TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

MASTERARBEIT

A JavaScript API for an eXtensible Virtual Shared Memory(XVSM)

Ausgeführt am

Institut für Computersprachen
Abteilung für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter der Anleitung von

Ao. Univ. Prof. Dipl.-Ing. Dr. Eva Kühn

durch

Lukas Lechner

Pezzlgasse 8/6
A-1170 Wien

Wien, Februar 2008

Abstract

This thesis presents a solution to coordinate distributed web applications using XVSM (eXtensible Virtual Shared Memory)[17], a new middleware application based on the shared data space paradigm. XVSM offers various advantages to distributed applications, reducing coordination complexity and enhancing performance. The goal of this thesis is to combine the advantages XVSM offers to distributed applications with the advantages web applications offer over traditional desktop applications. Furthermore communication restrictions of current web applications are avoided by the use of Bayeux as transport mechanism. A new protocol, called XVSMP/Bayeux is developed to regulate the communication between the web application and the XVSM. A web server component and a JavaScript client library, implementing the protocol, are created. The server component provides the web application with access to the XVSM. The JavaScript library offers a simple API to facilitate the development of distributed web applications. Operations on the XVSM can be performed using the methods of the API. The API makes the XVSMP/Bayeux protocol and the message exchange with the XVSM transparent to the web application developer. The protocol messages are exchanged through a new transport mechanism called Bayeux. Bayeux was designed to overcome current communication restrictions web applications suffer due to the nature of the HTTP protocol. This work also includes two sample applications to demonstrate the use of this technology.

Kurzfassung

Diese Masterarbeit präsentiert eine Methode zur Koordinierung von verteilte Webanwendungen mit XVSM (eXtensible Virtual Shared Memory)[17], eine Middleware basierend auf dem Prinzip von verteilten "Data Spaces". Der Einsatz von XVSM bringt verteilten Anwendungen diverse Vorteile von einer Verringerung der Komplexität der Koordinierung bishin zu erhöhter Performance. Das Ziel dieser Arbeit ist es, die Vorteile von XVSM für verteilte Anwendungen mit den Vorteilen von Webanwendungen gegenüber herkömmlichen Desktopanwendungen zu vereinen. Darüber hinaus werden Einschränkungen hinsichtlich der Kommunikationsmöglichkeiten von aktuellen Webanwendungen durch den Einsatz von Bayeux als Transportmechanismus vermieden. Die Kommunikation zwischen den Webanwendungen und XVSM wird durch ein neu entwickeltes Protokoll namens XVSMP/Bayeux geregelt. Weiters wird eine Webserverkomponente und eine JavaScript Bibliothek, welche das Protokoll implementieren, erstellt. Die Webserverkomponente ermöglicht Webanwendungen den Zugang zum XVSM. Die JavaScript Bibliothek bietet eine einfache Programmierschnittstelle um die Entwicklung von verteilten Webanwendungen zu beschleunigen. Um eine Operation auf XVSM auszuführen, genügt es die entsprechende Methode der Programmierschnittstelle aufzurufen. Die JavaScript Bibliothek macht das Protokoll und den Nachrichtenaustausch mit XVSM transparent für den Entwickler der Webanwendung. Der Nachrichtenaustausch des Protokolls erfolgt über den neunen Transportmechanismus Bayeux. Bayeux wurde entwickelt, um die momentanen Einschränkungen, welchen Webapplikationen aufgrund des HTTP Protokolls unterliegen, zu umgehen. Diese Arbeit inkludiert zwei Beispielapplikationen welche die entwickelten Techniken demonstrieren.

Contents

1 Introduction

In 2007 the number of computers connected to the Internet passed the 500 millions mark[15] and it keeps growing at an extremely high rate. As mobile devices (like mobile phones, PDAs and even digital cameras) are more and more capable of connecting to the Internet, the overall number of connected devices will grow even faster. With such a great number of potential users, distributed applications are gaining a lot of importance. For its users, distributed applications offer new services and ways of interaction. For developers, on the other hand, distributed applications impose a lot of new challenges. Distributed applications need to cope with heterogeneous hardware, various operating systems and different connectivity. Processes in distributed applications need to be coordinated and failures of single nodes must not affect the overall system. With mobile devices participating in distributed applications complexity further increases as they constantly change their locality and connectivity.

Many of these challenges for distributed applications are very well addressed by middleware systems based on the shared data space paradigm. Shared data spaces provide means for easy and efficient coordination of distributed applications. They offer a high level of abstraction by decoupling the distributed application participants in time, space and reference. This leads to reduced complexity and easier application development [2]. This thesis uses the new space based middleware XVSM (eXtensible Virtual Shared Memory) [17] developed at the Institute of Computer Languages at the Vienna University of Technology. Additionally middleware systems offer value-added services like naming services or security features. They also try to address the heterogeneity of the user's hard- and software by using platform independent programming languages like Java, distributions for different operating systems and language bindings for different programming languages. However, another piece of software has been even more successful in making applications ubiquitously available on every hard-

and software: the web browser. A web browser is nowadays available on nearly every computing device and standard applications are more and more ported towards web applications, or at least accompanied by a web application equivalent.

This thesis tries to combine the advantages of the ubiquitously available web browser with the advantages of the middleware system XVSM. It aims at creating distributed web applications which coordinate themselves through a common shared data space. The user can thus benefit from a distributed application that she/he can run from every computing device connected to the Internet without any installation required. By using modern concepts of web application development and exploiting the HTTP protocol in a way to provide bidirectional asynchronous communication, distributed web applications can match modern desktop applications concerning reactivity and usability. The thesis consists of three components. The main building block is a newly developed protocol regulating the communication between the web applications and the space. This protocol is then implemented by a server component, translating the protocol messages into operations performed on the space. The third component is a JavaScript API, which facilitates the creation of web applications by providing a simple interface to access the space. As a proof of concept two examples of use, a chat application and a space monitoring application are presented. The chat application represents a distributed multi-user web application, which shall benefit from the asynchronous bidirectional communication. The second example of use displays the current state and content of the space. This application can be used to monitor the space and shall help in the creation and debugging of other distributed web applications.

1.1 Overview

Chapter 2 gives an introduction to the technologies used in this thesis. It will focus on XVSM, Comet, the Bayeux protocol and Dojo. Chapter 3 presents the general system architecture and communication flow. Chapter 4 introduces the XVSMP/Bayeux protocol, the implementation on the server side and the JavaScript API. Chapter 5 shows the two examples of use and Chapter 6 gives an evaluation of the presented work including a benchmark and an outlook on further enhancements. Chapter 7 concludes this thesis.

2 Technical Bases

This chapter gives a short introduction to the technologies and mechanisms used in this master thesis. The first section gives an introduction to middleware systems with focus on shared data spaces. The second section describes the middleware XVSM (eXtensible Virtual Shared Memory). The third section introduces the concept of asynchronous communication for web applications followed by the section describing the Bayeux protocol and its underlying transports. The last section gives a short overview of the Dojo toolkit.

2.1 Shared Data Spaces

Although distributed applications offer a lot of advantages and new possibilities, they also introduce a high level of complexity to software developers. The heterogeneity of the underlying hard- and software makes the creation of distributed systems a difficult task. Today distributed applications are often designed to run even on mobile devices. With mobile devices a new bulk of hard- and software systems need to be dealt with. Additionally the location and the connectivity of these devices are no more static but constantly changing. This imposes new challenges and additional complexity to the creation of distributed systems. To address this complexity, developers created middleware systems.

A middleware is a piece of software located between the native operation systems network environment and the distributed application (see figure 2.1). It is designed to mask the heterogeneity of the different hardware and operating systems, by providing a uniform high-level interface (API) to the developers. By using this API the developers

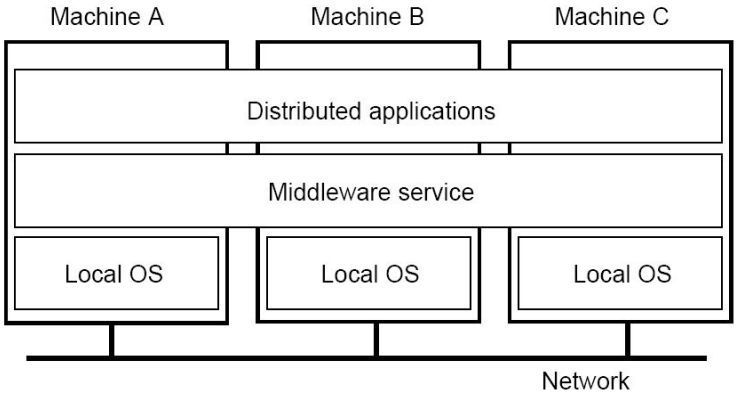

Figure 2.1: The middleware layer [25, pp. 3]

can concentrate on the application problem and no longer have to deal with different operating or hardware systems. Additionally a middleware might provide value-added services such as naming, transactions or security which further eases the development of distributed systems. Today a wide range of middleware systems based on different communication and coordination concepts exist. Middleware based on the shared data space paradigm has proven to provide a very elegant and simple to use solution for creating distributed applications [5].

Shared data spaces

The idea of a shared data space was first created by David Gelernter in the 1980s [14]. He introduced a coordination language called Linda which operates on an abstract computation environment called 'tuple space'. Concurrent processes of a distributed application coordinate themselves by communicating with the tuple space. Coordination is performed by writing and reading data tuples to/from the space (e.g. the master-worker pattern can be easily realised by letting the master process write the tasks to the space and one or multiple worker processes read the tasks from the space). The communication takes always place between the processes and the space. This way the sending process does not need to know about the receiving process and there is no need for both processes to be connected at the same time. This decoupling of processes in both time and space takes away a lot of complexity in creating distributed applications. Gelernter calls this communication paradigm which is both decoupled in space and time 'generative communication'. Linda offers a very simple interface to

operate on the tuple space consisting of only three operations:

- out(): Writes a tuple to the space
- in(): Withdraws a tuple from the space
- rd(): Reads a tuple without withdrawing it

The *out(t)* operation writes the tuple t to the space. The *in()* operation uses template matching to withdraw tuples from the space. If a template m matches a tuple t in the space, *in(m)* withdraws this tuple from the space and returns it. If there is no matching template found in the space the *in(m)* operation blocks until a matching template is found (following the example from above a worker process would issue a *in()* operation and if no task is present within the space it blocks until a new task gets available). The *rd()* operation acts just like the *in()* operation but the matching tuple remains in the space. These three operations were later extended by an additional *in()* and *rd()* method without a blocking behaviour called *inp()* and *rdp()*. If no matching tuple is found these non-blocking operations immediately return zero.

2.2 XVSM

eXtensible Virutal Shared Memory (XVSM) is a new middleware based on the paradigm of shared data spaces. It offers new advantages to developers of distributed systems, such as multiple synchronization methods, a range of different data access types, a flexible architecture which can be easily tuned to the application specific needs or a protocol as an interface to the space to achieve interoperability. XVSM was developed with the vision to make development of distributed applications as simple as the development of a single user, single threaded program [17].

2.2.1 Architecture

The XVSM architecture consists of three layers (see figure 2.2). The XVSM Kernel layer is the main component of the XVSM. It includes the basic components needed to operate a single server space. The XVSM Extensions are layered on top of the kernel and provide additional services (e.g. lookup mechanism, distribution of the space, persistency, etc.). These features can be added to the Kernel in a way which is completely transparent to the distributed applications. The interface of the XVSM is defined by the XVSM Protocol layer. It defines protocols to communicate with the XVSM using standard transport mechanisms. Thus interoperability between applications, kernels and extensions even of different implementations is achieved.

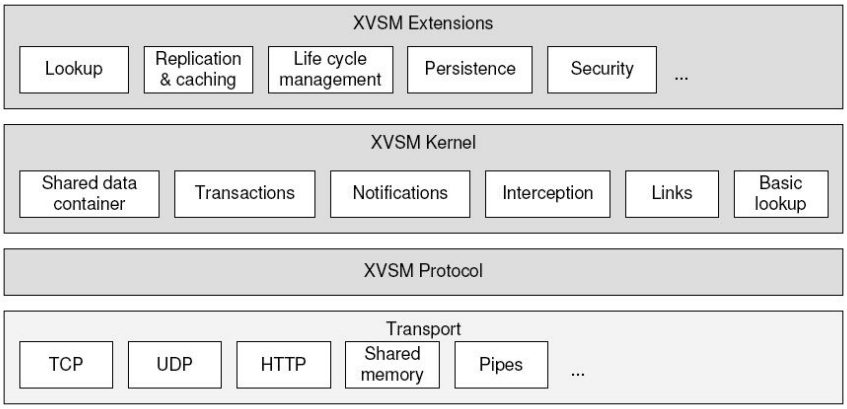

Figure 2.2: The layered XVSM architecture [17, pp. 4]

2.2.2 Data Access and Synchronization

To store data and synchronize processes XVSM uses data containers. In the space data can only exist within a data container and is organized in so called entries. A process might create, destroy, write data to or read data from a container. Every data container is assigned a unique container reference, a data access type, a size (defining the number of entries it can hold) and optionally a name. Together with the lookup mechanism a named container can be accessed by multiple applications making coordination and data exchange possible. An entry can hold different data types. So far strings, numbers, container references or tuples of entries are supported.

