher` is the component responsible for processing of `MdsMessage`s and publishing update events to remote clients. This is an active object that has one or more dedicated worker threads for the actual processing and delivery of data. This way, the input and output, represented by data receiver and data pusher respectively, are not only decoupled in terms of interfaces but also in terms of execution. The receiving of updates from MDS and their processing and publishing to remote clients executes independently and in parallel, if supported by the underlying physical machine (if multiple logical processors are available). After experimenting with several different setups, we have come to the conclusion that the best performance is achieved if data pusher has one worker thread per logical processor.

The first step in processing of incoming messages is their conversion from the format provided by the MDS API into the format expected by remote clients. The application-level protocol of the HTTP Push Service defines event data as a JSON map of key-value pairs. This correlates naturally to the .NET dictionary data structure, so the easiest way to provide remote clients with data in the expected format is to convert `MdsMessage` objects into dictionaries and

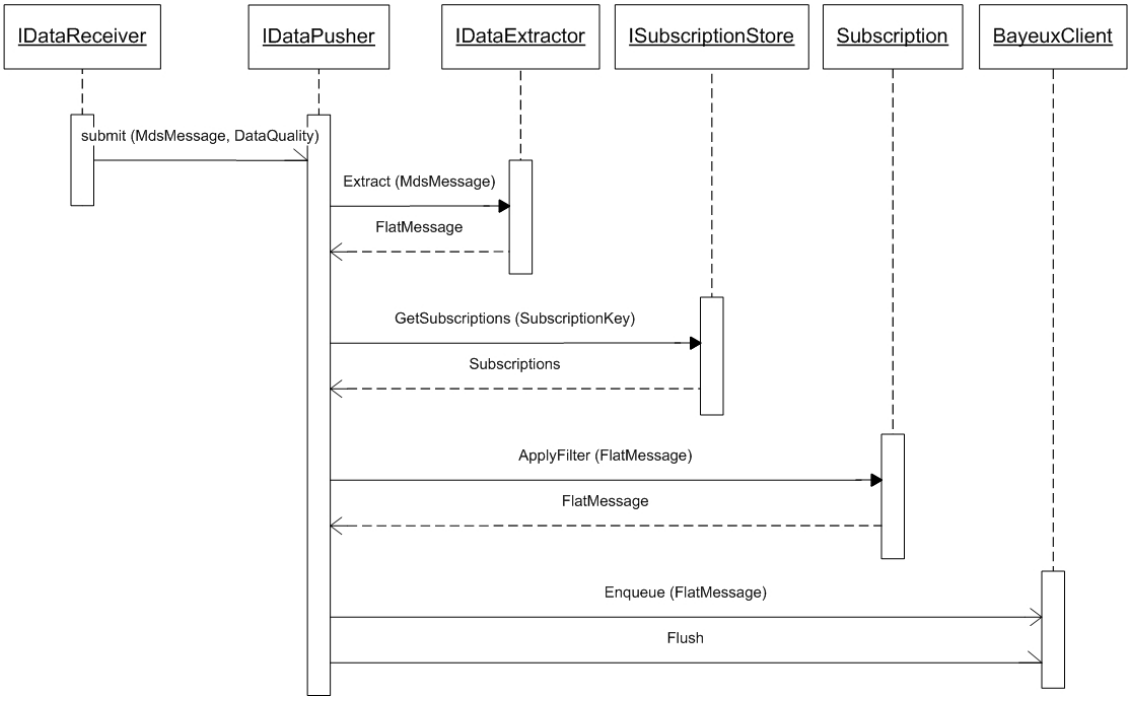

Figure 4.9: Interaction between components involved in processing and publishing updates

let Comet.NET serialize them automatically into JSON key-value maps. The task of converting MDS messages into dictionaries is delegated to `IDataExtractor`.

Once data is available as a collection of key-value pairs, data pusher proceeds to retrieve subscriptions and publish the data to clients associated with them. Every subscription knows what fields were requested by the client and can extract only those fields from the dictionary that was created from `MdsMessage`, so the final step is to iterate over subscriptions, apply the filter for each of them and enqueue the resulting dictionary into `BayeuxClient` referenced by the subscription. This ensures that every client gets a customized event message, as specified upon subscription.

The fine-grained filtering of event data poses certain architectural constraints that result in performance overhead. `DataPusher` cannot work directly with clients and channels, but has to operate on subscriptions instead. It has to retrieve a list of subscriptions and apply filtering for each of them on every incoming MDS message. It also has to publish a separate event message for every client. Since event messages are customized, there is no single message that can be broadcasted on the channel. This means that Comet.NET has to serialize and send multiple messages for each update message received from MDS. With exception of serializing event messages with overlapping content, which can be optimized to a certain extent, all mentioned aspects are “by design” and cannot be optimized.

Message aggregation The state of a symbol can change at a very rapid pace, with up to several hundreds of changes per second. The subscription-based, streaming MDS API was designed primarily for applications that need to receive all changes (such as backend services and desktop charting applications), so the aggregation features are almost nonexistent. MDS simply propagates all changes in symbol state to subscribed clients, in as many messages as needed.

The HTTP Push Service targets mainly Web clients that typically update Web pages with changes as they occur and rarely need to further process the data. Since only the latest value of a field is displayed and the human eye can perceive only a limited number of updates per second, it is safe to say that most Web clients do not need to receive all changes. For most of them, it is acceptable to receive only the latest values of fields that have changed during a predefined time interval. We call this time interval "aggregation interval" and define it as the amount of time between two deliveries of data to remote client. The aggregation interval depends on the transport type and on the server-side customer configuration and its range is typically between 0.25 and few seconds.

Even though most clients do not require all changes to be delivered to them, there is a small subset that poses this requirement. For example, Web applications that draw realtime charts or calculate financial indicators based on intraday quote data require every change in symbol state. Naturally, this type of clients induces more overhead on the server, because more messages have to be serialized and delivered. For a field that has 10 updates during the aggregation interval of 1 second, this type of client induces 10 times more overhead than the one that requires only the latest field values.

In order to support both target groups efficiently, the HTTP Push Service introduces a subscription parameter named *PushType*. For each subscription, a client can specify if it is interested in all updates (*Everything*) or only in the most recent changes that occurred during the aggregation period (*Snapshot*). In case of snapshot delivery, event data is filtered before each delivery so that it includes only the latest value of each field found in the data set. In case the delivery of all changes was requested, no filtering takes place.

Comet.NET offers a callback interface `IDataFilter` that can be used for this kind of filtering. An implementation of this interface can be attached to `BayeuxClient` and is then invoked every time messages are flushed to the underlying transport. This allows applications built on top of the library to implement any kind of message inspection and filtering prior to output: they can change existing messages by adding or removing fields, delete messages or even insert new ones.

In the HTTP Push Service, every client has its own data filter. It is a stateful filter that keeps track of client subscriptions and shapes the outgoing messages according to their parameters, with the goal to reduce the number of outgoing messages to a minimum. In the best case, when a client wants to receive only the latest snapshot for all subscriptions, the output is always one message per channel. In the worst case, when a client wants to receive all updates for all subscriptions, no reduction is possible and the filter returns the same message set.

Important prerequisite for message aggregation and filtering in the HTTP Push Service is that messages are not immediately sent to clients, but are rather buffered and sent in batches in a predefined interval. As briefly described in section 3.7.2, Comet.NET separates operations for enqueuing and flushing of messages, allowing applications to decide when data should be

flushed to the client. The actual flushing is executed in `DataPusher`, from one of its worker threads, and depends on global and customer-specific aggregation settings.

4.2.6 Security

Since Bayeux is a plain-text protocol, no credentials are exchanged using the protocol itself. The HTTP Push Service relies on a security provider for authentication and authorization of clients. The security provider is essentially a component that allows remote clients to authenticate and also communicates with the HTTP Push Service behind the scenes, offering authentication and authorization services. In a standard TeleTrader in-house setup, TeleTrader Web Service (TTWS) acts as the security provider.

Authentication is based on session tokens. Clients need to contact the security provider and get a session token, which needs to be supplied during the handshake with the HTTP Push Service. If the token is valid, the connection is established and the client can subscribe and receive data. If the token expired or is invalid, an appropriate error message is sent to the client and the connection is closed.

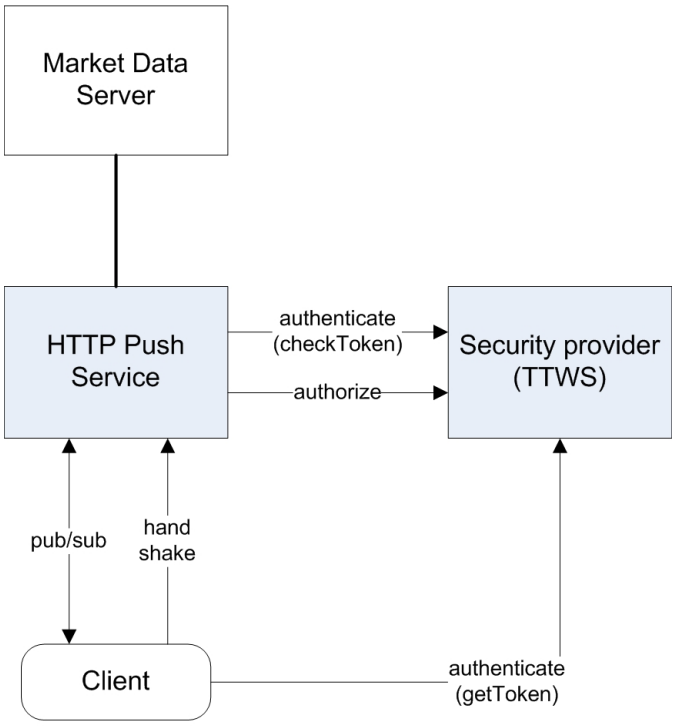

Figure 4.10: Security model of HTTP Push Service

The server provides a fine-granular authorization system that works on two levels: resources and fields. In order to be able to subscribe for a symbol or a news source, the client needs to have permission to access the resource. Additionally, each client can have access to all fields of

a resource or only to a specific set of fields, depending on its status in the system. Clients do not need to take care of permissions. This is handled by the server in combination with the security provider behind the scenes.

CHAPTER 5

Evaluation

The main goal of this chapter is the exploration of the actual performance and scalability of the presented solution. In addition to that, the chapter provides a performance comparison of the presented implementation with the before mentioned reference Bayeux server implementation - Jetty CometD.

For performance and scalability measurements, a series of repeatable benchmarks was defined along with relevant input and output variables and measurement methods. These benchmarks were then executed with predefined input variables such as the transport technique (polling vs. streaming) and the number of published messages per second. For each benchmark, relevant data was collected and analyzed afterwards.

General definition of benchmarks and their input and output parameters can be found in Section 5.1.1. The benchmarks and their results are presented and discussed in the Section 5.2. A comparison with the reference implementation can be found in Section 5.4.

5.1 Definitions and tools

This section describes the basic benchmark setup, testing environment and tools used for measurements of performance and scalability of Comet.NET.

5.1.1 Goal and general setup

The goal of benchmarks presented in this chapter consists of measuring the overall performance of Comet.NET in conditions that mimic a heavy-load real-life usage scenario. In every scenario, there is one server instance that runs on a dedicated server machine and many clients that are simulated by a client simulator application running on one or more machines. Every benchmark execution is started with a small predefined number of clients. The number of clients is progressively increased in batches until one of the two abort conditions has been reached: either the median message latency exceeds 1 second or more than 20.000 clients have connected to the server. All benchmarks are conceptually similar and have the same basic setup:

- There are 50 channels.
- Every client subscribes a predefined number of random channels.
- All published messages have the same size.
- The Publisher publishes messages to a subset of channels every second. How many messages are published per second is defined in the benchmark configuration.
- Clients are added in batches with a pause between each batch. The size of the client batch and the pause between the batches is defined in the configuration.
- The benchmark is stopped once the median message delay exceeds 1 second.
- The median message delay is the median delay of all messages received by a client during the 30 seconds.