The ability to store container references within an entry allows building more complex data structures.

XVSM extends the synchronisation model of Linda to support a blocking behaviour for write operations and introduces a new operation called shift. It now offers four operations to manipulate data: read, take, write and shift. The read and take operations act like their Linda equivalents read and in. The write operation tries to write to the container, but blocks if the container cannot accept another entry, e.g. because it reached its size limit. Shift is equivalent to write but instead of entering a blocking state it replaces exiting entries in the container. The read, take and write operation take a timeout parameter to define how long the operation shall block. If the timeout is reached an exception is thrown.

XVSM supports different data access (coordination) types for a container which allows application developers to choose the appropriate one depending on the needs of their applications. The data access type defines the behaviour of the XVSM operations on the container. The following data access types are available:

- Linda: Uses Linda-like template matching on all operations. A template can be associated to every entry passed to write and shift operations. Read and take operations can be given an additional parameter containing templates (e.g. a read/take operation will return/remove entries matching the template only).

- FIFO: Performs operations like on a queue. Read/take operations return elements from the beginning of the queue, write/shift operations append elements to the end of the queue.

- Set: The selection of entries in a set is done using a non-deterministic way for all operations.

- Map: This data access type implements a 'key:entry' mapping. Every operation takes a 'key/entry' pair as additional parameter.

- Vector: Provides access to entries similar to an array.

The data-containers in a space are disjoint. This means an operation on a container can never directly affect data in another container. This partitioning of the space enhances parallelism as concurrent operations on different data containers can be executed without further need for synchronization. Additionally, selective operations like template matching on a Linda container need not be performed on an entire space but a single container. It also adds to the logical understanding as different applications will use different containers and need not worry about the others.

2.2.3 Notifications

XVSM uses notifications to inform clients about changes at the XVSM in an asynchronous way. Clients can in advance declare their interest in a specific event by creating a notification on the space. If at a later stage such an event occurs the client is subsequently informed about it. XVSM supports notifications at a very granular level. A client might specify interest in read, write, take or shift operations on a specific container. It is even possible to specify only interest in operations with specific value (e.g. a write operation that matches a certain template).

2.2.4 XVSM Protocol

The XVSM Protocol layer (XVSMP) defines a protocol based on the Extensible Markup Language (XML) to access the XVSM. Using a standardized protocol to operate on the XVSM, interoperability between all components respecting the interface can be achieved. Additionally the XML-based protocol can be used with different underlying transport mechanisms. In this thesis an addition to this layer will be proposed by defining a protocol based on the Bayeux protocol (see section 2.4) which is using JavaScript Object Notation (see section 2.4.1) as message encoding format.

2.2.5 XVSM Extensions

There are two different possibilities to extend the functionality of XVSM: extension libraries and extension services. The extension library resides within the client application's computer and provides the client with additional functionality. It uses a standardized interface to provide its functionality to the client and communicates with the space using the XVSM Protocol. The extension service on the other hand resides within the space's infrastructure and directly enhances its functionality. An extension service uses an interception technique to integrate into the space's operation. The service registers on the space to intercept certain operations (e.g. a read operation) at a specific granularity level (e.g. all read operation, read on a specific container or even a specific read operation). Every time a registered operation is executed the service is notified. Interception of operations can be asynchronous (the space continues in executing the operation) or synchronous (the space waits for the service to complete its task before continuing the execution of the operation).

2.2.6 Implementation

This thesis uses the reference implementation of the XVSM system called MozartSpaces (version 0.9.0.1), which is currently developed at the Institute of Computer Languages at the Vienna University of Technology. MozartSpaces is implemented in the Java programming language. The core component of MozartSpaces is called the XVSM Core which comprises the functionality of the XVSM Kernel Layer. The Core offers a Java Interface called the Core API (CAPI) which provides all methods to operate on the space. For internal data storage the core uses the embedded database Derby. The XVSM Core itself can already be used as embedded library in any Java application. However, MozartSpaces also provides a server solution which provides access to the XVSM core via the XVSM Protocol. Figure 2.3 shows the different installations of the MozartSpace implementation. Figure 2.3.a shows the XVSM core and its components. Figure 2.3.b shows the XVSM core embedded into a web server to provide access using the XVSM Protocol. It uses an additional component called XVSM-Servlet which

implements the XVSM Protocol. It is based on the Java servlet technology[1].

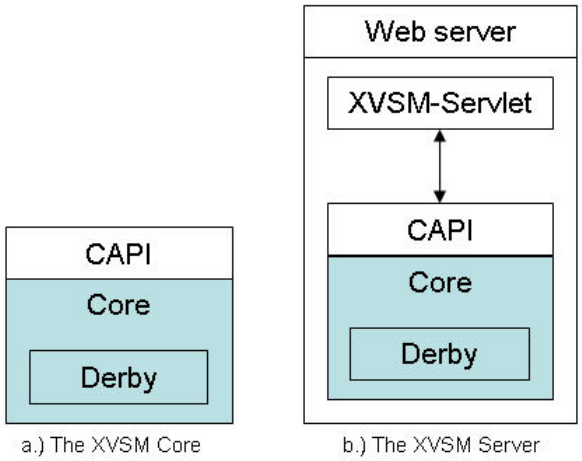

Figure 2.3: The MozartSpaces implementations[20].

2.3 Comet

2.3.1 Introduction

In the beginning of the World Wide Web (the WWW) Internet Browsers were only able to display static content. Web pages were electronic copies of text for the purpose of distribution. These pages were mainly used by scientists, and there was no commercial use of the WWW at that time. With the increasing popularity of personal computers however, the number of people with access to this new media grew very fast. Soon companies discovered the business potential of the Internet. In order to exploit the World Wide Web as a new platform to do business transactions, web pages needed to evolve from static text to dynamically changing pages, interacting with users and servers at the same time [4, p.3].

Soon many different technologies which offered dynamic behaviour to web pages (thereafter referred to as web applications) were developed. All of these technologies can

[1]Servlets are Java objects within a web server which are linked to a specific URL. Requests to this URL are processed and answered by the Servlet. See http://java.sun.com/products/servlet for details.

be categorized into being either server-side or client-side, with respect to where the actual computation takes place.

Server side technologies

The main characteristic of server side technologies is, that all the processing is done by the server. The Internet browser is only used to display static pages and to convert user actions (clicks on buttons, or links) into requests sent to the server. Due to this even small user actions with very little or no impact on the displayed site (e.g. adding an item to the virtual shopping cart) causes the entire page to be retransferred from the server to the browser. Nevertheless server side technologies soon became very popular because it is much easier to develop and maintain software for a single server, than for the different browsers running on the user's computers.

Client side technologies

Client-side technologies refer to programs which are executed within the user's browser environment. They are either scripts embedded into web pages, or binary programs which are downloaded and afterwards executed. Scripts need to be supported by the browser itself, whereas binary programs need a software-part (usually a plug-in) to be installed on the user's computer before they can be executed (e.g.: Macromedia Flash or Java Applets). Although binary programs are generally more powerful and efficient (because the code can be preliminarily optimized and compiled), the required installation of additional software is a big disadvantage. Especially in enterprise computer setups the installation of additional software is often disabled for security reasons. Scripts on the other hand are transferred as source code from the server to the client and interpreted by the browser. One of the first available and still very popular client-side scripting languages is JavaScript. When first introduced client-side scripting with JavaScript was restricted to simple operations like user-input validation. With the introduction of the Document Object Model (DOM) however, it became possible to create dynamic web applications with JavaScript. The DOM offers the possibility to navigate and modify an HTML[2] document from within a JavaScript program. This eliminates the need of transmitting a new page to change the look of a web page. Instead, the web page is changed dynamically by the JavaScript program.

[2]HTML - HyperText Markup Language

Traditional web applications

In traditional web applications user requests (e.g. adding an item to the virtual shopping cart) are directly mapped to requests sent from the browser to the server. This includes sending the data from the browser to the server, processing of the data at the server, regenerating the website (now including the item in the shopping cart), sending the website to the user's browser and displaying the retrieved page. During this procedure the user can not interact with the web application, but has to wait for its completion. Figure 2.4 shows the interaction of traditional web applications (the gap in the user activity line represents the forced waiting). This is a very slow and inconvenient solution, very different from the interactive user-friendly interfaces offered by modern desktop applications. The goal of Ajax applications is to close the gap between desktop and web applications concerning interactivity and user-friendly interfaces.

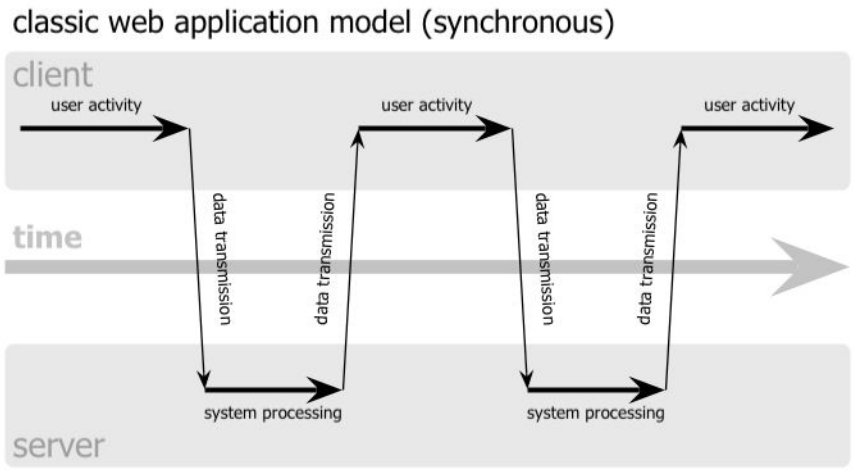

Figure 2.4: Traditional web application model [13].

2.3.2 Ajax

The term "Ajax" was coined by an information architect called Jesse James Garret in February 2005 to describe a technique for building rich and interactive browser based applications [13]. From a technical point of view Ajax is not a technology by itself, but rather a methodology to use the existing technologies in a way to go beyond the

possibilities of traditional web applications. All the technologies Ajax uses (which are mainly HTTP, JavaScript and XML have already existed for some years. It is the way these technologies are used and the way they mesh together which is new. Ajax is short for 'Asynchronous JavaScript and XML'. The name indicates that Ajax uses JavaScript as the client side programming language, XML as the data transfer format and asynchronous communication to transport the data from server to client. Today however, the term Ajax is often generally used for every dynamic web application using asynchronous communication to transfer data from server to client, no matter which programming language or data format used.

From a software engineering perspective Ajax introduces an additional software layer between the user interface of the Internet browser and the communication of browser and server. Figure 2.5 shows the application model of Ajax web applications. In

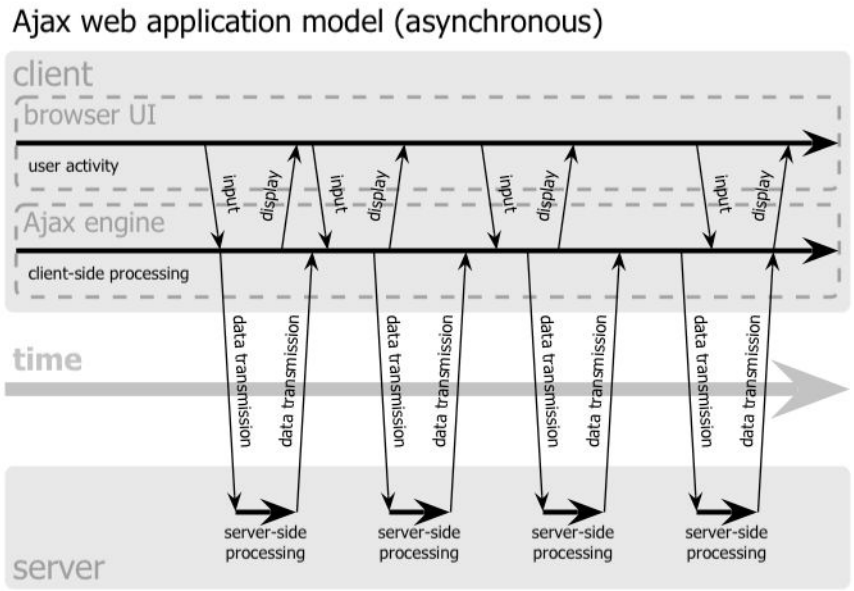

Figure 2.5: Ajax web application model [13].

Ajax applications user actions are processed by a JavaScript program, which is often referred to as the 'Ajax Engine'. The Ajax Engine can then send the request on the user's behalf. However it can also prefetch the data, so that it is already locally available, as soon as the user needs it. Or it might show new information to the user on its own, because the server informed the Ajax Engine about changes on the page. The 'Ajax Engine' decouples the user interface from the client-server communication.

There are no longer any interruptions of user activity. Even after a user request the Ajax application stays responsive and fully functional (e.g. the user might add an item to the shopping cart and continue searching the page for other items). The communication is performed asynchronously in the background by the Ajax Engine and the user interface changes (e.g. the item is added to the shopping cart and the new price is displayed) when the data from the server arrives. Additionally the amount of data to be transferred is reduced to the actual item that has to be changed in the page. There is no longer any need to transfer an entire web site with every request. This decoupling of the user interface from the client-server communication alongside the use of efficient asynchronous communication allows creating reactive user interfaces with a feeling similar to desktop applications.

2.3.3 Communication

Ajax uses asynchronous communication to transfer data in the background of a web application. There are two different ways to do this:

- Hidden Frame - IFrame

  The Internet browser 'Netscape Navigator 2.0' first introduced the concept of frames for web sites. The idea is to split up the display of a web page into several frames, each containing a different HTML document. To retrieve its document, each frame can issue a request to the server on its own. By defining a frame with a height or width of zero, it is possible to create a 'hidden frame' not visible to the user. This frame is then used to retrieve data in the background. A JavaScript function can induce the frame to send a request to the server and the frame issues an event which can be associated with a JavaScript method when the data returns. With HTML 4.0 the concept of frames was officially included into the HTML standard. A new element called 'IFrame' was introduced as well. Frames can not be dynamically created within a web site, whereas IFrames can be programmatically integrated into a web site like any other HTML element. Hidden Frames and IFrames were the first mechanism to support asynchronous communication for web browsers [31], [8].

- XMLHttpRequest

  The use of (hidden) frames for asynchronous communication was not originally intended, however the popularity of this technique showed the need to support asynchronous communication for web browsers. Hence the XMLHttpRequest object was introduced to the JavaScript language. It allows issuing an HTTP request from within a JavaScript program with full control over HTTP headers and status codes.

2.3.4 Server side information push

Both frames and the XMLHttpRequest object enable a web application to send requests to a server in the background (asynchronous), while remaining responsive to user interaction. Requests to the server are sent using the Hypertext Transfer Protocol (HTTP) [10]. HTTP was originally designed to follow a strict request-response pattern, with the browser sending the request and the server responding. The possibility of the server pushing information to the browser on its own was not thought of. In multi-user web applications however, the need of a server to inform clients about events arises very often (e.g. a chat application where the server wants to inform the clients about a new message). Following the specification of the HTTP protocol the server cannot open a connection to a client by itself, but the client has to open the connection and to look for new information on the server. This leads to 4 different methods of pushing data from the server to the client, which differ in their degree of asynchronism. Figure 2.6 shows the different solutions:

Page Refresh

Page refresh is simplest way to transfer new information from the server to the client. It is user initiated and synchronous. The user has to click the refresh button of his/her browser application. The website is then requested from the server once again. If the information at the server has changed in between the old and the new request, the new website will contain the updated content. This is the way traditional web applications work.

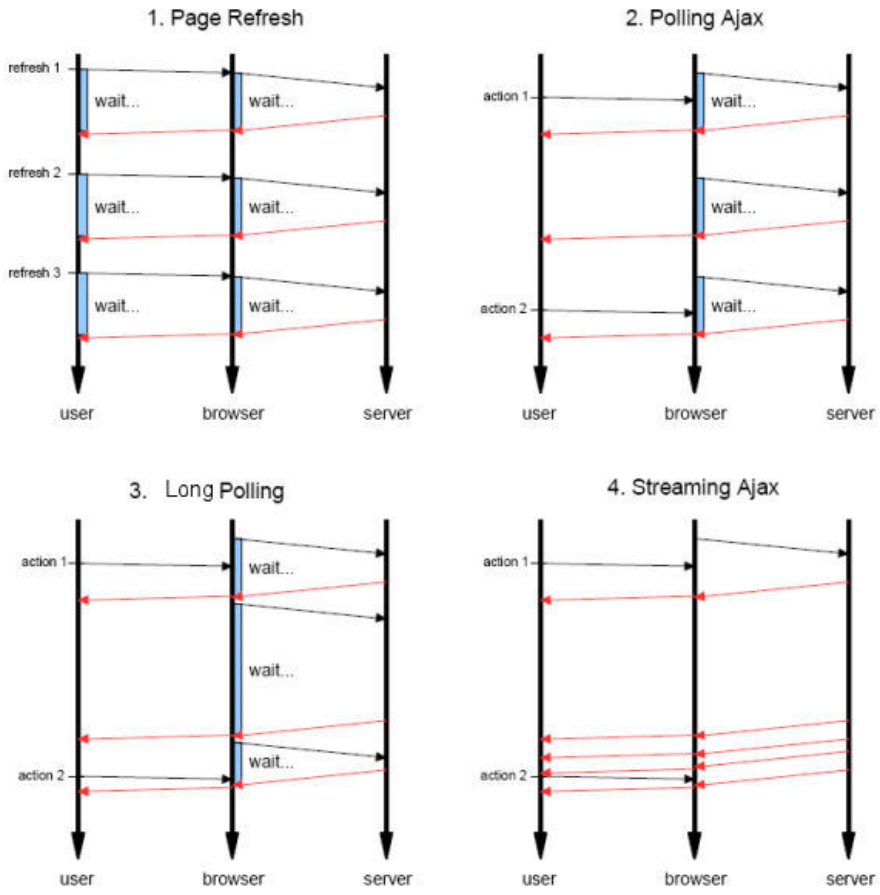

Figure 2.6: The different solutions to push information from the server to the client [3].

Polling Ajax

Polling Ajax is also referred to as classic Ajax. With Polling Ajax the Ajax Engine continuously polls the server for new information at a given interval. If new information is available, the Ajax Engine transfers the information to the user interface. This approach is asynchronous from a user's perspective (the fact that the Ajax Engine is polling for the information is hidden from the user), but synchronous from the Ajax Engine's perspective. However there is a trade-off between the latency of the information arrival and the polling overhead. A short polling interval reduces latency, but increases data transfer overhead as many requests are sent without any new data available.

Long-Polling

To reduce latency and data transfer overhead, long-polling is applied. With long-polling every polling request is maintained (the response is withheld by the server which keeps the connection open) at the server side until some data for the client is available. If some data becomes available, the response is sent and the client can start a new polling request. This approach minimizes latency and reduces overhead. The server can always respond immediately when new data arrives. However, the overhead of a complete TCP connection build-up and tear-down for every transferred piece of data remains when using version 1.0 of HTTP. This can be avoided when using HTTP version 1.1, which allows the reuse of a TCP connection for multiple HTTP requests, leading to more efficient use of long-polling [10].

Streaming Ajax

Streaming Ajax tries to avoid the TCP connection build-up and tear-down as well as the resending of HTTP requests by reusing an open HTTP request to send data whenever it becomes available. To do this, Streaming Ajax takes advantage of the progressive rendering feature of frames (IFrame or Hidden Frame, see 2.3.3). When a frame tries to retrieve its content, it progressively renders the arriving data. It does not wait for the entire document to arrive before starting to process it. This feature is exploited by Streaming Ajax. The client sends a HTTP request to the server which is maintained open. Whenever some data for the client is available, the server marshals the data into a JavaScript block and sends it using the open connection. The retrieving frame on the client side immediately interprets the retrieved script, but also continues to wait for the rest of the page. The script usually contains a method call informing the web application of the data arrival.

Both Long-Polling and Streaming Ajax provide the server with the ability to directly notify the client of an event (e.g. some data that arrived). These two transport mechanisms have been named Comet by Alex Russel in his article 'COMET - the next stage of Ajax' [23]. To ease the understanding, the term 'Comet applications' is used for Ajax applications using Long-Polling or Streaming Ajax as their transport mechanism throughout the rest of this thesis. Applications using standard polling are referred to as Ajax applications. Figure 2.7 extends the diagram showing the classic

Ajax application model to match the concept of the Comet application model. It introduces a comet-event-bus which handles the communication with the clients. If an event occurs at the server, it is passed to the comet-event-bus, which pushes the event to the specific web application. Thus with Comet the concept of event-driven applications is brought to the web.

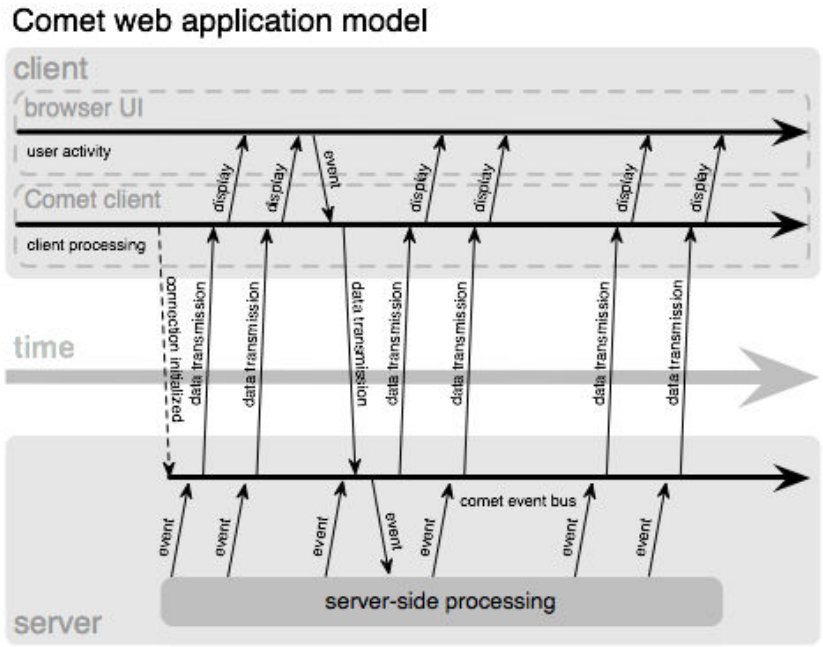

Figure 2.7: Comet web application model. Source[23]

In his article about Comet, Phil Windley says *"Ajax is about me"* [30] (referring to classic Ajax) to illustrate that Ajax applications are suitable solutions for single-user web applications that only interact with the server and are not affected by other users using the same application. When it comes to multi-user interaction, the server needs the possibility to notify a user about events generated by other users. This is only possible with a true server push solution like Comet. In space based computing with Comet the space can directly notify multiple users connected through web applications about changes within the space.

2.3.5 Server requirements

Although Comet offers a lot of advantages for web applications it imposes new requirements upon the server infrastructure. Current web servers are optimized for the load profile of classic web applications. In classic web applications a burst of requests is followed by pauses, assuming the user is first retrieving and then reading the page. While the user is reading the page, she/he does not use any of the web server's resources (the connection is closed, once the application transmitted the data). Web servers keep a pool of threads to treat incoming requests. Every incoming request is processed by a thread, which remains active until the request is completed. After processing the request is returned to the thread pool. In Comet applications however, a request is continuously maintained open to allow the server to push data to the client. Thus the thread remains in a blocking state until some data gets available. The one-thread-per-request model will result in a one-thread-per-user model which does not scale with an increasing number of users. The solution to this problem is to decouple the computation needed to complete the request from the actual input and output(IO) operations. This can be achieved with non-blocking IOs, which are currently developed for a lot of web servers.

The following web server products already support this concept:

- Tomcat 6.0 with NIO (Non-blocking IO extension)[3]
- Websphere Jetty (Continuations) [4]
- Lightstreamer [5]
- AsyncWeb [6]
- Glassfish (Grizzly) [7]

[3] http://tomcat.apache.org/tomcat-6.0-doc/aio.html
[4] http://docs.codehaus.org/display/JETTY/Continuations
[5] http://www.lightstreamer.com/
[6] http://docs.safehaus.org/display/ASYNCWEB/Home
[7] https://glassfish.dev.java.net/

In this thesis the web server Jetty is used as platform to host Comet powered web applications. Jetty is a web server with an included servlet container [8]. It uses a concept called Continuations to support scalability with Comet applications. It introduces a 'Continuation Object' which allows the processing thread to suspend the current request. Once suspended, the processing thread can be returned to the thread pool and start processing another request. With Continuations a thread is no longer needed for every request but only for actual sending and retrieving of data. Additionally Continuations do respect the servlet API [9]. As a consequence web applications developed for a servlet container supporting Continuations can also be deployed to any other servlet container. However without support for Continuations requests cannot be suspended and resource consumption will be significantly higher [28].