The *message delay* is defined as the difference between the time the message was created by the Publisher application and the time it was received by the client. Every time a message is created by the Publisher, a timestamp is attached to it. The client takes a timestamp when the message is received and calculates the delta to the message timestamp. This value shows how long it takes for a published message to reach the client. The message delay can be used to measure how fast a client gets notified with the latest events.

While all benchmarks have the same basic setup, they differ in three aspects that are represented by the following input variables:

- Number of messages per second (1, 5, 20)
- The payload of published messages (150, 500 bytes)
- Transport type used for communication between the server and the client (polling, streaming)

These variables are used to simulate different situations and evaluate how Comet.NET performs in each of them.

UTF-8 is used for encoding of messages, so a payload of 150 bytes represents at least 37 and at most 150 characters. The same payload equals to a JSON map with 7 key-value pairs where keys consist of 10 characters from the ASCII set in average and values are containing floating point values with 4 decimals. A message payload of 500 bytes represents at least 125 and at most 500 characters or a JSON map with 24 key-value pairs with the structure described above.

All benchmarks focus primarily on streaming, because of the advantages this communication paradigm has in comparison to polling (discussed in section 2.1.2). Every benchmark is also executed with clients using only polling transport type. Both results are presented and discussed, if not explicitly stated otherwise.

All components involved in the execution of benchmarks were configured to produce and/or collect data during the execution that is relevant for the analysis of Comet.NET's overall performance. In particular, the following output variables are used in the analysis:

- The CPU usage of the server application (and therefore, of Comet.NET)
- The number of concurrent clients connected to the server
- The number of messages sent out by the server every second
- The amount of data in bytes sent out by the server every second
- The mean time it takes for a client to receive a published message

5.1.2 Tools

Most of the tools used in the benchmarks are custom tools developed for this purpose. This section briefly describes both custom-made and already existing tools that were used for benchmarking.

Client simulator The Client Simulator is a very flexible .NET console application that can simulate any number of Bayeux clients. It uses a full-featured Bayeux API for .NET, which was also developed specifically for purposes of load and functional testing of Comet.NET. The client simulator can be configured to use polling, streaming or both, it can increase the number of clients gradually and can be instructed to subscribe every new client to all or a predefined subset of available channels. Most of the actions available in the Client Simulator involve a certain amount of randomness, which helps simulate real-life usage scenarios. The Client Simulator records the latency of each received message. Collected latencies are aggregated on-the-fly into bars with minimum, average, median and maximum values and subsequently written to a log file. The aggregation interval is configurable and is usually one minute or less. The log file is in a comma-separated format so it can be processed with Excel or some other spreadsheet processing application. The Client Simulator can be instructed to end the simulation after a predefined amount of time or as soon as the average delay reaches a certain point.

Server monitor For monitoring of basic server parameters such as number of concurrent clients, number of active channels and subscriptions to them, a special monitoring interface is used. This is a subset of monitoring capabilities incorporated into the TeleTrader HTTP Push Service (Section 4.2). The Publisher also writes aggregated statistics on number of concurrent clients, total number of subscriptions, number of messages and bytes sent out by the server into a special log file every minute.

Windows Performance Monitor The Windows Performance Monitor[80] is a very versatile tool for monitoring of computer performance by using data from various sources (performance counters and event trace data). It can be instructed to write all collected data into a log file in variety of formats. Logs can be used later to replay the behavior of recorded counters or to analyse their data with a tool such as Excel or some other spreadsheet processing application. The Performance Monitor was used to collect data on CPU utilization and network usage of the server application.

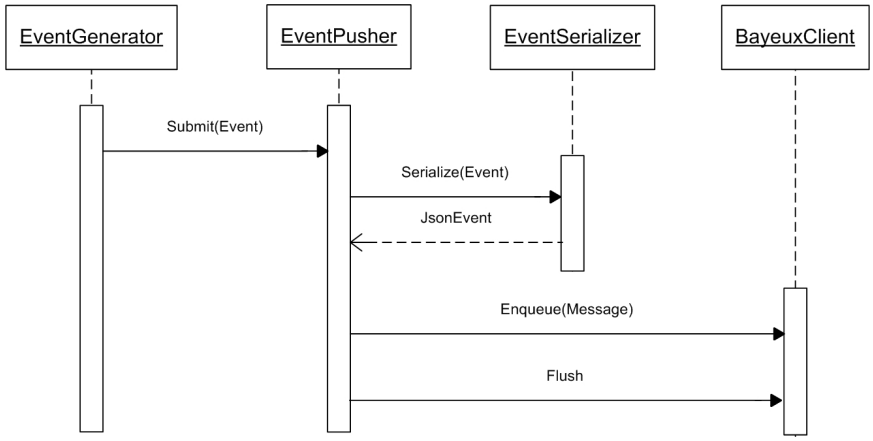

Figure 5.1: Event flow in the Publisher sample application

5.1.3 Sample Comet application

The server-side application used for performance evaluation of Comet.NET has basically two tasks: generation of synthetic events and orchestration of their routing and delivery to remote clients. The Publisher, as the sample server-side application is called, follows the same basic design found in the TeleTrader HTTP Push Service and consists of three main components: `EventGenerator`, `EventPusher` and `EventSerializer`.

`EventGenerator` is a component that generates synthetic events based on a range of configurable properties. Triggering of events is achieved with a simple timer with a configurable interval. The timer interval is set on instantiation and remains constant during the application execution. On each timer tick, the generator takes a certain number of channels from the list of active channels it maintains, generates events for them and forwards these events to `EventPusher`. The generator can be configured to generate events for all channels on each tick or just for a subset of them. In the latter case, channels are chosen randomly on each tick.

Event data is represented as a set of key-value pairs. Since the size of event data is fixed and the actual content of keys and values is not used on the client, event data does not have to be randomly generated on each tick. A much better solution in terms of performance is to pre-generate the event data on instantiation and reuse it every time the timer tick occurs. This strategy keeps the CPU usage for data generation as low as possible and leaves more resources for the tasks that we actually want to measure.

Tasks of `EventPusher` consist of accepting events from `EventGenerator`, enqueuing messages for delivery to clients and triggering the flushing of data to remote clients periodically. Since events are published per channel and do not contain any client-specific data, each event is serialized only once. This means that the overhead for event serialization is the same regardless of the number of clients it is being delivered to. The frequency of flush operations is configurable and is set to 100ms if nothing else is specified in test case definition. This means that the data is sent out to remote clients at most every 100ms.

`EventSerializer` is used for the serialization of event data into appropriate JSON

strings. The actual serialization is delegated to JSON.NET library[76], making `EventSerializer` a thin wrapper around this library.

5.1.4 Testing environment

Amazon's Elastic Compute Cloud (EC2)[81] was used for execution of all benchmarks. The server application was running on an Extra Large instance with 8 *EC2 Compute Units* and 15 GB RAM. The Compute Units were split among 4 virtual cores, with every core having 2 Compute Units. The client simulator was running on 4 identical Large instances with 4 EC2 Compute Units and 7,5 GB RAM. All instances were running 64-bit Windows Server 2008 and .NET Framework 4.0.

The machines were used with the standard configuration. Other than installing the latest version of .NET framework, no changes were done.

5.2 Benchmarks and results

This section presents the benchmarks, their specific setups and describes briefly the real-life scenarios they represent. Results of each benchmark are also presented and discussed.

5.2.1 Benchmark 1

Setup The first benchmark was designed to measure the performance of Comet.NET in combination with lightweight clients and a data source that has a constant and rather slow frequency of updates. A Web page with a tick chart for a single quote that is updated in real time is one example of such a scenario. Another example is a "details" page in a Web-based monitoring cockpit of a service that publishes health and status data periodically.

The goal of the test was to show that Comet.NET can support a very large number of clients interested in a very small number of resources that have low update frequency. The specific setup of the test was as follows:

- Every client subscribed 5 random channels (out of 50 channels).
- The Publisher published 1 message per second on 10 randomly selected channels. This means that there was approximately 1 message per second per client.
- The payload of each message was 150 bytes.

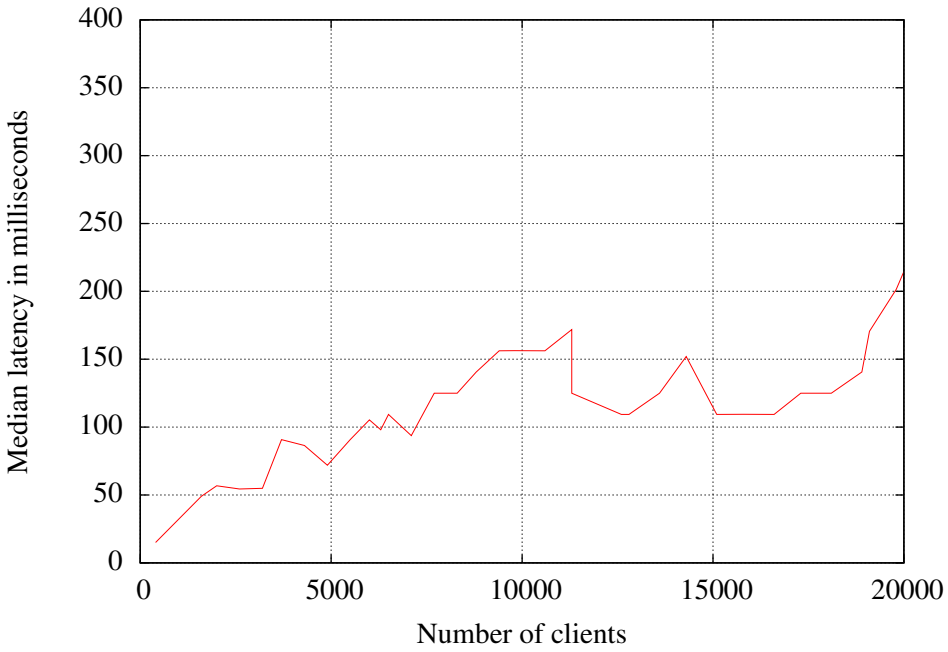

Figure 5.2: Number of clients and latency during benchmark 1 with streaming

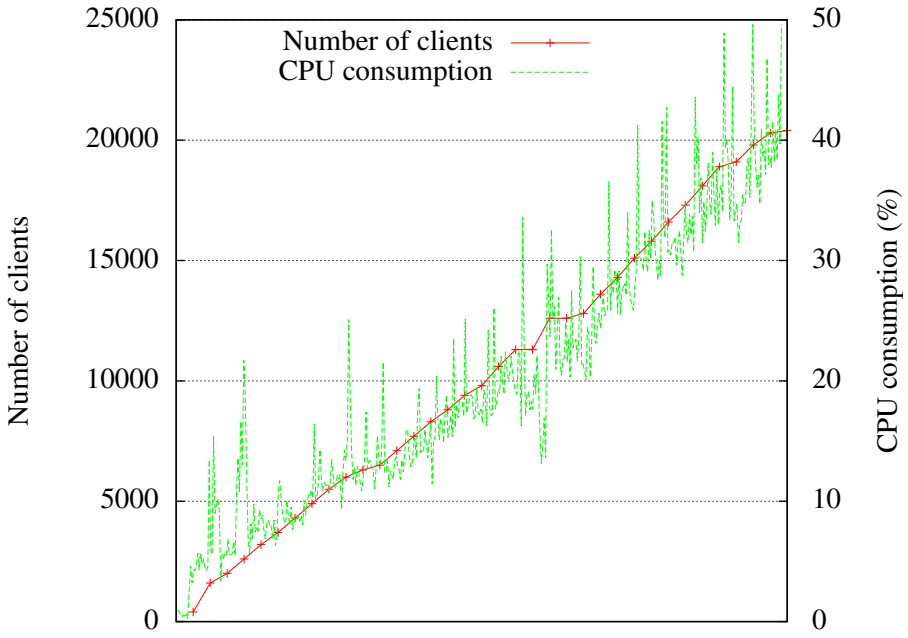

Figure 5.3: Number of clients and CPU utilization during benchmark 1 with streaming

Streaming Comet.NET was able to support over 20,000 concurrent streaming clients with an average message latency of less than 200ms. Note that this number does not include the network-induced latency, but only latency caused by the engine itself. Reaching this number of concurrent clients is one of the two abort conditions defined in Section 5.1.1, so the benchmark was terminated shortly after. Each client received approximately one message per second, this means that the library is capable of sending 20,000 messages with a payload of 150 bytes to different clients while keeping the average latency under 200ms. The relation between the median message latency and the number of clients is presented in Figure 5.2.