	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 2.8: Comparison: Web Server Resource Usage [29].

The Figure 2.8 shows the resource usage of different web applications. Column number one shows a classic web application. The second column a Comet application deployed on a traditional web server and the third column shows the same application deployed on a web server implementing the Continuations concept. The chart tries to estimate the number of concurrent requests issued by the web applications for

[8] The servlet container is the part of the web server where servlets are executed.

[9] The Servlet API defines the interface between the servlet container and the servlets running in a servlet container.

10.000 simultaneous users. Based on this number the required threads and memory consumption are calculated. It shows that a Comet application deployed on a web server supporting Continuations, requires little more resources than a classic web application. On a traditional web server the Comet application however dramatically degrades in performance.

2.4 Bayeux Protocol

The Bayeux Protocol [22] is a message based publish/subscribe protocol for web applications. It is designed to provide asynchronous message delivery and supports different transport solutions including Long-Polling and Streaming Ajax. The Bayeux Protocol is based on named channels. A client can publish messages to a channel, or retrieve messages from a previously subscribed channel. The intent of the Bayeux developers is to simplify the development of event-based web applications. By using an existing Bayeux server and client implementation web application developers no longer need to worry about communication matters. The Bayeux protocol supports the negotiation of the best suited transport mechanism. It also includes methods for securing communication and authentication mechanisms. The protocol is still under development and some features are not yet fully specified. Nevertheless there are already some products on the market implementing the first versions of the Bayeux protocol which are very promising (notably the web server Jetty and the Dojo toolkit).

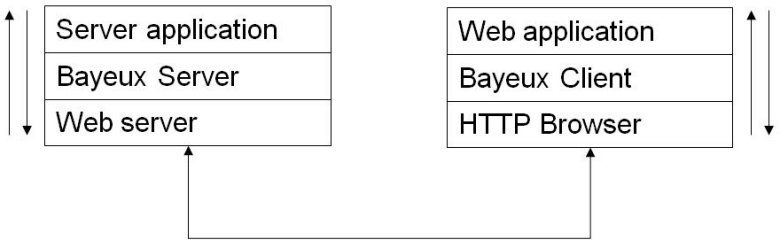

Figure 2.9: Bayeux as communication layer for web applications.

Figure 2.9 shows Bayeux as an additional layer between web application and Internet browser on one side and server application and web server on the other side. Server and web applications subscribe to channels within Bayeux to communicate with each other.

The Bayeux protocol takes care of establishing connections and sending messages. Bayeux includes an advice mechanism which allows the Bayeux server to inform its clients about the best way of communicating with the server (e.g.: following the HTTP Specification every Internet browser should open no more than 2 connections to the same server). If there is more than one Bayeux client running within an Internet browser this might lead to problems with connection numbers. The Bayeux server may then advice the client to switch to classic polling operation.

2.4.1 JSON

The Bayeux Protocol uses JavaScript Object Notation (JSON) [9] as its message exchange format. JSON is short for JavaScript Object Notation. It is a lightweight data-exchange format based on a subset of the object literal notation of the JavaScript language. Although JSON is mainly used in JavaScript applications, data structures of many other programming languages can be represented using JSON. JSON uses a very simple syntax consisting of only two constructs:

- Key/Value pairs
- An ordered list of values

A key is represented using the string data type. A value can be a string, a number, 'true', 'false', a key/value pair or an ordered list itself.

Compared to the de facto standard in data interexchange languages XML, JSON offers a few advantages for Ajax applications. It has a much simpler syntax than XML and reduces the amount of data being transfered compared to XML (due to a smaller amount of data encapsulation overhead). Additionally in web applications using JavaScript, JSON objects need not be parsed by a JSON parser, but can be directly interpreted in JavaScript (because JSON objects are syntactically legal JavaScript code). Further comparisons between JSON and XML can be found in [21],[16].

2.5 The Dojo Toolkit

Dojo is a JavaScript toolkit designed to facilitate the development of (large) JavaScript applications [11]. It is mainly targeted at Ajax empowered websites but it can also be used for any other JavaScript application, even those not residing within a browser environment. It is developed by a non-profit organization called the Dojo Foundation and distributed under the BSD Licence and the Academic Free Licence. It is supported by big companies like IBM or AOL.

Essentially Dojo is a JavaScript library, but by being extremely rich on features and offering support for developers to do more structured programming with JavaScript it is often seen as a framework or toolkit. In fact the Dojo Foundation did name it the Dojo Toolkit.

The Dojo Toolkit offers the following functions:

- A packaging system

  Dojo offers a packaging system similar to the Java programming language allowing to structure the functionality offered and to include exactly those modules needed for a specific application. Code written with the help of the Dojo Toolkit can also be put into packages and included into the packaging system.

- Widget System

  Widgets are building bricks for web applications. They are defined using HTML elements, CSS design declarations and they can be enriched with JavaScript code. They are especially useful to reuse common design patterns which appear in many web applications (e.g. a date-selection field). Dojo already provides a lot of widgets commonly used in web pages, and the widget system can be augmented by custom built widgets. The Widget System for web applications is comparable to the Java Swing Package for Desktop applications.

- IO Package

The IO Package offered is essential for every Ajax application. It includes a easy-to-use API to perform asynchronous communication and also a Bayeux Client implementation.

- DOM Package

  To easily navigate within an HTML document, add, remove or move nodes the DOM package offers a lot of convenient functions. This package is used to change the look and content of the web application.

3 System Architecture

The goal of this thesis is to facilitate the creation of distributed web applications that coordinate themselves, communicate with each other and exchange data through a common virtual space. The virtual space is created using the eXtensible Virtual Shared Memory (XVSM) described in section 2.2. In its simplest configuration the space can be run as a single application on one computer, however to increase performance and reliability it can be replicated over different connected computers, thus spanning a space in between those computers. Regarding the web application it does not matter if the space is created on a single server or on several machines. XVSM makes the replication completely transparent to the client applications and thus appears to them as a single coherent system. Figure 3.1 shows a distributed web application using one central space server, figure 3.2 shows a replicated space environment.

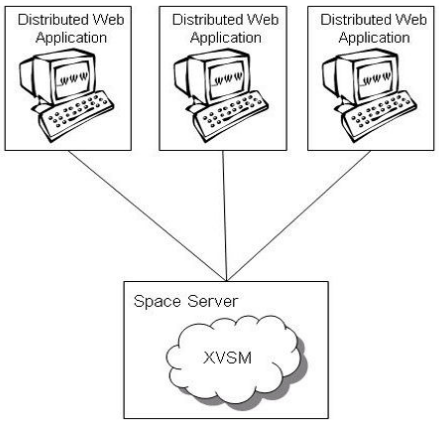

Figure 3.1: XVSM on a central server

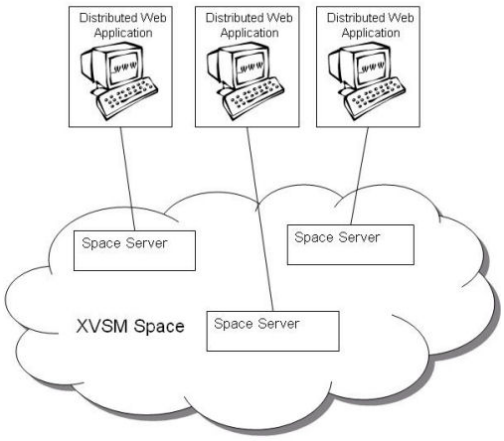

Figure 3.2: A replicated space environment

The system architecture is split up into different components on both the server and on the client side as shown in Figure 3.3. On the left side (the server part) all components

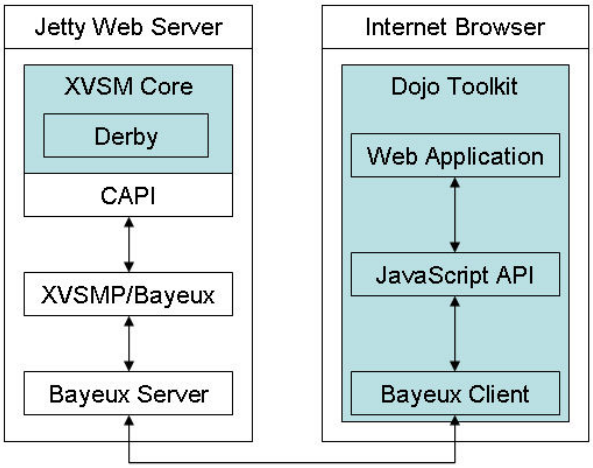

Figure 3.3: The system architecture

are run within the web server Jetty and on the right side (the client part) all components are run within an Internet browser. The two sides communicate with each other by using the XVSMP/Bayeux protocol (see 4.1). It is based on JSON 2.4.1 encoded messages on top of the Bayeux protocol, which is used as the underlying means of transport. Both sides are built upon a multi-layered design where the different layers utilise functions from the lower layers and provide functions to the upper layers. Every layer on one side is communicating with the layer of the same level on the other side by passing information through the lower levels. In the following sections I will give a detailed explaination of how the different components on both sides as well as how the communication works.

3.1 The Space Server

The server consists of three components: The XVSM space, the XVSMP/Bayeux application and the Bayeux server application. They are all executed within one servlet container. Jetty was chosen as the web server because of its support of Comet applications. The developers of Jetty also provide an open-source implementation of a Bayeux server. The Bayeux server uses a servlet to process incoming requests from clients, and provides a Java interface for interaction with other applications running in the servlet container. The XVSMP/Bayeux application connects the two parts with

each other. It receives incoming requests from the Bayeux server, interprets them and executes them on the XVSM space. Responses or notifications from the space are forwarded to the Bayeux server that distributes them to the addressed clients.

3.2 The Distributed Web Application

The client side also consists of three components that are running within an Internet browser. The Bayeux client is responsible for communicating with the Bayeux server. On top of the Bayeux client the JavaScript API is implemented. It uses the Bayeux client to send messages to the XVSMP/Bayeux application on the server side. The JavaScript API provides simple methods to perform operations on the XVSM space, which are used by the actual distributed web application. The Dojo toolkit takes care of all the functions for the Bayeux client. The JavaScript API and the sample web applications of this thesis make use of the facilities of the Dojo toolkit. Figure 3.3 shows those three components located within the Dojo frame. However, the web application can also stand alone and is then located outside the Dojo frame.

The intention of the JavaScript API is to facilitate, aid and quicken the development of distributed web applications using XVSM spaces by making the protocol details transparent to the application developer. However, other web programming languages (e.g. ActiveX, Flash ActionScript) as well as non web-applications (e.g. traditional desktop applications) can also access the space server by using a Bayeux client and having implemented the XVSMP/Bayeux protocol. Nevertheless, for non web-applications the advantage to use XVSMP/Bayeux is limited to its usability in firewall protected networks.

3.3 Communication

For communication between the two parts a protocol called XVSMP/Bayeux [18],[19] is defined. It uses the Bayeux protocol as the underlying means of transport and JSON as the message encoding format. Bayeux provides a publish-subscribe message exchange

mechanism based on named channels. However, as direct communication between web applications and the XVSM space is sufficient for this thesis, a simple request-response message exchange pattern is run on top of the publish-subscribe mechanism. Therefore a common server channel '/xvsm/server' is defined to support client to server (XVSM) communication. Messages from XVSM to the client are published to the client's private channel. Within Bayeux every client has its own private channel consisting of the string '/meta/connections' and of the client's id. Upon connecting to the Bayeux server the client is automatically subscribed to its private channel. Figure 3.4 shows the message flow. The dashed line shows the subscription of the XVSMP/Bayeux application at the Bayeux server. This operation is performed only once at the startup. The continuous lines show the normal message flow between the XVSMP/Bayeux application and one instance of the JavaScript API (note that a server will also publish to and retrieve messages from other clients). The protocol itself is described in section 4.1

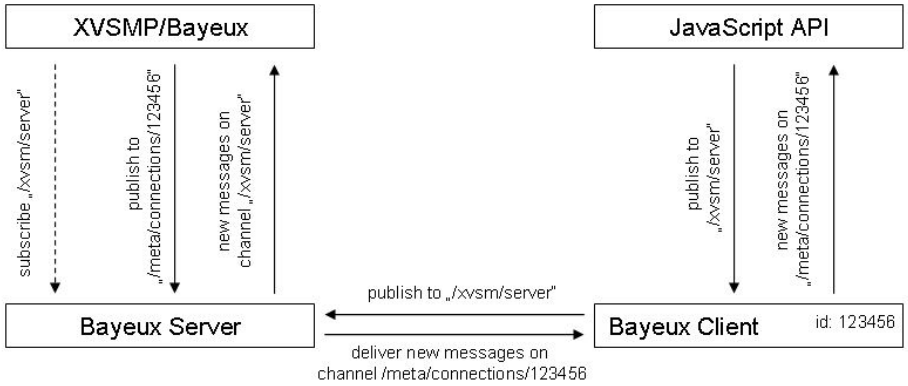

Figure 3.4: Communication between XVSM and web application

4 Design & Implementation

4.1 XVSMP/Bayeux Protocol Design

The XVSMP/Bayeux is a message-oriented protocol using Bayeux as underlying means of transport and JSON as message encoding format. Messages are represented using JSON objects. Every message contains exactly three parameters ('operation','request' and 'data'). The 'operation parameter' contains the name of the operation to be executed on the XVSM space. The 'request parameter' is an increasing number, which is used to bind a request and the corresponding response together. It is used to establish a request-response message exchange on top of Bayeux's publish-subscribe mechanism. The 'data parameter' is a JSON object itself, which's content depends on the type of the operation.

The following example (adapted from [18]) shows the required command to create a container within the XVSM space:

```
{
    "operation"     : "CreateContainer",
    "request"       : 1,
    "data"          : { "size": 10 }
}
```

The 'operation parameter' tells the XVSM to create a container, the 'request parameter' is set to the lowest number which has not yet been used in the previous communication. The 'data parameter' contains a JSON object with a 'size parameter'

of the value of 10. This message will cause the XVSM to create a container that is bounded to size 10. A corresponding response from XVSM will look like this:

```
{
    "operation"     : "CreateContainer",
    "request"       : 1,
    "data"          : { "cref": "cref_117_0" }
}
```

The 'operation' and the 'request' parameters of this message contain the same values as in the previous request. This way the client knows that this response belongs to the previously issued request. The 'data parameter' contains the reference of the created container.

This example shows how a write operation can be performed on the container created above:

```
{
    "operation"  : "Write",
    "request"    : 2,
    "data"       : { "entries":  [ {  "cref" : "cref_117_0",
                                      "type" : "STRING_UTF8",
                                      "value" : "test" }
                                 ]
                 }
}
```

Every new request from the client to the server must be given a so far unused number in the 'request parameter' (in this case the number 2). The 'data parameter' this time contains a JSON object with only one parameter called 'entries'. This parameter contains a JSON array. Each element within the JSON array represents an entry which shall be written to the XVSM. An entry is marshalled using a JSON object containing a 'cref', a 'type' and a 'value' parameter. The 'cref parameter' defines the container

the entry shall be written to. An entry can be any of the following types: string, number, container reference or tuple. The 'value parameter' contains the actual value of the entry. A tuple is a list of entries used to build up more complex data structures [18, pp. 4].

4.2 XVSMP/Bayeux Implementation

The XVSMP/Bayeux implementation is designed to run within the servlet container of the web server Jetty and uses Java as programming language. To handle JSON messages, an open-source Java implementation of JSON helper classes is used [1]. The server implementation itself is organised into two packages: org.xvsm.server and org.xvsm.server.json. The org.xvsm.server package includes three classes which are mainly responsible for connecting the XVSM and the Bayeux server. The org.xvsm.server.json package is responsible for interpreting the messages of the XVSMP/Bayeux protocol, executing the commands on the XVSM space and sending the responses.

4.2.1 The org.xvsm.server package

- The ClientListener Class

  The ClientListener class is responsible for the startup procedure. It implements the ServletContextAttributeListener interface to listen for changes of attributes in the servlet context. The Bayeux server will automatically add itself as attribute to the servlet context upon startup. The ClientListener will then start up the XVSM space and create an instance of the XVSMClient Class. The XVSMClient object is subsequently registered with the Bayeux server (it subscribes the "/xvsm/server" channel). Additionally it sets the security restrictions to the Bayeux server using the XVSMSecurityPolicy class.

- The XVSMClient Class

[1] http://www.json.org/java/index.html

The XVSMClient implements the Listener interface from the Bayeux server implementation. It is the main connection point between the Bayeux server and the XVSM space. The Bayeux server calls its deliver method whenever a client publishes a message to the '/xvsm/server' channel. The deliver method looks up which operation should be performed on the XVSM space and instantiates an object of the respective class using reflection. The operation processing is delegated to the newly created object. Every operation is handled by a specific class. Removing or adding an operation only requires removing or adding a class with the name as the operation. The XVSMClient class also provides a send method which is used by the processing classes to send answers back to the web applications.

- The XVSMSecurityPolicy Class

  This class implements the SecurityPolicy interface of the Bayeux server implementation. The SecurityPolicy can be used to regulate creation, subscription and publishing rights on channels. The XVSMSecurityPolicy class allows clients to send messages to the '/xvsm/server' channel only. Clients are not allowed to perform any other operations (creating channels, subscribing channels or publishing to other channels). It is important that no client can subscribe the '/xvsm/server' channel or another client's private channel to prevent malicious clients from eavesdropping. Additionally, it is also important to keep clients from publishing to other client's private channels so that they cannot infiltrate other communications.

4.2.2 The org.xvsm.server.json package

The org.xvsm.server.json package includes a class for every operation performed on the XVSM. They all extend the JSONObject class. Additionally, it contains a Util class which helps transforming JSONObject into XVSM Entries, Tuples or Selectors and vice versa. Whenever a message is retrieved by the XVSMClient class the operation to perform is extracted and a corresponding class from the org.xvsm.server.json package is created. All classes provide a 'doProcess' method accepting two arguments: The

operation details encoded in a JSON object and a client object (representing the Bayeux client from which the message originated). The 'doProcess' method executes the operation on the space and returns the response to the web application. In case of an error during execution the error message is packed into a JSON message and returned as well. Fatal errors, hindering the server from returning a response to the client, are written to the server's log file.

4.3 JavaScript API

The JavaScript API is developed as a module of the Dojo toolkit. It uses the Bayeux client of the Dojo toolkit to communicate with the space server and provides methods to the application developer to easily perform operations on the XVSM space. It makes the communication between client and server transparent and allows the application developer to access the XVSM space as if it was a locally running program. However, due to the asynchronous communication between server and client, the programmer needs to register callback objects when performing an operation on the space which leads to a slightly different programming style. The JavaScript API is programmed in an object oriented way (the Dojo toolkit makes this possible) defining several classes that are organized into different packages, similar to the Java packaging system:

- xvsm.jsapi

  The xvsm.jsapi package contains the main class of the JavaScript API, the 'JSapi'. It provides methods for all operations that can be performed on the XVSM space. Additionally, it contains a 'connect method' taking the Uniform Resource Locator (URL) of the XVSM server as parameter.

- xvsm.jsapi.notification

  In this package the two enumerations NotificationMode and NotificationTarget are located. The mode and target parameter of every notification can only accept values from these enumerations. The NotificationMode combined with a timeout value defines the duration a notification remains valid. The NotificationTarget

defines on which operations the notification shall fire.

- xvsm.jsapi.interfaces

  This package contains the interfaces for operations that return values. Every operation that can be performed on the XVSM has a corresponding interface class. When calling such an operation, an instance of the interface class is passed as parameter and when the XVSM answers the request, the corresponding function of the interface is called (depending on the return value). The interface parameter can be omitted if the application has no interest in the response of the XVSM. Some operations use the same interface as they return the same values (e.g. 'Write' and 'Shift', or 'Read' and 'Take' operations).

- xvsm.jsapi.json

  Operations acquiring complex information as parameters (e.g. the write operation might take many entries as argument) accept objects from the xvsm.jsapi.json class. These classes are used to gather and fill out all the information the operation needs to perform. They all provide a 'json method' transforming the contained data into JSON objects.

- xvsm.jsapi.objects

  The xvsm.jsapi.objects package contains two classes representing an entry and a tuple as well as two enumerations (CoordinationType and ValueTypes). The Tuple class extends the Entry class as it does in the XVSM implementation and both provide a 'json method' transforming data into a JSON object. The CoordinationTypes enumeration contains all possible coordination types of a container. The ValueTypes enumeration contains all possible data types an entry/tuple might consist of.

- xvsm.jsapi.properties

  The ContainerProperty class and two enumerations (ContainerProperties and PropertyMode) make up this package. A ContainerProperty object can be used

to get or set the properties of a container. The different possible properties of containers are listed in the ContainerProperties enumeration. The PropertyMode sets the behaviour of the ContainerProperty object (get, set or reset).

- xvsm.jsapi.selectors

  Several operations on the space (e.g. a read operation) take selectors as arguments. The different selectors are arranged in the xvsm.jsapi.selectors package. A selector is used to broaden or restrict the influence of an operation on a container. It has to match the data access type of the container (e.g. a LindaSelector reduces the number of returned entries of a read operation to the ones matching the template included in the selector). Selectors are integrated into XVSM and each selector class of the JavaScript API represents its counterpart in the XVSM implementation.

- xvsm.jsapi.exceptions

  The xvsm.jsapi.exceptions package contains the exceptions which might occur when calling a method of the API.

- xvsm.jsapi.tests

  Dojo provides JavaScript developers with facilities to test their applications in a similar way the well established JUnit testing framework does for Java applications. The xvsm.jsapi.tests package contains the test cases for the JavaScript API. Test cases are created by calling the 'register method' of the test suite and some code for 'setup', 'runTest' and 'tearDown' can be included. All testcases are listed in the xvsm.jsapi.test.module class and the tests can be executed by pointing an Internet browser to the path 'dojo installation directory/util/doh/runner.html?testModule=xvsm.jsapi.tests.module'.

5 Examples Of Use

This chapter presents two examples of use for the JavaScript API and the XVSMP/-Bayeux system. The first example, the chat application, represents a distributed web application that coordinates itself through the XVSM space. The second example, the XVSM viewer, is a web application monitoring the current state of the XVSM. All the data the XVSM currently holds is displayed and constantly updated within the web application.

5.1 Chat Application

The chat application allows multiple users to join a chat room and communicate with each other by exchanging text messages. It is a simple application to show the use of the XVSMP/Bayeux protocol and the JavaScript API.

5.1.1 Requirements

The Chat application fulfils the following requirements:

- Nickname selection
  Every user can choose a nickname to log into the chat application. The nickname is a unique identifier of a user within the system. If the nickname is already chosen by another user, a warning shall be displayed and the login is denied.

- Channels

The chat application shall support different channels (chat rooms), in which participants can communicate with each other. Every user can create, join or leave a channel. Channels are identified by their channel name which must be unique within the system.

- Notification

  Users shall be constantly notified about changes within the system (e.g. a new channel is created by some other user or a message is written to a channel, that the user previously joined).

5.1.2 Data structure

An important part when building an application that coordinates itself through a space is the data structure used for the coordination. Figure 5.1 shows the data structure of the chat application.

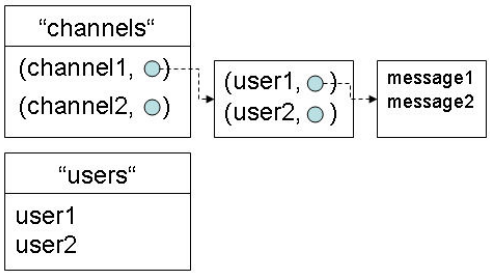

Figure 5.1: Chat application data structure[18, pp. 7]

Every rectangle within figure 5.1 represents a container, tuples within a container are written in parentheses and circles represent container references. There are two named containers 'channels' and 'users', which represent the starting point for every connecting application. Their references can be obtained by using the lookup mechanism of XVSM. To ensures the uniqueness of every username and channel within the system, both containers use a key/value access type. Every attempt to write an entry to the containers with an already existing key/value pair (e.g.: 'user'/'mike' for the 'users' container or 'channel'/'music' for the 'channels' container) will result in a blocking behaviour of the write operation. The operation blocks until a certain amount of time, defined within the timeout parameter of the operation (possible values: 0 - infinite),

elapsed and then throws a TimeoutException. Using a timeout value of zero results in an immediate notification of the failure of the write operation. The chat application then informs the user about the already existing user/channel name. The 'channels' container holds a tuple for every channel that currently exists. It contains the name of the channel and a reference pointing to the channel's private container (the dashed line in figure 5.1). Within the channel's private container every user who is currently member of the channel is represented by another tuple. This tuple contains the username and a reference to the user's container within the channel. If a new message is sent to the channel it is written to all the containers of the users currently member of the channel. The fact that messages are stored separately for every user allows every chat application to retrieve the messages at its own speed, unaffected by the actions of the other participating applications. To notice new messages within the container ever application registers a notification on its container within the channel. Additionally, notifications are also registered on the channel's private container to stay informed about users joining or leaving the channel. If a user leaves a channel, her/his container within the channel is destroyed and the notifications are cancelled. Two more notifications are used by every application to keep the list of existing channels up to date.