As shown in Figure 5.3, the CPU utilization and number of concurrent clients show a very strong linear correlation. With 5,000 concurrent clients, the CPU utilization was at 11%. Supporting 10,000 concurrent clients required 21% CPU. With 20,000 concurrent clients the test application was utilizing 45% CPU.

Polling When the client simulator was instructed to use only long polling transports, Comet.NET was able to support 9,000 concurrent clients. The reconnect interval, as defined by the Bayeux protocol, was set to one second. This means that every client waited one second between the termination of the HTTP response and the issuing of a new HTTP request, which in turn means that the expected average message latency was 500ms. The actual mean message latency was fairly constant around 500ms until 6,000 concurrent clients were connected and than began to

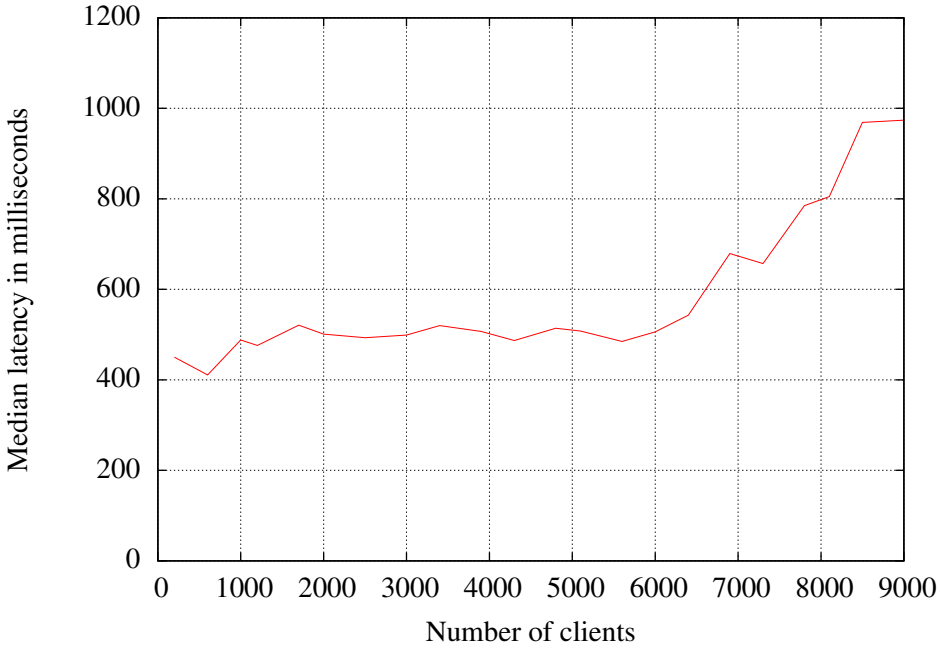

Figure 5.4: Number of clients and latency during benchmark 1 with polling

rise steadily. At 9,000 concurrent clients, the mean message latency reached the maximal allowed value of 1 second. The relation between the median message latency and the number of clients is presented in Figure 5.4.

The CPU usage had a very similar pattern to the one shown in Figure 5.3, so it is not depicted separately. At 9,000 concurrent clients and at the saturation point, Comet.NET was consuming approximately 75% of CPU.

In addition to this benchmark, a slightly modified version was also executed in an attempt to reach 20,000 concurrent clients with long polling. The only difference is that the requirement that the maximal allowed median message latency is one second was dropped. The reconnect interval was set to 5 seconds, so every client received a bulk of updates approximately every 5 seconds. Comet.NET was able to support maximal 16,200 concurrent long polling clients with the maximal CPU utilization of 78% and quite high median message latency of approximately 3 seconds.

5.2.2 Benchmark 2

Setup This benchmark was designed to put more pressure on the Comet engine per channel when compared to the benchmark 1. Each client was interested in the same number of resources, but each resource had more frequent updates, which resulted in more messages being published per second. A typical real-life example is a Web-based monitoring console of a software system

that displays status data for each component. There are few components, but each of them updates its status very frequently.

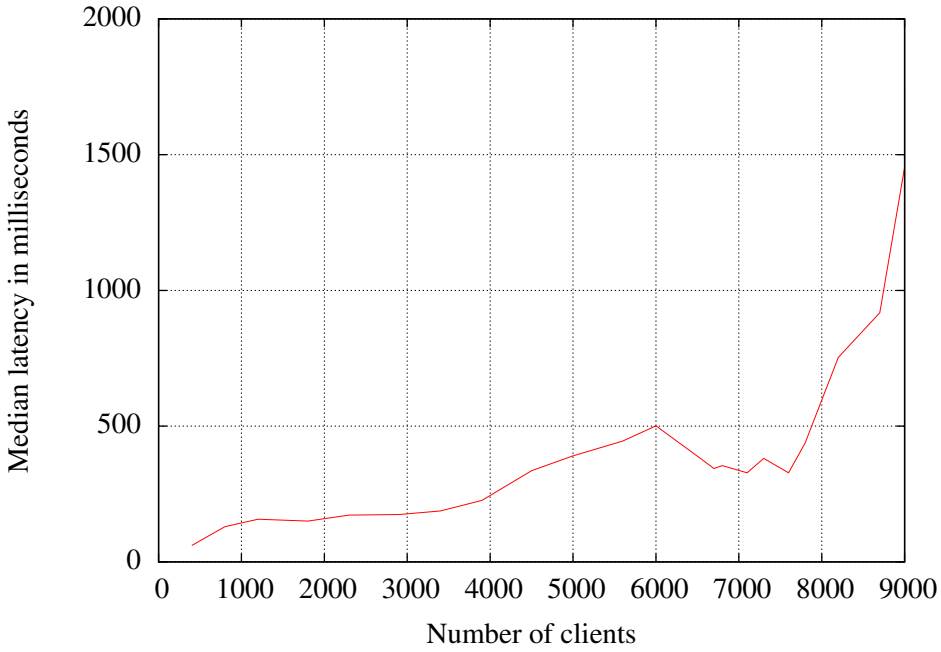

Figure 5.5: Median latency in relation to number of clients during benchmark 2 with streaming

The specific setup of the test was as follows:

- Every client subscribed 5 random channels (out of 50 channels).
- The Publisher published 1 message per second on every channel. This means that there were approximately 5 messages per second per client.
- The payload of each message was 150 bytes.

Streaming The engine was able to support 8,700 concurrent streaming clients before the maximal acceptable median latency of 1 second was reached. At that point almost 40,000 messages per second were sent out to clients via 50 channels. After that, the latency began to increase and the number of messages per second began decreasing. This can be interpreted as the saturation point for Comet.NET in this particular scenario. We can conclude that Comet.NET is able to distribute 40,000 messages per second with a payload of 150 bytes to 8,700 clients with latency under 1 second.

Figure 5.5 shows the relation between median message latency and the number of clients. It is important to note that Comet.NET maintained the median latency below 500ms until the number of clients approximated 8,000.

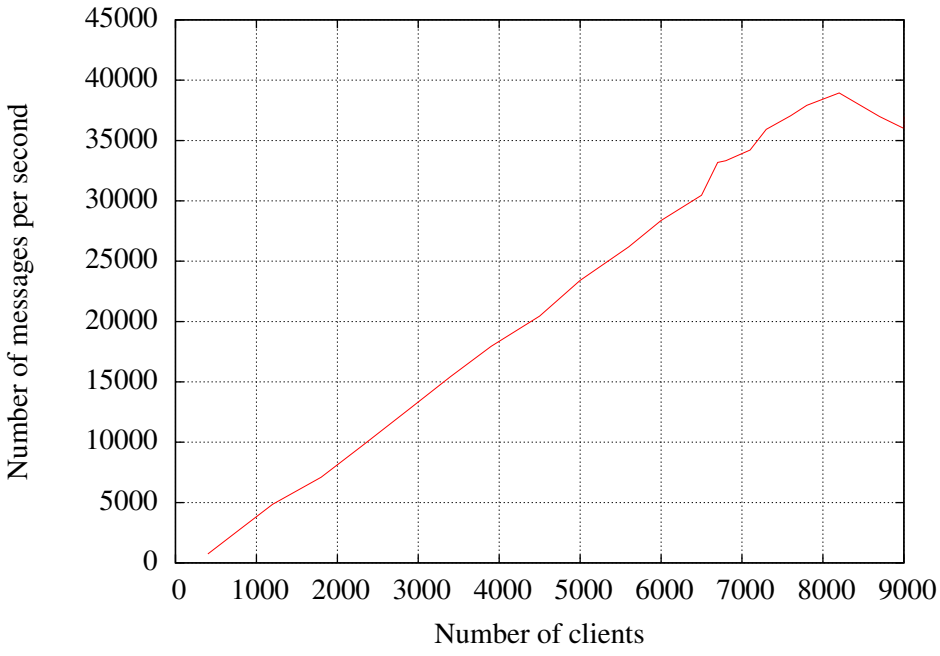

Figure 5.6: Number of messages per second in relation to number of clients during benchmark 2 with streaming

Figure 5.6 shows the relation between the number of clients and number of messages per second. The number of messages per second published by the test application and delivered by Comet.NET increased steadily as the number of clients rises. When the maximal acceptable median message latency was reached, at 8,700 clients, the number of messages per second reached the maximum and started decreasing. This is due to the fact that Comet.NET cannot deliver all messages generated by the test application in a timely fashion. This causes internal data structures to start filling up and the coordination overhead increases, which results in fewer messages being delivered.

The CPU utilization shows a very similar pattern to the one from benchmark 1: there is a strong correlation between the number of clients and the CPU usage. When serving 6,000 clients the CPU usage was approximately 50% and with 9,000 clients the CPU utilization reached 78%.

Polling As expected, when long polling is used as transport mechanism, the library was able to support less concurrent clients. The saturation point for this benchmark was reached with 4.300 concurrent long polling clients. Since the Bayeux reconnect advice interval was one second, the expected average message latency was 500ms. The actual mean message latency was kept around 500ms until 2,000 clients were connected, after which it began to rise steadily and reached the maximal acceptable value of one second at 4,300 concurrent clients.