5.1.3 Implementation

The chat application essentially consists of 3 parts. The index.html file, which is responsible for the main layout, the chat.js file, where all the application logic is implemented and the channel widget containing the layout and the logic of a chat room. The chat application uses widgets from the Dojo toolkit for its main layout. To support chatting within different chat rooms at the same time a TabContainer widget is used as the main component. The TabContainer widget allows creating multiple windows within one containing frame, where only one window is shown at a time and the others are kept invisible in the background. The user can switch between the different windows by selecting their title from the title bar shown on top of the TabContainer widget. The TabContainer widget shows a main window, which contains the elements for login, channel creation and joining the channel (see Figure 5.2). When joining a channel a new channel widget is created and added to the TabContainer. The

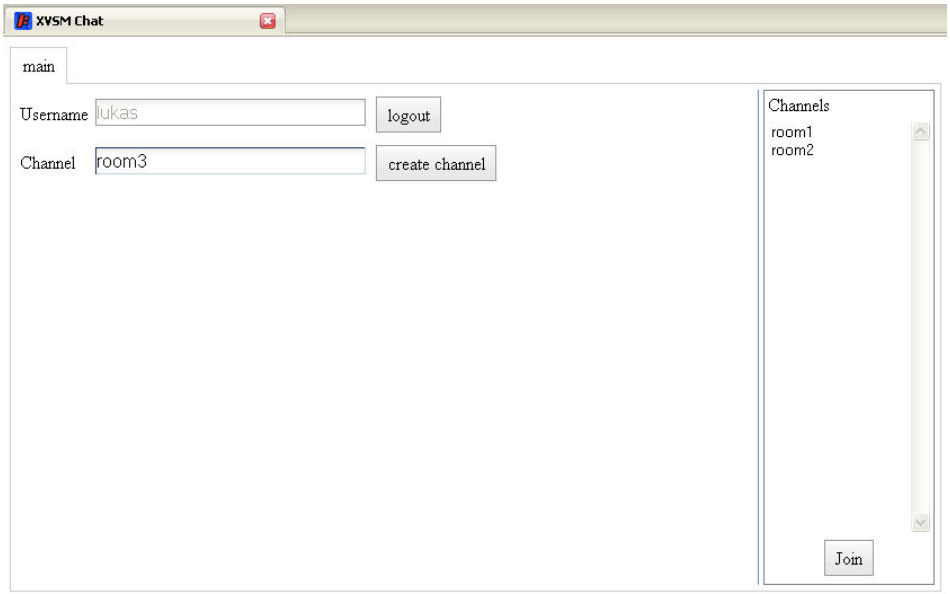

Figure 5.2: Chat application - Main Tab

channel widget consists of two files, the channel.html file describing the user interface of the widget and the channel.js containing methods to display a new message, a new user (who joined the channel) or to send a message to the channel. The user interface shows a list of users currently present in the channel, a field to send new messages and a field where all messages exchanged within this channel are displayed (see Figure 5.3).

5.2 The XVSM Viewer

The XVSM Viewer displays the current state and content of a XVSM space within an Internet browser. It takes advantage of asynchronous communication to constantly inform the Internet browser about changes in the space. It can be used to debug XVSM based applications by monitoring the XVSM space and the changes the application performs on it. The layout is again created with the help of widgets included in the Dojo toolkit. The following section will show and explain the different components of the XVSM Viewer.

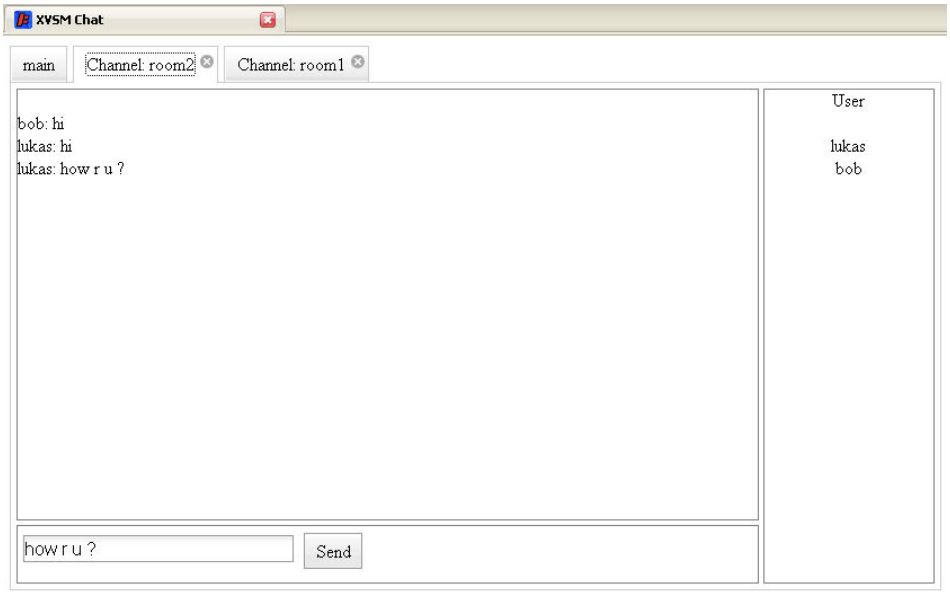

Figure 5.3: Chat application - Channel Tab

5.2.1 Components

The browser window is split up into 3 sections: The title on top of the window, the list of containers on the left side and a window reserved for container details (container window) on the right side (see Figure 5.4).

ContainerList

The list of containers on the left side is divided into 4 sections by the use of the TabContainer widget (see 5.1.3): a list showing the references of all containers, a list for named containers only, a list for unnamed containers and a list of favourites. Containers can be added to or removed from the favourites list to keep track of interesting containers easily. In addition to splitting up the list of containers into four sections, a filtering mechanism is integrated which allows to filter containers based on their reference string. Whenever a container is newly created or a container is removed from the space the XVSM Viewer is informed about it (through notifications) and displays the modification. A new container is highlighted red (see Figure 5.5 whereas a removed container is crossed out for three seconds (see Figure 5.6).

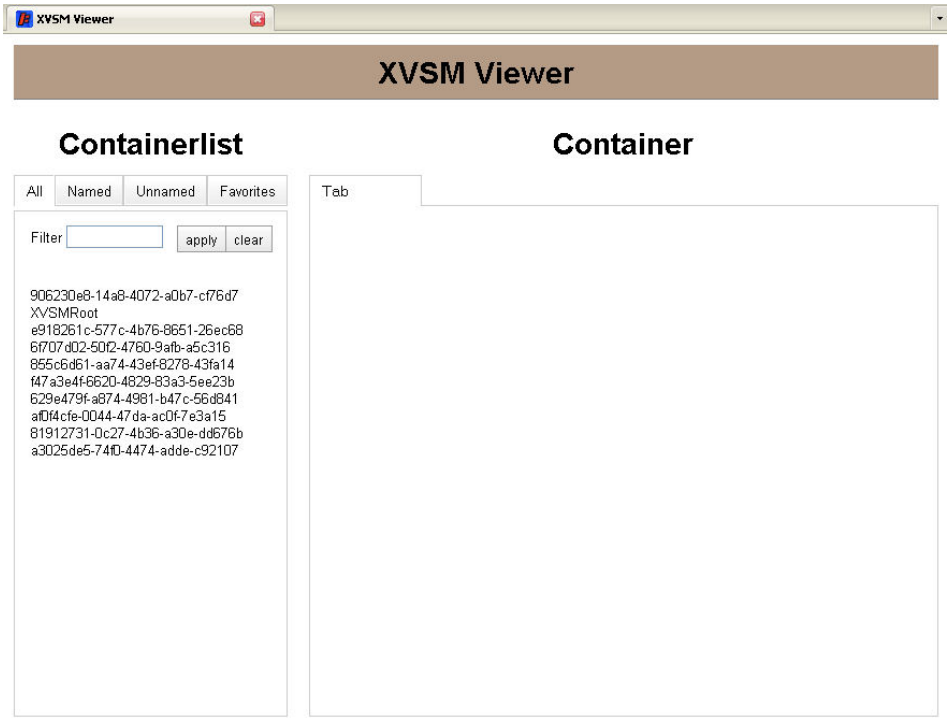

Figure 5.4: XVSM Viewer

By right-clicking a container reference a menu pops up, allowing the user to add the respective container into the current tab, into a new tab, to add the container to the list of favourites or to remove it from this list (Figure 5.7).

Container Window

The right side of the browser window is used to show the details of the different containers. As many distributed applications use a lot of containers to coordinate themselves or to exchange data the container window is also organised into different tabs. Whenever the option 'open in new tab' is chosen from the containerlist menu, a new tab is added to the container window and the container is shown within it. The 'open in current tab' option displays the container in the currently displayed tab of the container window. A tab can show an arbitrary number of containers by extending its viewport with scrollbars. Tabs can also be given names by clicking on their title. Figure 5.8 shows the container window with 2 open tabs, named 'users' and 'channels' representing the data structures from the chat application.

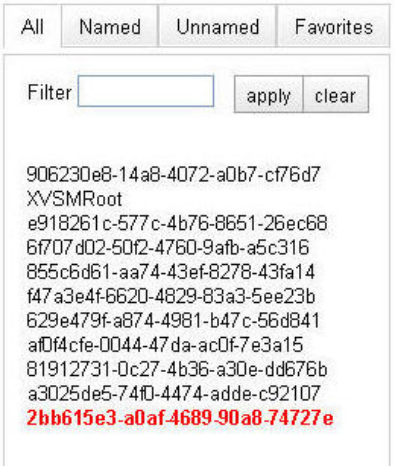

Figure 5.5: ContainerList - New Container

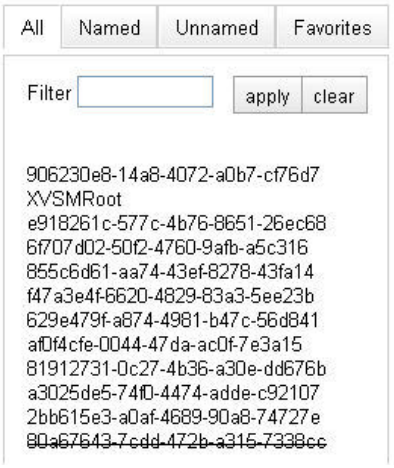

Figure 5.6: ContainerList - Removed Container

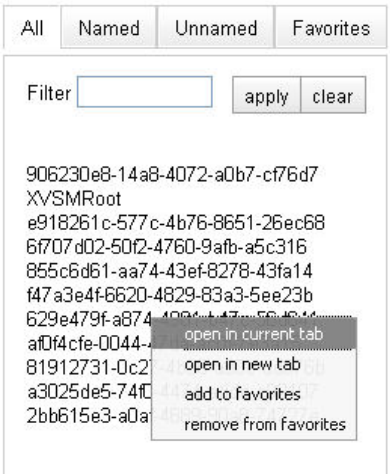

Figure 5.7: ContainerList - Menu

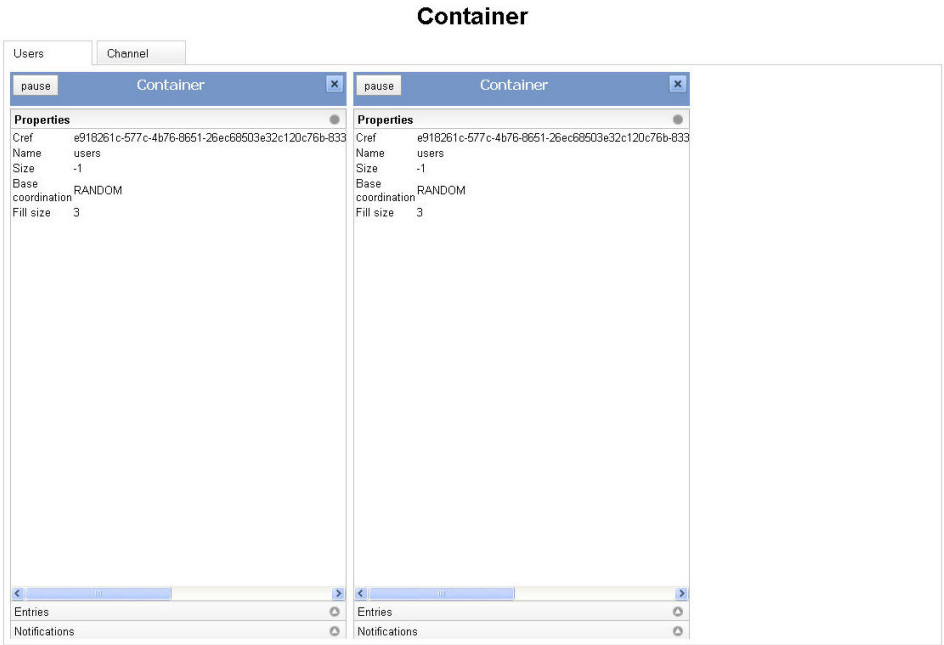

Figure 5.8: Container Window with 2 Tabs

In this picture the 'users' tab shows the same container, in this case the named container 'users' from the chat application, two times. In some cases it does make sense to open up a container two times. On the upper left end of the window showing the container details the 'pause' button allows the user to stop the updating of the content of the container. This way the user can create a snapshot of the current container's content. Opening a container multiple times allows creating snapshots of one container at different points in time. In figure 5.8 the properties of the container are displayed in the picture. By clicking on the 'Entries' or 'Notification' bar, the window showing the properties will close and the window showing the container's entries or notifications will emerge. Figure 5.9 shows the 'Entries' window of the two instances of the 'users' container. One of the examples has been paused and a new user has been added to the container (again highlighted red).