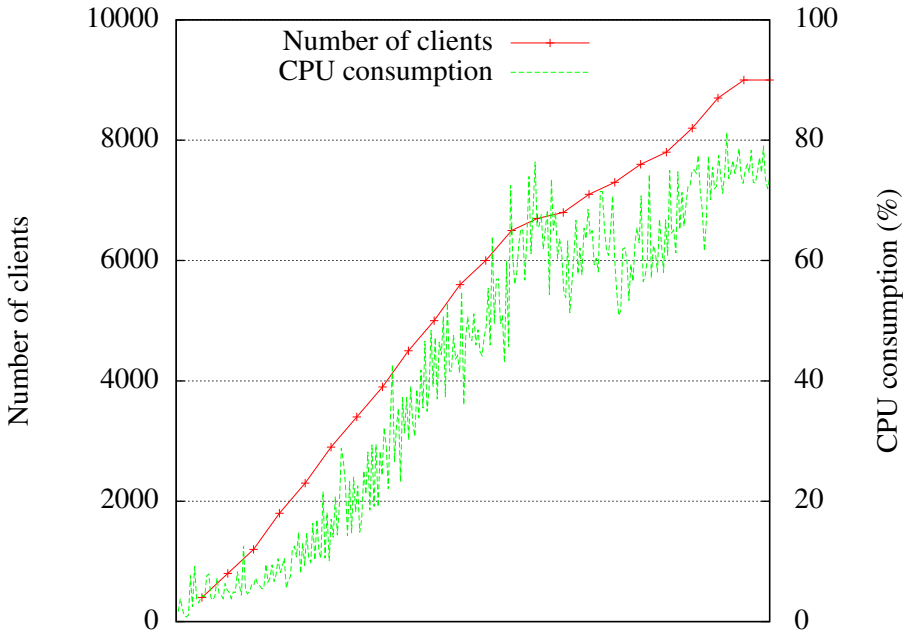

Figure 5.7: Number of clients and CPU utilization during benchmark 2

5.2.3 Benchmark 3

Setup This benchmark was designed to simulate a typical scenario for delivery of quote updates for a list of stocks to Web clients. The basic assumption was that every client had a medium-sized list of stocks with 20 items and wished to receive real-time updates for them. Stocks in the list were highly traded and had one or more updates per second. Clients wanted to receive one message per stock per second with the latest values of all changes fields. This resulted typically in 150 bytes of data per outgoing message.

The data source for this benchmark had the same set of resources and generated the same number of updates per second as the data source in the previous benchmark, but clients subscribed significantly more channels (20 vs. 5). This resulted in a significantly larger number of published messages.

The specific setup of the benchmark is as follows:

- Every client subscribed 20 random channels (out of 50 channels).
- The Publisher published 1 message per second on every channel. This means that there were approximately 20 messages per second per client.
- The payload of each message was 150 bytes.

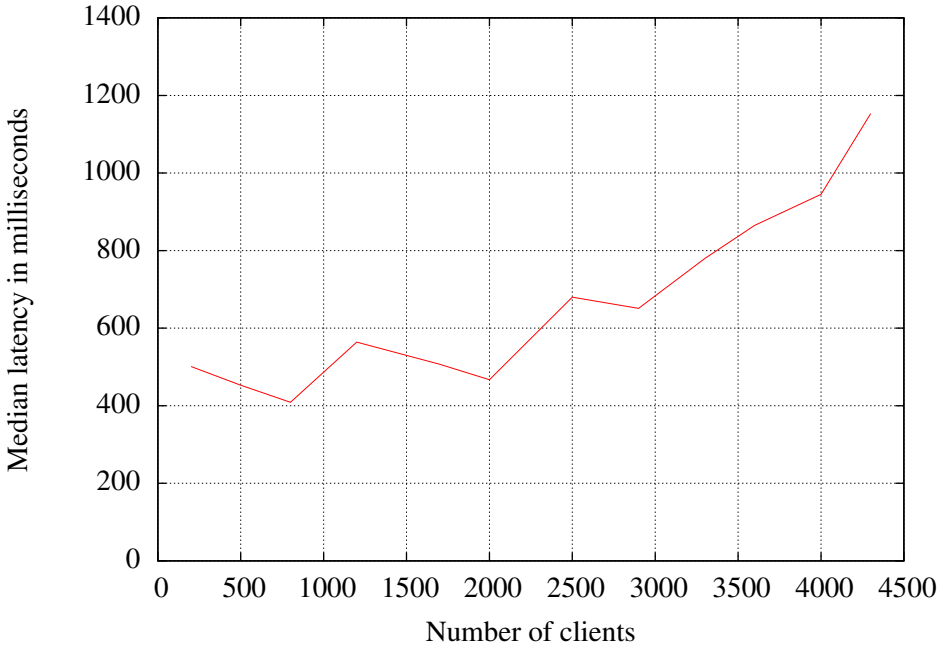

Figure 5.8: Number of clients and latency during benchmark 2 with polling

Streaming The engine was able to support 3,000 concurrent clients before the maximal acceptable median latency of 1 second was reached. At that point approximately 57,000 messages per second were sent out to clients via 50 channels. Although the median latency has reached 1 second and continued to rise after that point, there was no decrease in the number of published messages like in benchmark 2. The Comet engine managed to deliver all published messages to clients even though it was near the saturation point.

Figure 5.9 shows the relation between the number of clients, the number of messages per second and the median message latency. The increase in the CPU usage as the number of clients and the number of delivered messages increased can be seen in Figure 5.10.

Polling Comet.NET was able to support almost the same number of concurrent clients with long polling transport. The saturation point for this benchmark and polling transport was reached with 2,900 clients.

Due to the nature of long polling transport and its reconnect mechanism, the median message latency was always higher than the median message latency achieved with streaming. While the latency with streaming was under 250ms until 1,900 clients were connected, the latency with polling was never under 400ms. In fact, it reached 600ms with 1,200 concurrent clients and kept rising until the saturation point was reached.

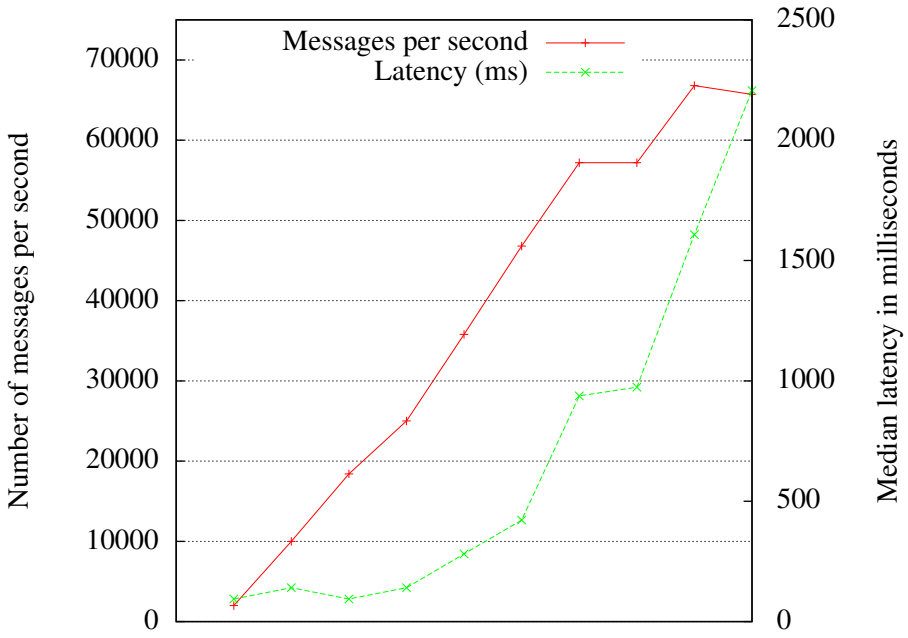

Figure 5.9: Median latency and number of messages per second in relation to number of clients during benchmark 3 with streaming

5.2.4 Benchmark 4

Setup The primary objective of this benchmark was to test the performance of Comet.NET in situations where very large amounts of data need to be delivered to clients. The payload size in this benchmark was more than three times the size of payloads in previous benchmarks: each message carried 500 bytes of data. This amounts to 25 key-value pairs with keys 10 characters long in average and values containing floating point values with 4 decimals. The update frequency was the same as in benchmark 2 and 3, as was the number of subscriptions per client.

In essence, this test scenario measured how well Comet.NET can handle the transmission of large amounts of data to each client. The specific setup of the benchmark was as follows:

- Streaming connection type is used by all clients
- Every client subscribed 5 random channels (out of 50 channels).
- The Publisher published 1 message per second on every channel. This means that there were approximately 5 messages per second per client.
- The payload of each message was 500 bytes.

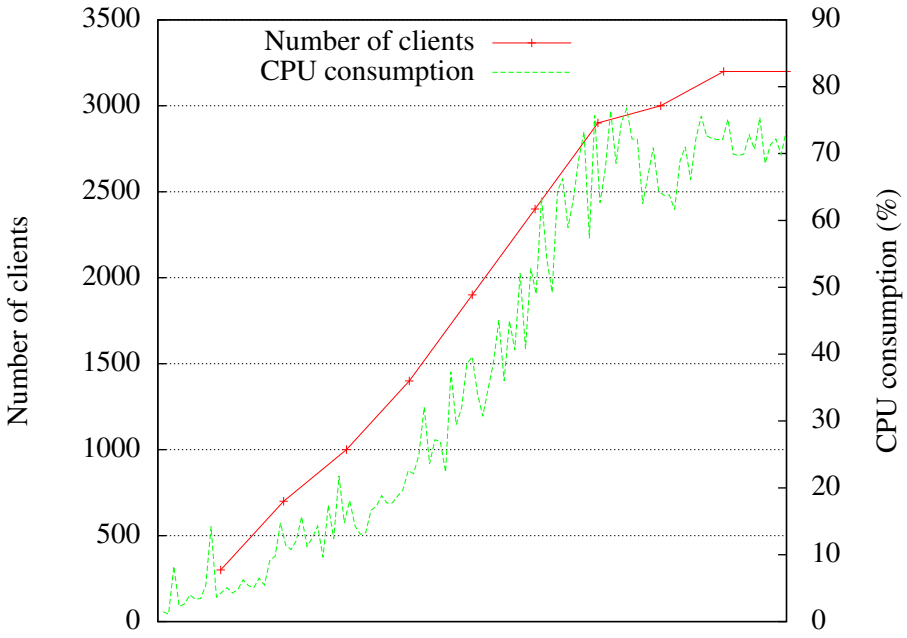

Figure 5.10: Number of clients and CPU utilization during benchmark 3 with streaming

Streaming The maximal number of concurrent clients that could be supported with median message latency under 1 second was 6,000. At the point where the latency reached this limit, approximately 30,000 messages were delivered per second via 50 channels.

In previous benchmarks, the latency started to rise continuously once the saturation point was reached. In this benchmark, the latency remained fairly constant and moved between 1.2 and 1.5 seconds as the number of clients rose to 7,000. It is also worth noting that the number of messages did not start to drop, as in previous benchmarks. While this does not influence the maximal number of concurrent users that can be supported in this test scenario, it does mean that Comet.NET is more stable in situations in which the amount of outgoing data is extremely high and still rising, but the amount of clients and their subscriptions is comparatively low.

Figure 5.12 shows how many messages are distributed per second to how many clients and what is the median message latency during the message delivery. This is a standard diagram that was also used to display the relation between these three variables in previous benchmarks.

The diagram in Figure 5.13 focuses on the amount of data distributed by Comet.NET per second (throughput) and shows the relation between the number of clients, median message latency and throughput. The size of the bubbles in the diagram represents the throughput. Numbers in the last three bubbles represent the amount of data in megabytes that is sent to clients every second.

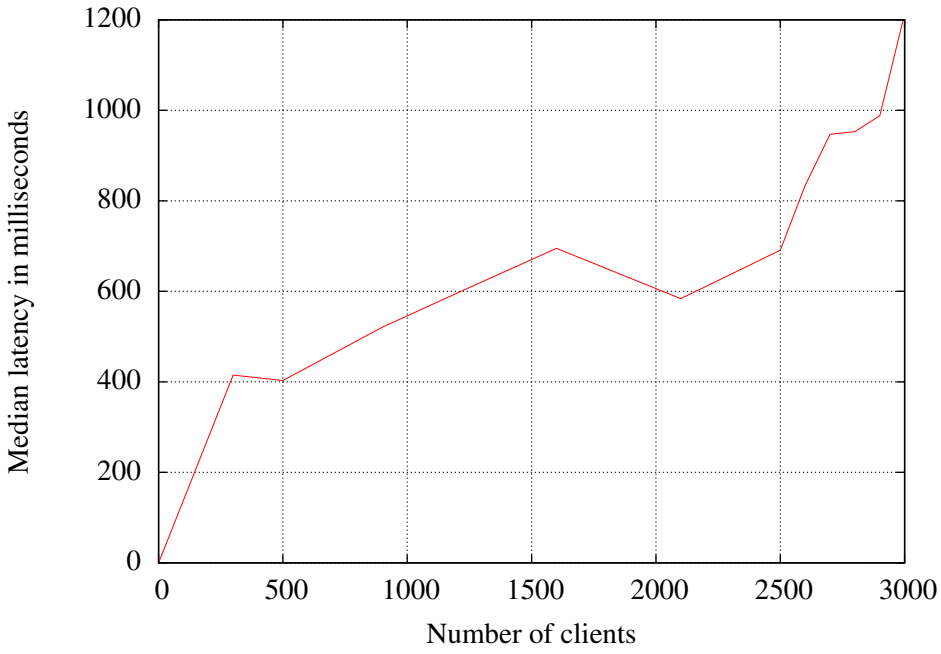

Figure 5.11: Number of clients and latency during benchmark 3 with polling

5.3 Benchmark result interpretation

This section summarizes the benchmark results presented in sections 5.2.1 through 5.2.4 and emphasizes important findings.

In benchmark one (Section 5.2.1) we showed that Comet.NET can support 20,000 clients that receive approximately one message per second. The median message latency was 250ms and the CPU utilization was at 45%. Furthermore, we showed that a median message latency of 100ms could be achieved for 7,000 concurrent clients and CPU utilization of only 17% . It should be noted that the Comet engine was configured to flush messages to clients every 100ms.

If the number of messages per second per client is increased to 5, Comet.NET is able to support 8,700 clients with the median message latency under 1 second (as shown in benchmark two, Section 5.2.2). The benchmark three, presented in Section 5.2.3, showed that even if the rate of outgoing messages is increased to 20 per second per client, Comet.NET is able to support 3,000 clients with the median message latency under 1 second. It should be furthermore noted that the median message latency was maintained under 250ms with almost 2,000 clients. At this point, there were approximately 40,000 outgoing messages per second. The maximal allowed latency was reached with approximately 60,000 outgoing messages per second.

In the first three benchmarks, the size of the message payload was 150 bytes. Benchmark four, presented in Section 5.2.4, showed the performance of Comet.NET when large amounts of data need to be pushed to clients (payload size of 500 bytes). The Comet engine was able to

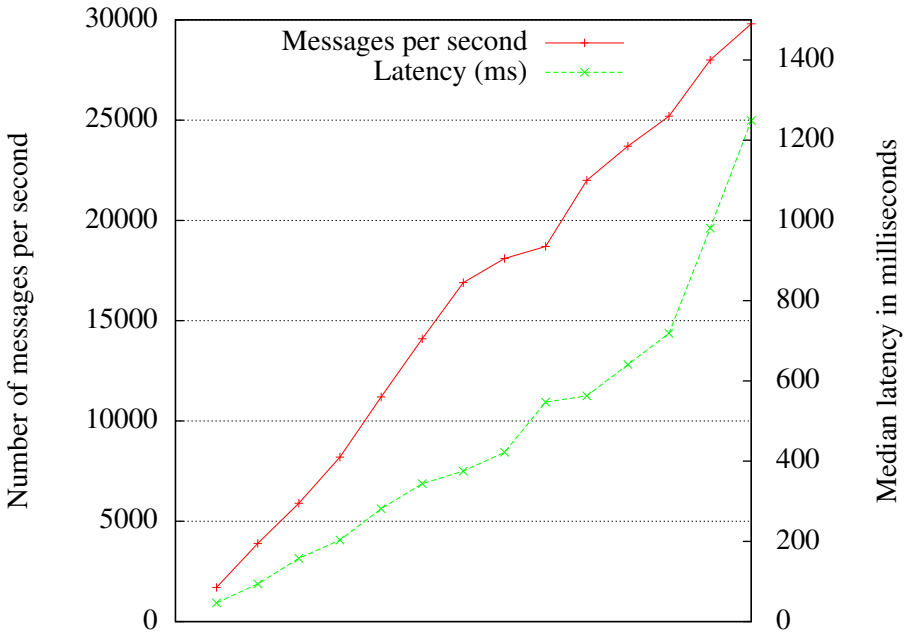

Figure 5.12: Median latency and number of messages per second in relation to number of clients during benchmark 4

support 5,900 clients with the median message latency under 1 second. At this point, there were approximately 29,000 outgoing messages per second. This amounts to 15 MB per second.

The results presented so far were achieved with streaming. When the polling communication technique is used, the performance is significantly degraded in all test cases. Due to the nature of long polling and its reconnect mechanism, the median message latency is always considerably higher than the median message latency achieved with streaming. In all benchmarks, the reconnect interval was set to 1 second, which means that the expected median message latency was 500ms. Another important aspect of long polling is that a new HTTP request/response cycle is required after each flush to a client (that is, after a bulk of updates was delivered to a client). This puts considerably more pressure on the Comet engine.

Benchmark one showed that Comet.NET can support up to 9,000 clients that use long polling transport while maintaining the median message latency below 1 second. The frequency of outgoing messages was one per second per client, which means that there were approximately 9,000 outgoing messages per second. This is 45% of the throughput that was achieved in the same benchmark with streaming. Even after the reconnect interval was increased to 5 seconds, the Comet engine was not able to support 20,000 clients with one outgoing message per client per second. The saturation point in this scenario was reached with 16,200 clients.

If the number of outgoing messages is increased to 5 per second per client, the saturation point is reached with 4,300 clients. With 20 messages per second per client, Comet.NET was

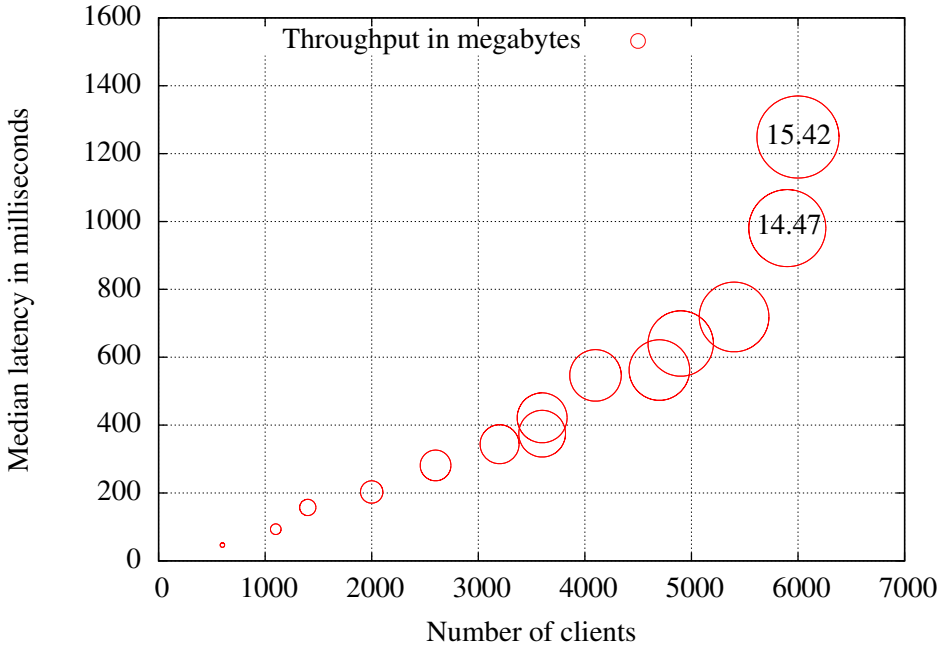

Figure 5.13: Median latency and throughput per second in relation to number of clients during benchmark 4

able to support 2,900 clients. These findings are shown in benchmarks two and three.

5.4 Comparison to Jetty CometD

One of the goals of the Comet.NET evaluation is to see how it compares to Jetty CometD - the reference Bayeux implementation - in terms of performance. The performance tests of Comet.NET and Jetty CometD were conducted on different hardware and have slightly different setups, so a direct comparison between the two solutions on the basis of these tests is not possible. We present performance findings for both solutions in this section without comparing them.

Bozdag et al [31] have done performance testing of Jetty 7 as part of their studies that compare push and pull approaches in modern Ajax-based Web applications. According to their findings, CometD is able to support at least 10,000 concurrent clients when one message was published to each client every second. The CPU utilization was slightly below 50%. The *publish triptime*, which is defined as the difference between the data creation time and data receipt time, was in average at 1 second.

G. Wilkins, the lead developer of Jetty and the CometD project, argues in [82] that Jetty 6 is able to support 20,000 concurrent clients with a subsecond latency. However, this result was achieved with a very low message frequency: a total of 3,800 messages were delivered

per second. This accounts for approximately 0.19 messages per second per client. The current version of CometD, delivered as part of Jetty 7, was tested in [83]. This paper shows that CometD is able to support 20,000 concurrent clients with a throughput of 10,000 messages per second with a latency under 500ms. This accounts for approximately 0.5 messages per second per client.

It is important to note that all presented performance tests were conducted with long polling as communication mechanism. Jetty CometD does not support streaming transports.

Comet.NET was not able to support 20,000 concurrent clients that use long polling, even with 5 second reconnect interval. It was however able to achieve a throughput of 9,000 messages per second, which is only slightly below the throughput achieved by CometD.

We showed that 20,000 concurrent clients can be supported with one message per second while maintaining median message latency below 250ms. A total of 20,000 messages per second were delivered in this benchmark. The maximum throughput achieved with Comet.NET was 60,000 messages per second, which was achieved with 3,000 concurrent clients. During this benchmark, every client received 20 messages per second.

CHAPTER 6

Alternative implementations of the Bayeux protocol

In this chapter we present three existing open-source Comet solutions that are based on the Bayeux protocol.

6.1 Jetty

Jetty is an open-source Web server component and servlet container written entirely in Java. It can be used as a stand-alone traditional Web server for serving static and dynamic content, as a dynamic content server behind a dedicated HTTP server such as Apache, or as an embedded component within a Java application. Its flexible component based architecture and small memory footprint allow it to be deployed and used in a variety of different scenarios, from embedded systems to clustered enterprise applications. Jetty is used by several other popular projects including JBoss and Apache Geronimo application servers.

The first server implementation of the Bayeux protocol was introduced with Jetty version 6. As of version 7, which is the current version at the time of writing, the Bayeux server implementation is extracted into a separate project called CometD, but still delivered as an integral part of the Jetty release.

CometD is a servlet implementation of the Bayeux protocol and provides mechanisms that allow Web applications to easily utilize Comet concepts by using familiar servlet interfaces. The *CometdServlet* and accompanying classes from the CometD module handle all protocol-specific tasks, making the addition of Comet functionality to Web applications fairly easy. The following are the three main components responsible for handling Bayeux requests:

- CometdServlet: the servlet that processes the incoming request and delegates work to other objects
- Handlers: each Bayeux request type is handled by a different handler

- Transport: this object takes care of sending the actual response to the client, formatting it depending on client's connection type

The included Cometd module is the reference server implementation of the Bayeux protocol in Java. The Jetty project also includes a reference client implementation of the Bayeux protocol in Java, based on the Jetty HTTP client.

6.1.1 Continuations

As described in section 2.1, Comet applications have very different traffic profiles compared to traditional Web applications. The classical one-thread-per-request model leads to one-thread-per-user in scenarios typical for Comet applications, and that does not scale well with an increasing number of concurrent users. Jetty uses a concept called Continuations[54] to solve this problem and achieve scalability with Comet applications. This section describes the concept and its implementation in Jetty.