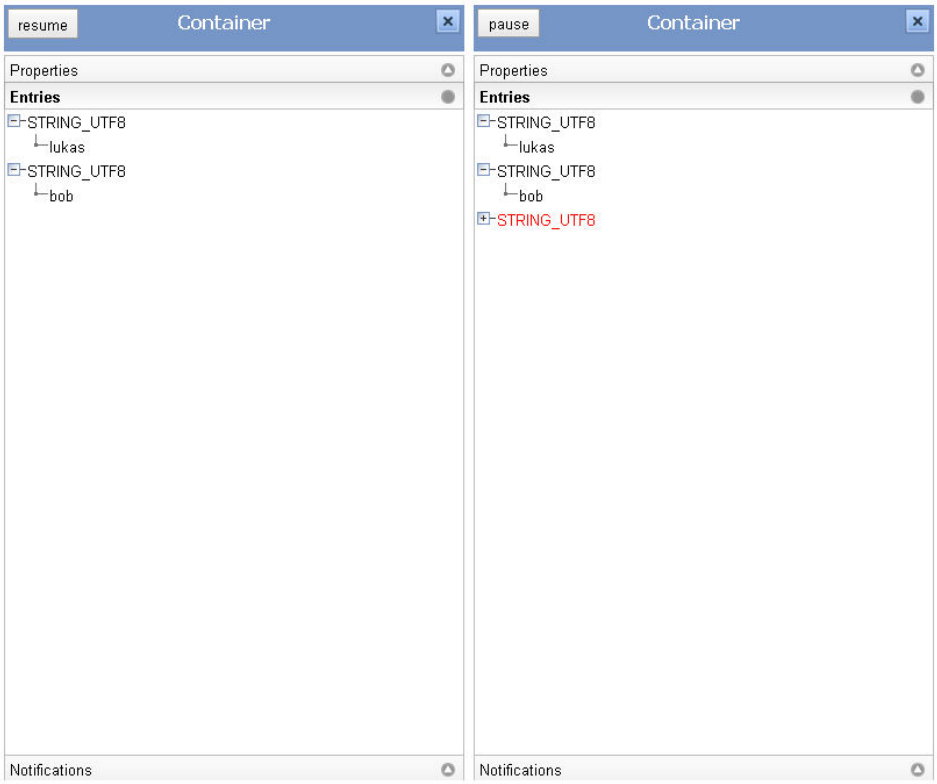

Figure 5.9: Left side: Paused Container
Right side: The same container while a user has been added.

As entries can form complex data structures, they are organized in a tree structure. Entries displaying container references provide the same menu the containerlist offers

for its container references. Notifications on the specific container are displayed when clicking on the last bar within the container details window. See Figure 5.10.

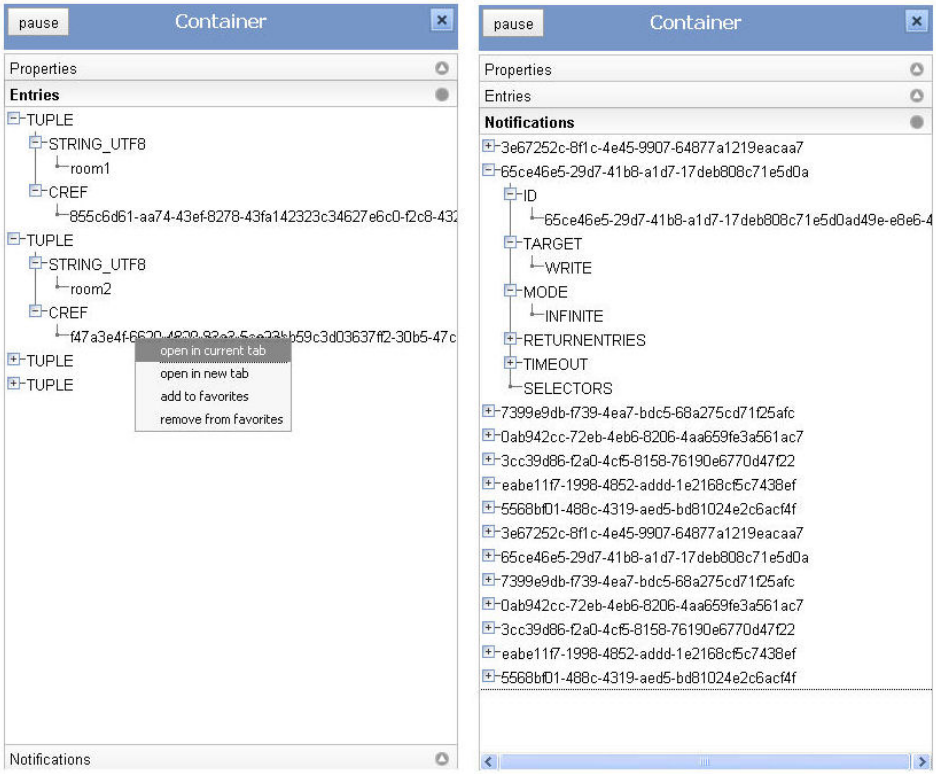

Figure 5.10: Left side: Tree structure of entries and container reference menu.
Right side: Notifications on the container

5.2.2 Implementation

The XVSM Viewer consists of 3 custom widgets, the index.html file, the viewer.js file, the viewer.Container class and two CSS[1] files. The index.html file defines the viewer's outline. Standard Dojo widgets are used to split the browser window into the three areas and the custom widgets are used to display the list of containers and the actual container content. The viewer.js JavaScript file contains the application logic of the XVSM viewer. It establishes the connection to the XVSM space and retrieves the list of containers. The list of containers is stored in an instance of the dojox.collections.SortedList object (an object provided by the Dojo toolkit that sup-

[1] CSS - Cascading Style Sheet

ports the storage of arbitrary sort able data). It uses two notifications (one for creation and one for removal of a container) to keep the list up to date. The viewer.Container class is used every time a user wants to display the details of a container. If the container is opened the first time a new viewer.Container object is created. It subsequently retrieves the container's data and creates several notifications to keep itself up to date about the container's status. It can be seen as the container's equivalent within the XVSM Viewer. The actual displaying of the data is done by the viewer.widget.Container widget. A viewer.widget.Container widget must be registered at the viewer.Container object. The viewer.Container object then calls the widget's methods to display the data. So even if the user opens up one container multiple times, the data is transferred and the notifications on the specific container are created only once.

The three custom widgets are:

- The viewer.widget.FilteringList widget

  The list of container references is implemented through the viewer.widget.FilteringList widget. Every list (named, unnamed, favorites and the common list) is displayed by one instance of the FilteringList widget. It needs an dojox.collections.SortedList object as the data basis. The data object can be set with the setStore method. Additionally, the FilteringList provides methods to set and clear the filter to be applied on the list.

- The viewer.widget.TabContainer widget

  The viewer.widget.TabContainer widget is a very simple extension to Dojo's TabContainer providing the widget with the ability to let the user change the name of the tabs. Basically it only changes the HTML element that displays the tab's title from a 'SPAN' element to an editable 'INPUT' field.

- The viewer.widget.Container widget

  This widget displays the content of a container in the XVSM space. It provides methods to set the properties of the container, the entries of the container and

the notifications of the container. It includes the pause method invoked by the pause button. This method unsubscribes and resubscribes the widget at the viewer.Container object. While unsubscribed the viewer.Container object will not inform the widget about changes in the container. When the updating is resumed (another click on the pause button), the widget is subscribed again and its information about the container is updated.

6 Evaluation

This thesis presents the combination of different already existing and well established technologies and methods to provide reactive, user-friendly and high-performance distributed web applications. They benefit from synergies that emerge by combining the space based computing paradigm for coordination of distributed application and modern concepts of web application including asynchronous communication. This approach offers a lot of advantages, but also includes some restrictions and inconveniences. This chapter gives a general overview of the pros and cons discovered throughout this thesis. Furthermore, it presents a benchmark measuring the performance of the system using Comet transport compared to traditional polling, gives an outlook on developments and discusses the risks this technology might face.

6.1 Advantages

- Coordination in the web

  The main advantage of the technology presented in this thesis is the coordination of distributed web applications using the full potential of the space based computing paradigm. Web applications can make use of any function or service provided by the space without any restrictions compared to traditional desktop applications. The key element to achieve this is surely the asynchronous bidirectional communication between the web application and the space.

- Zero-Install application

There is no longer the need to download and install the application on the workstation to use it. Zero-install applications run within an Internet browser and perform complex operations while providing a neat and reactive user interface comparable to desktop applications. Zero installation applications also relieve machines of the installation overhead of desktop applications and thus reduce the resource consumption.

- General availability

  In addition to the advantage that web applications do not need any installation routine they run within the ubiquitously available Internet browser. As a result they are available on nearly every hardware and software platform, even including modern mobile phones. In addition, the distributed web application can also be accessed from computing devices which only provide the user with very limited access rights (e.g. an Internet cafe, a public Internet terminal, etc.).

- No security restrictions

  By using JavaScript as programming language and abstaining from the use of partly more powerful technologies like Java Applets or Flash applications, the distributed applications presented in this thesis do not suffer from security restrictions, neither on side of the Internet browser, nor on the side of network security (e.g. firewalls). However, JavaScript can be disabled within Internet browsers, rendering the application unusable (see 6.2 Problems).

- Easy application development

  While the combination of these technologies offers a lot of advantages it is also important not to complicate application development. Due to the introduction of the JavaScript-API this thesis provides developers with methods to easily create distributed space-based applications. There is no need to know the protocol or the underlying technology. A developer can easily change from distributed desktop to distributed web application development.

- Communication performance

The use of Comet as transport mechanism reduces the latency between the event generation at the space and the delivery of the event at the client application. However an overhead for the Bayeux protocol as well as the HTTP requests is obviously not avoidable. High-end applications needing the best possible performance might not suit well to be ported to web applications.

- Scalability

  The performance of the presented system can easily be scaled up by adding another space server to the system. XVSM is especially designed to span a space between different machines and thus increases performance of the overall system. If a space server reaches a very high load, an additional server can be added and requests can be redirected to this server. Having different space servers to handle incoming requests will also increase availability as the request dispatcher can easily redirect requests to another server if one crashes.

6.2 Problems

- JavaScript availability on mobile devices

  While most of the mobile devices are already provided with an Internet browser, many of them include a very limited version which often does not support JavaScript. However, with the progress in development of applications for mobile devices and the constant improvement of hardware, it's very likely that soon even small mobile devices will include a JavaScript capable Internet browser. Already Mozilla's MiniMo browser for the Windows Mobile operating system and Opera's Mobile Browser (versions for Windows Mobile, Symbian S60 and UIQ exist) support JavaScript.

- Different programming style

  Despite of the presented JavaScript API the asynchronous communication enforces a slightly different programming style and some additional code lines.

Methods executing operations on the space cannot directly return values (as there is no way to block the operation until the answer returns from the server). Instead, the code needs to be registered as call-back function which is called when the operation returns the value. This creates some coding overhead and sometimes nested constructs.