	Formula	Web 1.0	Web 2.0 + Comet	Web 2.0 + Comet + Continuations
Users	u	10000	10000	10000
Requests/Burst	b	5	2	2
Burst period (s)	p	20	5	5
Request Duration (s)	d	0.200	0.150	0.175
Poll Duration (s)	D	0	10	10
Request rate (req/s)	rr=u*b/20	2500	4000	4000
Poll rate (req/s)	pr=u/d	0	1000	1000
Total (req/s)	r=rr+pr	2500	5000	5000
Concurrent requests	c=rr*d+pr*D	500	10600	10700
Min Threads	T=c T=r*d	500 -	10600 -	- 875
Stack memory	S=64*1024*T	32MB	694MB	57MB

Figure 6.1: The advantages of Jetty Continuations in Web 2.0 scenarios [54]

Java does not provide a mechanism to suspend a thread and then resume it later, so Comet servers are left with two options: block the thread handling the request until there is data to be pushed as part of the response or find a workaround for this limitation. The Java Servlet API 3.0[56] introduces support for asynchronous request processing, but the final version was released recently in early 2011 and there is still no wide support for it.

Jetty has a non-traditional approach in solving the issue with synchronous servlets. It introduces a "Continuation object" which allows the processing thread to pause the current request if there is no data to be sent to a client and make itself available for processing of another incoming request. The suspended request is resumed after a timeout or if the resume method of the Continuation object is called. This is typically done from another thread, when new data is available to be sent to the client. Behind the scenes, Jetty signalized that the request should be suspended and the processing thread returned to the thread pool. Continuations in Jetty 6 use an exception to signalize that a request should be suspended. While exceptions allow the thread to legally exit the handling method and effectively put the request on pause, it is rather unusual to use exceptions as control flow mechanism. As of Jetty 7, the Continuation mechanism has a clean API and implementation and is compatible with the Java Servlet API 3.0.

The Continuation mechanism is also fully compatible with the Java Servlet API 2.5[55]. Applications that use Continuations and are executed on a server that does not support them, will still work without any modifications. However, when the suspend method is executed, the thread will block and will not be returned to the thread pool.

Figure 6.1 shows the usage of server resources for classic 1.0 and Comet-based, 2.0 Web applications. The first column shows data for a classic Web application running on Jetty server. The second column shows data for a Comet application running on Jetty without Continuations (traditional one-thread-per-request approach) and the third column shows data for the same application with support of Continuations.

The number of concurrent requests is an estimate for 10,000 concurrent users of Web applications with regards to average behavior of Web 1.0 and Web 2.0 applications. Based on the number of concurrent requests, a minimal number of threads and needed stack memory is calculated. The figure shows clearly what benefits Continuations provide for Comet applications in terms of server resource usage and scalability.

6.2 Grizzly

Grizzly is an HTTP Connector based on Java NIO[57]. Originally developed as a part of the application server Glassfish, it is now available as a separate framework and can be easily embedded into any Java application. One of the primary goals of Grizzly is to help developers build scalable and robust servers by utilizing the power of Java asynchronous I/O operations. It offers a natural and very extensible abstraction on top of the low-level and inherently complex Java NIO API.

There are two different implementations of Comet in Grizzly[58]. There is Grizzly Comet, a server-side push solution specific to Sun Glassfish Enterprise Server and on top of it, a full featured implementation of the Bayeux protocol. The Comet implementation in Grizzly is positioned considerably lower in the stack than it is the case with Jetty (the main reason here is the

fact that Grizzly does not implement the Servlet API), but the implementations are quite similar. This section focuses on the Bayeux server part of the Grizzly framework.

The main parts of the Bayeux Grizzly implementation are:

- CometEngine: The entry point to any component using Comet. Components can be servlets, JSP, JSF or plain Java classes
- CometContext: a shareable comet data store belonging to one application. This is used as a central place for publishing events
- CometEvent: an object containing the data about an event relevant for the CometContext
- CometHandler: the interface defining ways of communicating with CometContext

6.2.1 Asynchronous Request Processing (ARP)

Grizzly uses the technique called asynchronous request processing (ARP) to avoid the classical one-thread-per-request model and its limitations. Instead of running a thread for each open connection, Grizzly's ARP mechanism efficiently uses the thread pool system and also keeps the state of requests so that it can keep requests alive without holding a single thread for each of them. In other words, the ARP allows "parking" of a request on the server without blocking the thread that was initially assigned to the request. Requests that are put on hold can be resumed at a later time and are then processed by an arbitrary thread from a thread pool. The ARP is an extension of the Grizzly Framework and sits directly on top of it.

The Comet module can make use of the ARP's ability to suspend and resume requests to efficiently implement the typical use case where a long-polling request is kept on the server until data is available. Of course, the same technique can be used for streaming scenarios, where responses are kept open for a long time with only occasional connection recycling. An expiration mechanism for suspended requests is also easily implemented.

The ARP achieves the same goal as Continuations in Jetty. However, it is a much cleaner approach because it uses Java NIO that natively supports separation of threads and connections and requests. There is no need to use exceptions as a workaround for suspending a working thread. This is also due to the fact that ARP, unlike Jetty Continuations, does not have the inherent limitation of the Java Servlet API. The obvious downside of this is that Web applications cannot integrate Comet functionality directly (i.e. as a Servlet). Grizzly's Comet server needs to run in a separate process and listen on a dedicated port. It can be used by any client (same domain or cross-domain), provided the firewall is configured to allow traffic on the used port, but there is one non-trivial challenge: sharing data between the Web application and Comet server. Since they run in separate processes, some type of common shared medium is needed for communication. This makes adding new Comet features to Web applications more complicated by several orders of magnitude.

6.3 Atmosphere

Atmosphere[59] is a POJO (Plain Old Java Object) portable Comet framework designed to make it easier to write and deploy Web applications that include a mix of typical RESTful and Comet behavior. It is essentially an abstraction layer that introduces a common API for asynchronous operations and hides differences and incompatibilities between various native asynchronous APIs introduced in Java application servers. One of the protocols that can be used for communication with remote clients is the Bayeux protocol.

There are three main operations that Atmosphere supports. The first one is the ability to suspend the execution of a request/response cycle until an asynchronous event occurs. Once the event occurs, the application may want to resume the normal request/response processing, depending on the technique the application supports (polling or streaming). The third operation consists of being able to broadcast or actively push asynchronous events and deliver them to suspended responses.

Atmosphere consists of several modules, most of them optional. For Comet scenarios, the following modules are relevant:

- Atmosphere Runtime

  This is the main module that represents the portable Comet runtime. It can be used with POJOs written in Java, JRuby or Groovy. The main component of this module is an AtmosphereHandler. An AtmosphereHandler can be used to suspend, resume and broadcast and allows the use of the usual HttpServletRequest and HttpServletResponse APIs

- Atmosphere Bayeux

  This module includes the implementation of the Bayeux protocol.

The framework supports all major application servers and their asynchronous HTTP processing APIs, as well as the standard Java Servlet API 3.0. It dynamically inspects the environment in which it runs and tries to use native asynchronous API available in that environment. If it fails, it falls back to its own mechanism that simulates asynchronous behavior by blocking the thread involved in processing of the request. This is the same behavior exhibiting by Comet applications written for Jetty CometD when they are deployed on a server that has no support for Continuations or Java Servlet API 3.0.

CHAPTER 7

Related Work

In this chapter we discuss the related work in the field of push-based systems in general (Section 7.1), the Comet application model and push-based delivery of data to Web browsers (Section 7.2) and concrete Comet implementations (Section 7.3).

7.1 Background on push technology

The push-based communication style has been researched very extensively in the distributed systems research community. However, most of the work is focused on systems that do not operate in Web environments.

In [22], S. Archarya et al present a push-based system for broadcast delivery of volatile and time-sensitive data to a very large number of clients. The *Broadcast Disk* paradigm, presented in the paper, provides improved performance, scalability and availability of the networked applications based on asymmetric communication capabilities. They further improved the proposed solution in [23] by augmenting the push-only model with the possibility of explicit pull operations. According to the findings presented in the paper, the enhanced model, based on push communication extended with a pull mechanism, performs better and is more flexible than the original one.

K. Juvva and R. Rajkumar propose a middleware layer called *Real-Time Push-Pull Communications Service* [24] and present the design, implementation and evaluation of the middleware. The main goal of the middleware is to enable quick data dissemination across heterogeneous nodes with flexible communication patterns. This is achieved by supporting both push and pull model, providing several levels of data delivery frequency (various levels of acceptable delays) and two common communication styles: synchronous and asynchronous communication. The middleware is targeted at real-time and multimedia systems.

M. Hauswirth and M. Jazayeri [25] define the architecture, the communication model and the component model for scalable push systems. The model is based on a publish/subscribe paradigm and multicast delivery of data. It is accompanied by an open protocol suite for the

content distribution and a reference implementation called *Minstrel*. However, the presented protocol is not designed to work over HTTP, does not take into account browser-specific limitations and is as such not suitable for the Web.

M. Ammar et al [26] define a set of protocols and a Web server architecture for scalable multicast delivery of Web pages. The authors present a model in which a Web server delivers pages with one of the three delivery options: cyclic multicast, reliable multicast and reliable unicast. The performance analysis presented in the paper shows that incorporating all three delivery options in Web servers offers better performance and flexibility. The authors state that the solution was not deployed and tested over the public Internet, because one of the requirements - reliable multicast protocol - is not met in the target environment.

R. Khare and R. Taylor [27] propose several extensions of the World Wide Web's REpresentational State Transfer (REST[28]) architectural style that support distributed and decentralized systems. They define extensions for asynchronous broadcasting, message routing, transactional processing of updates and explicit formulation of data delivery estimations. These extensions form the basis of four new architectural styles derived from REST: ARREST for centralized, ARREST+D for distributed, ARREST+E for estimated and ARRESTED for decentralized resources. The presented architectural styles are based on asynchronous events, a feature that poses "a significant implementation challenge across the public Internet".

A combined push-pull solution for dissemination of dynamic data in the Web is presented by P. Deolasee et al [29]. The paper presents two dynamic adaptive algorithms for data delivery: *Push and Pull (PaP)* and *Push or Pull (PoP)*. The algorithms essentially use a range of static and dynamic parameters to calculate the optimal data delivery technique for a given scenario. Although the reference implementation presented in the paper is based on the HTTP protocol, it relies on custom proxies and is therefore not deployable to the public Internet. However, the results of the extensive performance analysis suggest that their adaptive data dissemination model is superior to both pull-only and push-only in terms of temporal coherency, resilience to failures and efficiency and scalability.

7.2 Comet application model

Rich Web applications with real-time asynchronous updates based on server-side push have received a lot of attention in the Web development community recently. However, only a few scientific papers have been published on the topic so far.

Mesbah and van Deursen [30] present a new architectural style for Ajax and Comet Web applications called SPIAR (*Single Page Internet Application aRchitectural style*). The style emphasizes user interface components, intermediary delta-communication between client and server components, as well as push-based event notification of state changes. The authors argue that Web applications built with SPIAR have improved user interactivity, user-perceived latency, data coherence and the ease of development.

The presented architectural style results from a study of several popular Ajax frameworks. The only analyzed framework that supports the Comet communication paradigm is Dojo CometD on the client accompanied by Jetty CometD on the server. The framework is based on the Bayeux protocol.

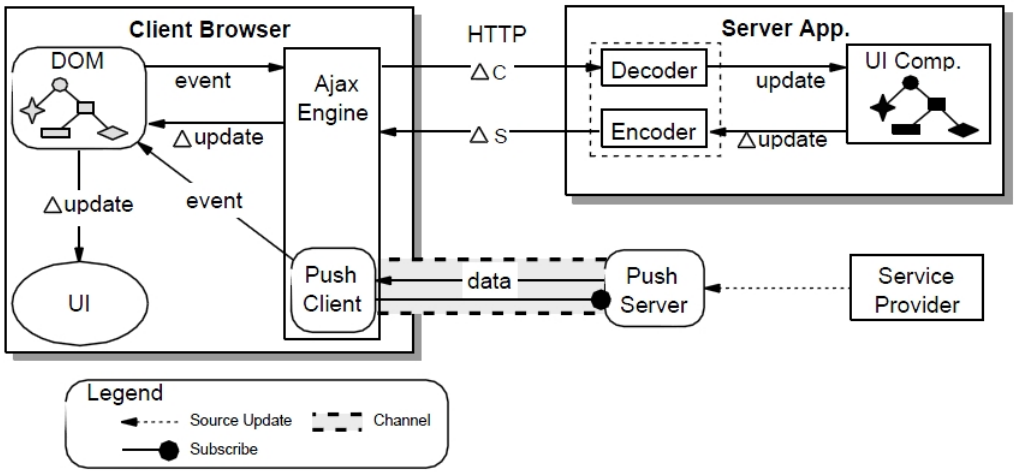

Figure 7.1: Direct push integration as defined in SPIAR

The paper presents two possible ways to integrate Comet into Web applications. Both approaches involve the two main processing elements: push server and push client. The push server resides as a separate module of the server application and has the ability to keep an HTTP connection open and push data to the client. The push client resides within the Web client. It can be a separate module or a part of the Ajax engine. The push client is responsible for receiving notifications from the server and providing it to the upper layers of the Web application.

The first approach for integration of Comet in Web applications is presented in Figure 7.1. The approach is very simple: data is pushed directly from the server to the client, bypassing any server-side representation of the client UI, delta-decoders and other components defined in SPIAR. Authors state that this approach has one flaw: it leaves server-side applications out of sync with the client-side UI. However, this is only the case for applications with an explicit requirement that the client-side UI and server-side application always remain in sync. Though, the authors do not present scenarios where such requirements exist.

The solution presented in this thesis is designed to be integrated into Web applications in a way that is very similar to this approach. The Comet server is a separate logical unit that resides on the server and communicates directly with the push client. The data flow between the server and the client is independent of other server-side application parts.

The second approach for Comet integration, presented in Figure 7.2, is significantly more complicated: it requires the push server to be an integral part of the server application and pass all events through several components of the application before they are sent to the client. Although it is more complicated, this approach has the advantage of keeping all involved components in sync and requires the client to use only one logical communication point towards the server instead of two.

Bozdag et al [31] conducted an empirical study comparing push- and pull-based delivery of data to Web applications with the goal to determine the performance trade offs between the two approaches. The study focuses on data coherence, scalability, network performance and

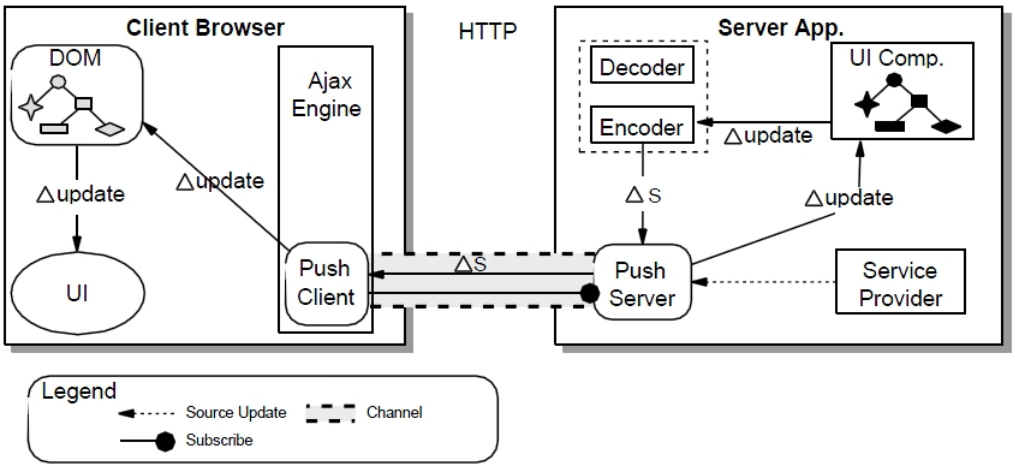

Figure 7.2: The indirect push integration as defined in SPIAR

reliability of push and pull communication style.

Tests involved two Web applications with push functionality, one using the CometD framework and the other using DWR (Direct Web Remoting), and one application based on the simple pull approach. All applications simulated the same scenario: market data distribution in form of a stock ticker. During the test, applications were supplied with a variable number of data items produced by a dedicated *service provider* server-side application. A variable number of concurrent users was simulated by a distributed user simulator.

The authors conclude that the push-based data delivery is superior to pull-based in terms of data coherence and network performance. Push-based delivery has significantly better mean trip-time, defined as the average time between the creation of a data item and its delivery to the client. According to the results of the study, CometD has a mean trip-time of 1,200 ms for 10,000 concurrent clients and one message per second. DRW has mean trip-time of 3,500 ms for the same number of clients and messages. Mean trip-time in pull-based application depends highly on the refresh interval, but is never lower than the trip-time of the CometD application. Data coherence, defined as the ratio between number of messages published and the number of messages received by the application, is also vastly better with the push approach.

Another important finding is that the CPU utilization on the server is always lower with pull approach than with push. At 10,000 concurrent clients, CPU usage reached 45% with CometD, 55% with DRW and was always below 18% for pull.

Since the solution presented in this thesis supports both the pull-based and the push-based delivery of data, it can be set up to use the push-based delivery as the superior one in all situations where it is technically possible.

Based on the work of Deolasee et al [29], Bozdag et. al present an adaptive algorithm that combines push and pull delivery of data for Ajax and Comet Web applications[32]. The main goal of this combined approach is to increase scalability and network performance while at the same time reducing user-perceived latency.

The authors argue that because push and pull communication paradigms have their own set of performance trade-offs and environmental factors such as data coherence requirements of users, data publish intervals or computational overhead tend to change over time, the hybrid approach to data delivery should be a better solution. The solution they propose is a modified and extended version of the *Push or Pull (PoP)* algorithm presented in [29].

Their dynamic adaptive solution is based on three algorithms. The *register* algorithm is used when a new user registers for a real time delivery of data. The remaining two algorithms are *monitoring* algorithms that keep track of the server and channel performance and adjust push and/or pull settings if necessary.

The work is theoretical, but the assumptions about push systems in the Web are heavily based on CometD and its features. While the underlying Bayeux protocol allows pure streaming to be implemented, CometD only supports long polling. This communication technique is inferior to streaming, as discussed in 2.1 and it can be argued that this has a severe impact on the presented algorithms and their performance.

7.3 Comet-based server applications

While there are several commercial and open-source Comet servers available on the market today, there are very few research papers that discuss the architecture and implementation of such a server.

Actually, the only published work on the topic is by Pohja [33]. The paper evaluates how an instant messaging protocol, namely XMPP[34], can complement HTTP-based Web applications and presents an implementation of a push server based on that protocol. The goal of the research is to define an additional protocol to support server-side push and to provide a reference implementation.

The design of the system is presented in Figure 7.3. For serving of content in the classical pull manner, a standard Web server is employed. A separate XMPP server is responsible for the delivery of dynamic content by using server-side push. On the client side, the data delivered this way is received by a XMPP client running in a browser.

The system is based on the publish/subscribe messaging paradigm. Clients subscribe to topics or channels, which are expressed by URIs. There is a one-to-one match between URIs on the Web server and topics on the push server.

Components responsible for providing event notifications that are delivered to clients are called *event sources*. They are activated by the Web application running on the Web server, usually when a client issues a subscription for a certain topic. Each event source holds the information what data is relevant for a certain subscription and publishes an event every time the data changes. In the implementation presented in the paper, event sources are able to track only database changes, but could be implemented to track any kind of data store. After an event has been published by an event source, the XMPP server delivers it to all interested clients.

The author states only that “the performance of the push server is similar to Comet that has been shown to outperform legacy HTTP polling applications clearly”. However, no details on the conducted performance evaluation of the server are provided in the paper.

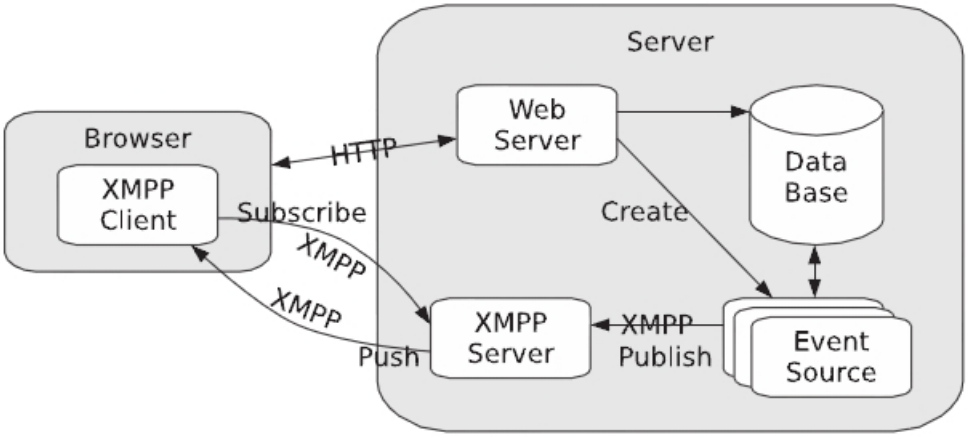

Figure 7.3: The components of the Comet server presented in [33]

The part of the system that is responsible for pushing event data to Web clients is designed very similarly to the Comet server presented in this thesis. Both servers are based on publish/-subscribe protocols, run separately from the Web server, have one or more sources of data and deliver event notifications asynchronously to clients running in Web browsers.

There is however one crucial difference: XMPP supports only long polling and has no extensions for streaming, whereas Bayeux allows streaming transports to be used if both server and client support them. The Comet server presented in this thesis uses the extension possibilities of the Bayeux protocol and implements several streaming techniques in addition to long polling technique.

Another Comet server with a similar architecture is Lightstreamer[35]. It is a commercial server that is based on a proprietary publish/subscribe protocol, delivers data asynchronously to clients running in Web browsers, supports both long polling and streaming and pushes events published by one or more sources called *data adapters*.

CHAPTER 8

Conclusion and Future Work

Inspite of the lack of standardization through the W3C, the real-time push-based delivery of data to Web applications has been gaining a lot of attention in the Web development community. The Web application model, in which a server delivers data to a browser without the browser explicitly requesting it, is called Comet. It is an umbrella term for various techniques that try to reduce delays and deliver data asynchronously from a Web server to Web browsers.

One of the recent developments in this domain is the application protocol called Bayeux. The Bayeux protocol is a JSON-based protocol designed to overcome the traditional, client/server nature of HTTP. It is based on the publish/subscribe paradigm and provides means of two-way, low-latency communication between a server and a browser. Initially developed for the Jetty CometD project, it is now an open-source protocol with implementations in several languages and platforms.

This thesis presented Comet.NET, a stand-alone implementation of the Bayeux protocol based on Microsoft's .NET Framework. Comet.NET is a fully Bayeux-compliant class library that can be used by any .NET application that wishes to provide Comet functionality and deliver data in real-time to Web clients. The presented solution fully leverages the .NET technology stack and uses some of technologies not available in Java and other platforms. Most important examples are the Windows HTTP Server API (HTTP.SYS) and the Windows Communication Foundation (WCF). It supports the two common Comet communication styles - long polling and streaming - and works with all modern browsers as well as with fat HTTP clients. Since Comet.NET is intended for integration into other applications, it is quite extensible and has a wide range of configuration possibilities.

The functionality of the presented solution was demonstrated by the two sample applications: a Web chat and an stock market ticker. The chat sample demonstrated the basic features of Comet.NET and its compatibility with the existing Bayeux client-side applications. The stock market ticker provided an in-depth demonstratation how an enterprise-level application can be built on top of Comet.NET.

The performance and scalability of the presented solution was evaluated with a series of repeatable benchmarks. It became apparent that Comet.NET satisfied the *high-performance*

requirement defined in the goal of the thesis and can deliver a very large number of messages per second to a very large number of concurrent users.

8.1 Future work

Even though the presented server-side implementation of the Bayeux protocol is fully functional and is used as a foundation of an enterprise-level application in a production environment (Teletrader HTTP Push Service, Section 4.2), the development is not considered finished. There are several segments of the solution that can be improved in terms of functionality and performance. New features are under consideration.

The following improvements and optimizations are planned:

- WebSocket support

  The version 2 of CometD server and Javascript client introduced support for the WebSocket protocol [84]. Even though the protocol specification is still in an early draft stage and is expected to change, it has been gaining support in all major browsers in the last several months. WebSocket represents a large advance, especially for real-time, event-driven Web applications [85] and will without doubt play a significant role in the future of Web. Comet.NET does not currently support WebSocket. The support for this protocol will be introduced in the next version of the library. Due to layered design and strict separation of concerns, the introduction of WebSocket will have no systematic impact and can be done without much effort.

- Improve long polling performance

  Even though streaming is a superior communication technique, long polling is an acceptable fallback transport for scenarios where streaming does not work or the frequency of events is low. Performance evaluation of Comet.NET, presented in Section 5, showed that there is room for performance optimization in this segment. Further testing and profiling is needed to determine bottlenecks, which will then be improved.

- Reduce overhead for instantiation and configuration

  As briefly discussed in Section 4.1.4, the initialization and instantiation of a Bayeux endpoint with Comet.NET poses a certain overhead in terms of lines of code. The class `BayeuxServer`, which is the entry point of the library, has several dependencies and also requires a reference to a fully-initialized configuration object. Some dependencies in turn have their own configuration properties. While this approach offers great flexibility and is justified for complex applications, it also introduces unnecessary overhead for simple applications and services.

  This problem will be solved by introducing a factory for `BayeuxServer` and possibly other classes on which it depends. A set of configuration objects with meaningful defaults for most typical use cases will also be created.

Bibliography

[1] M. Jazayeri: *Some Trends in Web Application Development*. Future of Software Engineering, 2007.

[2] F. Garzotto: *Ubiquitous Web Applications*. Advances in Databases and Information Systems, 2001

[3] A. Ginige and S. Murugesan: *Web Engineering: An Introduction*. Multimedia, IEEE 8, 2001

[4] Facebook, `http://www.facebook.com`, visited 28.04.2011

[5] Business Insider: *Startup 2011 Tickets Facebook Has More Than 600 Million Users, Goldman Tells Clients*. `http://www.businessinsider.com/facebook-has-more-than-600-million-users-goldman-tells-clients-2011-1`, visited 28.04.2011

[6] Twitter, `http://twitter.com/`, visited 28.04.2011

[7] FriendFeed, `http://friendfeed.com/`, visited 28.04.2011

[8] reddit, `http://www.reddit.com`, visited 28.04.2011

[9] Youtube, `http://www.youtube.com`, visited 28.04.2011

[10] Hulu, `http://www.hulu.com`, visited 28.04.2011

[11] Flickr, `http://www.flickr.com`, visited 28.04.2011

[12] Google Docs, `http://docs.google.com`, visited 28.04.2011

[13] Thompson Reuters Eikon, `http://thomsonreuters.com/products_services/financial/eikon/`, visited 28.04.2011

[14] CometD project. `http://cometd.org/` visited 21.02.2011

[15] IBM: WebSphere Feature Pack for Web 2.0. `http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/web20/`, visited 20.01.2011

[16] Oracle: WebLogic HTTP Publish-Subscribe Server. http://download.oracle.com/docs/cd/E12840_01/wls/docs103/webapp/pubsub.html, visited 20.01.2011

[17] E. Kuehn, J. Riemer, R. Mordinyi and L. Lechner: *Integration of XVSM Spaces with the Web to Meet the Challenging Interaction Demands in Pervasive Scenarios.* 16th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, June 2007

[18] E. Boznag: *Integration of HTTP Push with a JSF AJAX framework*. Master's thesis, December 2007

[19] Oyatel's CometD .NET, https://github.com/Oyatel/CometD.NET, visited 20.01.2011

[20] WebSync, http://www.frozenmountain.com/websync/, visited 28.04.2011

[21] Microsoft Windows HTTP Server API, http://msdn.microsoft.com/en-us/library/aa364510%28v=VS.85%29.aspx, visited 11.05.2011

[22] S. Acharya, M. Franklin and S. Zdonik: *Dissemination-based data delivery using broadcast disks*. IEEE Personal Communications Journal, 1995

[23] S. Acharya, M. Franklin and S. Zdonik: *Balancing push and pull for data broadcast*. In SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data, 1997

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

[25] M. Hauswirth and M. Jazayeri: *A component and communication model for push systems*. ESEC/FSE '99, Springer-Verlag, 1999

[26] 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

[27] R. Khare and R. Taylor: *Extending the representational state transfer (REST) architectural style for decentralized systems*. Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, 2004

[28] R. Fielding and R. Taylor: *Principled Design of the Modern Web Architecture*. ACM Transactions on Internet Technology (TOIT), 2002

[29] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy: *Adaptive push-pull: Disseminating dynamic web data*. IEEE Transactions on computing, 2002

[30] A. Mesbah and A. van Deursen: *A component- and push-based architectural style for ajax applications*. Journal of Systems and Software, 2008

[31] E. Bozdag, A. Mesbah and A. van Deursen: *Performance testing of data delivery techniques for AJAX applications*. Journal of Web Engineering, 2009

[32] E. Bozdag and A van Deursen: *An Adaptive Push/Pull Algorithm for AJAX Applications*. Third International Workshop on Adaptation and Evolution in Web Systems Engineering (AEWSE'08), 2008

[33] M. Pohja: *Server Push with Instant Messaging*. Proceedings of the 2009 ACM symposium on Applied Computing, 2009

[34] P. Saint-Andre: *Extensible Messaging and Presence Protocol (XMPP): Core*. Proposed standard, IETF, October 2004.

[35] Weswit Srl: *Lightstreamer whitepaper*. `http://www.lightstreamer.com/Lightstreamer_WhitePaper.pdf`, visited 02.02.2011

[36] Caplin Liberator. `http://www.freeliberator.com`, visited 21.02.2011

[37] J. Garrett: *Ajax: A New Approach to Web Applications*. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`, visited 20.01.2011

[38] Sun Microsystems: Java Applets. `http://java.sun.com/applets/`

[39] Adobe Systems Inc: Flash. `http://www.adobe.com/products/flash/`

[40] A. Russel: *Comet: Low Latency Data for the Browser*. `http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/`, visited 17.02.2010

[41] D. Crane, P. McCarthy: *Comet and Reverse Ajax: The Next-Generation Ajax 2.0.* Apress, 2008

[42] Colliding Comets: Battle of the Bayeux. `http://cometdaily.com/2008/02/07/colliding-comets-battle-of-the-bayeux-part-1/`

[43] Eugene Letuchy: *Facebook Chat*. `http://www.facebook.com/note.php?note_id=14218138919&id=9445547199`, visited 15.03.2011

[44] Meebo Chat. `http://www.meebo.com/`, visited 15.03.2011

[45] Netscape: *An Exploration of Dynamic Documents*, 1996

[46] S. Gundavaram: *CGI Programming on the World Wide Web*. O'Reilly, March 1996

[47] World Wide Web Consortium (W3C): *W3C Working Draft 19*. `http://www.w3.org/TR/2009/WD-XMLHttpRequest-20091119/`, visited 2.09.2010

[48] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach and T. Berners-Lee.: *Hypertext Transfer Protocol – HTTP/1.1*, RFC2616, June 1999

[49] T. Berners-Lee, R. Fielding and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*, RFC3986, January 2005

[50] A. Russell, G. Wilkins, D. Davis and M. Nesbitt: *Bayeux Protocol - Bayeux 1.0.0*, The Dojo Foundation, 2007

[51] K. Birman and T. Joseph: *Exploiting virtual synchrony in distributed systems*. Proceedings of the eleventh ACM Symposium on Operating systems principles, 1987

[52] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.M. Kermarrec: *The Many Faces of Publish/-Subscribe*. ACM Computing Surveys, June 2003

[53] D. Crockford: *The application/json Media Type for JavaScript Object Notation (JSON)*, RFC4627, July 2006

[54] G. Wilkins: *Jetty Continuations*, `http://docs.codehaus.org/display/JETTY/Continuations`, visited 20.01.2011

[55] Sun Microsystems: *Java Servlet 2.5 Specification MR2*, July 2007

[56] Sun Microsystems: *Java Servlet 3.0 Specification*, January 2011

[57] Sun Microsystems: *JSR 51: New I/O APIs for the Java Platform,* May 2002

[58] Sun Microsystems: *Sun GlassFish Enterprise Server v3 Prelude Developer's Guide*, 2008

[59] J. Arcand: *Atmosphere Framework White Paper*, Version 0.6

[60] J. Postel: *Transmission control protocol (TCP/IP)*, RFC761, September 1981

[61] M. Stonebraker: *The Case for Shared Nothing*. Database Engineering, Volume 9, Number 1, 1985.

[62] F. Buschmann, R. Meunier, H. Rohnert and P. Sommerlad: *A System of Patterns: Pattern-Oriented Software Architecture.*, Wiley, 1996

[63] Object Management Group. Unified Modeling Language (UML). `http://www.uml.org/`, visited 24.01.2011

[64] W.J. Brown, R.C. Malveau, H.W. McCormick and T.J. Mowbray: *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley, 1998

[65] Microsoft .NET Framework 4.0. `http://www.microsoft.com/net/`, visited 15.03.2011

[66] P. Pirkelbauer, Y. Solodkyy, B. Stroustrup: *Report on language support for Multi-Methods and Open-Methods for C++,* 2007

[67] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995

[68] R. C. Martin: *Agile Software Development, Principles, Patterns, and Practices.* Prentice Hall, 20002

[69] Unicode, Inc*: The Unicode Standard (6.0 edition)*, `http://www.unicode.org/versions/Unicode6.0.0/`, visited 20.01.2011

[70] Microsoft: *Event-based Asynchronous Pattern*. `http://msdn.microsoft.com/en-us/library/wewwczdw.aspx`, visited 11.12.2010

[71] Microsoft: *I/O Completion Ports*. `http://msdn.microsoft.com/en-us/library/aa365198(VS.85).aspx`, visited 11.12.2010

[72] A. Jones, J. Ohlund: *Network Programming for Microsoft Windows, Second Edition*. Microsoft Press, 2002

[73] N. Kew: *The Apache Modules Book: Application Development with Apache*. Prentice Hall PTR, 2007

[74] K. Schaefer, J. Cochran, S. Forsyth, R. Baugh, M. Everest and D. Glendenning: *Professional IIS 7*. Wrox, 2008

[75] M. Welsh, D. Culler and E. Brewer: *SEDA: An Architecture for Well-Conditioned, Scalable Internet Services*, Eighteeth Symposium on Operating Systems Principles, 2001

[76] JSON.NET, `http://json.codeplex.com/`. Visited 19.09.2010

[77] Teletrader Software AG. `http://www.teletrader.com`. Visited 19.09.2010

[78] TeleTrader HTTP Push Service 1.0 - Internal Technical Specification.

[79] TeleTrader Market Data Server 4.3- Internal Technical Specification.

[80] Microsoft Technet: Windows Performance Monitor. `http://technet.microsoft.com/en-us/library/cc749249.aspx`, visited 10.08.2010

[81] Amazon Elastic Compute Cloud (Amazon EC2), http://aws.amazon.com/ec2. Visited 10.08.2010

[82] G. Wilkins: *20,000 Reasons Why Comet Scales*. `http://cometdaily.com/2008/01/07/20000-reasons-that-comet-scales/`, visited 20.01.2011

[83] G. Wilkins: *CometD 2 Throughput vs. Latency*. `http://blogs.webtide.com/gregw/entry/cometd_2_throughput_vs_latency`, visited 20.01.2011

[84] World Wide Web Consortium (W3C): *The Web Sockets API Working Draft 22*. `http://www.w3.org/TR/2009/WD-websockets-20091222/`, visited 26.01.2011

[85] P. Lubbers, B. Albers and F. Salim: *Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development*, Apress, 2010