- Access to resources on the local machine

  Some distributed desktop applications might not be easily ported to web as they require resources on the local machine, which cannot be accessed by a JavaScript application within an Internet browser. Even access to the file system is only possible with the help of other technologies like Java Applets or ActiveX. The use of these technologies has been avoided in this thesis explicitly, because they impose security problems.

- Two connections problem

  Every Internet browser is limiting the number of concurrent HTTP connections to a single server. To perform asynchronous bidirectional communication two HTTP connections are needed per web application, which is the standard limit in the Internet Explorer and Mozilla's Firefox. Whenever a server is hosting more than one XVSMP/Bayeux enabled web application every user can only access one of these applications at a time. A solution to this problem is to raise the limit of simultaneous connections, which is easily done in Mozilla's Firefox (open the URL 'about:config', search and double click the field 'network.http.max-persistent-connections-per-server' and insert a value higher than 2). However, this requires user action and a solution working with a standard browser installation would be preferable. A further possible solution would be trying to reuse existing connections to a server through JavaScript.

- Security

  To increase security all communication can be easily run on a secured Secure Socket Layer (SSL) connection, which should avoid attacks from malicious users. However, the protocol implementation is easily accessible for everybody (as the

JavaScript code is not compiled) and the distributed application (the data on the space) must also be secured against malicious users. So far, there is no mechanism to avoid such a behaviour, but methods to restrict access to containers (read-only or no access at all) might be integrated into the XVSM.

- Performance

  JavaScript is a scripting language interpreted within an Internet browser. Therefore, its performance compared to compiled and optimized computing languages is inferior. The performance was nevertheless satisfying for the two sample applications presented. As already mentioned in the communication performance section, the presented technologies might not be suitable for high-performance applications.

6.3 Benchmark

This thesis focuses on bringing the concepts of reactive programming from the XVSM space to the web applications. The benchmark measures the performance gain when using notifications with Comet as transport mechanism instead of the traditional Ajax polling method. A thread is used to write entries to a container at random speed within a certain interval. The entries contain a timestamp indicating the time of their creation. On the one hand, a web application using the JavaScript API (with Bayeux and Comet as underlying transport) will register a notification on the space. Whenever the notification fires (an entry is written to the container), the application will display the difference between the current time and the time in the entry. On the other hand, a servlet registers a notification at the server and a web application constantly polls the servlet for fired notifications. It also displays the time difference as well as the number of polls where no information was returned. The benchmark is run on a local system to avoid clock synchronization issues. Figure 6.1 shows the results.

The upper part of figure 6.1 shows the changes that occur when the polling interval (the 'Read' column) shrinks. When polling every 2 seconds the latency is about 9 times higher than the latency when using Comet as transport. When polling at maximum

Write (delay Ø)	Read (freq.)	Comet (latency)	Polling (latency)	Polling (empty)
500ms	2s	132ms	1243ms	0.34%
500ms	1s	114ms	675ms	3.78%
500ms	0.5s	102ms	399ms	28.10%
500ms	max	116ms	236ms	545%
200ms	max	134ms	219ms	750%
100ms	max	122ms	207ms	375%
50ms	max	128ms	221ms	320%
min	max	123ms	215ms	223%

Figure 6.1: Benchmark between classic Ajax and Comet transport

speed the latency is reduced to twice the latency of Comet, but the polling overhead increases dramatically (545 percent of the messages are unnecessary!).

The second part of the benchmark shows the changes when the speed of the writing is changed. The polling is performed at maximum speed (which is around 60ms at the testing device). As one might expect, the faster messages are written to the space, the closer polling gets to Comet in performance. However even at the maximum writing speed of the XVSM space (which is not likely to be reached by any real web application) the latency as well the overhead of traditional polling are doubled compared to the Comet transport style. Summarizing it can be stated that the latency with Comet remains constant and at a clearly lower level compared to polling throughout all measurements.

6.4 Related Work

There are several existing approaches to coordinate web applications through shared data spaces, which differ in the degree of supported features (e.g. blocking operations, event based notifications, transactions, etc.) and applicability (e.g. browser security or firewall restrictions). Early approaches used CGI[1] as gateway between the web applications and tuple spaces based on the Linda primitives [24], [7]. In [24] operations on the space are performed one after another following a strict HTTP requests/response pattern. Blocking operations are possible, however the user has to wait for the oper-

[1] CGI - Common Gateway Interface

ation to unblock before a response is send back to her/his browser. Possible timeout problems are not addressed. [7] proposes the use of an intermediate process decoupling the browser-server communication (using CGI) from the execution of the tasks on the space. As a consequence of this solution the web application (using HTML or Java Applets) has to poll the server for the outcome of the previously initiated operations. This approach is in principle slightly similar to the XVSMP/Bayeux approach, however with XVSMP/Bayeux using asynchronous communication the decoupling of communication and execution takes place on both sides, at the web application and at the space server. XVSMP/Bayeux also offers many other advantages like responsive Ajax based user interfaces, support for notifications, different data access types for containers or the support of transactions. Other approaches, which do not specifically focus on clients running in web browsers are [26],[27],[6]. Nevertheless they all provide access to the space using the HTTP protocol as underlying transport. [26] focuses on the storage of XML documents in spaces and uses SOAP over HTTP to provide access to the space. [6] proposes a proprietary protocol on top of HTTP to access the space. Both approaches do not provide support for blocking operations, notifications or transactions. [27] uses XML encoded messages on top of HTTP and proposes to use a dedicated HTTP Request for every operation, thus supporting blocking read operations. This is however not suitable for web clients as the HTTP protocol restricts the number of simultaneous HTTP connections to the same server to two. Furthermore notifications, transactions or different data access types are not supported neither. Approaches providing more enhanced features for web applications mostly rely on the use of Java Applets and Remote Method Invocation (RMI). Examples are [12] or [1], which provide blocking reads, notifications and transactions. However the use of Java Applets has specifically been avoided in this work due to security restrictions within the Internet browsers. Furthermore these approaches might not be usable in firewall protected networks, as they require additional open ports.

6.5 Outlook

Caching

An important restriction of web applications compared to desktop applications is the fact that they remain usable only as long as the connection between the application and the server exists. However, with mobile devices and people constantly moving from one place to another a permanent connection between web application and server is very hard to maintain. Desktop applications are less affected by a loss of connection. The user can often continue to work with only little restrictions and once the connection is re-established the desktop application synchronizes the changes the user made while being off-line with the server. Due to the clear distinction between the application logic and the communication with the server, the presented approach in this thesis could be easily extended to provide such a service even with web applications. The page design and the application logic are loaded at the start-up of the application. Upon a connection failure, operations performed on the space need to be internally cached at the web client and executed as soon as the connection is re-established. Even saving those web applications to a file, being able to shut-down the computer, restart it and reopen the application while being off-line might become possible. The often proposed example of a mobile worker performing operations in the field while being off-line and later synchronizing with the enterprise server could thus be made feasible with a web application.

Efficiency

The caching facilities could not only be used for masking connection disruptions but also to optimize communication behaviour (e.g. waiting for a bigger amount of data before transferring all the data at once or transfer of data depending on the server load, etc.). Generally speaking the approach of this thesis can be extended to create very efficient and highly productive web applications. High efficiency will be especially useful for web applications aimed at mobile devices, which is a big future market.

Restrictions

When looking at the current working solution, it still suffers from a few inconveniences and restrictions which could be solved in future work. The 'two connections problem' restrains the user to only use one web application running on the same space server at the same time. As mentioned in section 6.2 using a single instance of the JavaScript API for all web applications using the same space server could potentially solve this problem, though its feasibility needs to be evaluated first (security restrictions might apply).

JavaScript Spaces

With the introduction of EcmaScript 4.0 (JavaScript 2.0) in 2008 programming in the large with JavaScript shall get easier, allowing to create more complex web applications. Additionally, native support for JSON will be integrated. As a result the packaging and object oriented programming features currently provided by the Dojo toolkit will be natively supported by JavaScript. With JavaScript evolving towards a fully object oriented programming language, the possibility of creating a JavaScript XVSM space implementation, running within a web browser, becomes feasible. Web applications running their own spaces that are communicating directly between each other (e.g. to replicate the space contents) would represent a new generation of web applications, where the server no longer plays any part after the application has been delivered to the web browser. However, many security restrictions (e.g. cross site XMLHTTPRequests or opening a port for incoming requests) apply with current web browsers, which inhibit a fully featured space running within a web browser. Nevertheless with newer web browser versions current restrictions might be removed (e.g. Mozilla's Firefox version 3.0 already allows cross site XMLHTTPRequests). In the mean time it is possible to run spaces within Internet browsers, that communicate with each other using the Bayeux protocol and a common Bayeux server as a message proxy. Using multiple Bayeux servers to build up a sort of Bayeux network cluster would make such a system resistant to failures of single nodes.

XVSM Evolution

A lot of research is currently going on in the field of space based computing and

the XVSM specification as well as its reference implementation MozartSpaces are still subject to constant improvements and evolution. During the elaboration of this theses MozartSpaces evolved from version 0.8 to version 1.0 . This theses is build on version 0.9.0.1 and does not take into account the latest development steps of MozartSpaces. The following section describes the necessary steps to adapt the current XVSMP/Bayeux implementation to changes of the XVSM implementation.

The XVSMP/Bayeux implementation uses the Core API to communicate with XVSM. It is not affected by changes of the implementation details of XVSM itself as long as the Core API remains unchanged. If the Core API changes the XVSMP/Bayeux implementation must be adapted. The following changes are necessary if a parameter is added to a function within the Core API:

- The parameter must be integrated into the XVSMP/Bayeux protocol specification.

- XVSM-Server implementation
  The method handling class within the org.xvsm.server.json package must be adapted to extract the parameter from the retrieved JSON object and include the parameter when calling the Core API method.

- JavaScript API
  The object, containing the method's parameters (in the xvsm.jsapi.json.* package), must be extended by the new parameter and the corresponding get and set methods. Additionally the object's json method must be changed to include the parameter into the created JSON object.

Removing a parameter from the Core API requires similar steps within the XVSMP/Bayeux implementation, however instead of adding the parameter processing logic, it has to be removed. If an entire method is added to the Core API, a new class must be added to the org.xvsm.json.package of ther XVSMP/Bayeux server implementing the processing of the corresponding JSON object. Within the JavaScript API the new method must be added to the main class. The parameters of the method are encoded in a new class within the xvsm.jsapi.json package. The xvsm.jsapi.interfaces package must be extended by a new class for the method as well.

The most recent Mozartspace implementation does no longer use container references to address containers but instead uses container URLs to uniquely identify containers across different spaces. As a result an operation on a container can be performed through any Core API, which will forward the operation to the space holding the container or directly execute the method if the container is locally available. To adopt this principle for XVSMP/Bayeux the cref parameters must be replaced by the URL parameter. If the target space is running an XVSMP/Bayeux server and the web browser allows cross site XMLHTTPRequests the JavaScript API might directly contact the addressed space.

Portability

The XVSMP/Bayeux implementation is designed to run within the web server Jetty. Jetty is entirely implemented in Java and thus it can be deployed to any platform supported by the Java Virtual Machine (including Windows, Linux, Unix, etc.). XVSMP/Bayeux also respects the Java Servlet API and can be run in any other servlet container as well. However, without the support of Continuations (which is so far only included in the Jetty) the performance of the web application will degrade if there is no other mechanism included in the servlet container to support web applications using Comet communication.

One of the most popular web servers, Microsoft's Internet Information Services (IIS), does not natively include a servlet container nor support for the Java programming language. However different servlet container plugins for the IIS exist, allowing the use of XVSMP/Bayeux with IIS. The support of XVSMP/Bayeux within the native IIS programming environment comprises several problems. The implementation of the server part of XVSMP/Bayeux must be rewritten in an Active Server Pages (ASP) .NET programming language. The Bayeux Server implementation must be ported to ASP .NET and most importantly the XVSM implementation, which is written in Java, still requires a servlet container (e.g. Tomcat) to run. The XVSMP/Bayeux ASP.NET server implementation must subsequently forward the Core API method calls to the XVSM running within the servlet container.

7 Conclusion

This thesis presents the combination of different technologies to achieve high-performance, efficiency and user-friendliness for distributed web applications, which are coordinated through a common virtual space. The virtual space is created using the new middleware application XVSM. It offers means for efficient coordination of distributed applications, reduces the development complexity and provides services like naming or transactions. To overcome communication restrictions of traditional web applications (which are using unidirectional synchronous client/server communication), the Bayeux protocol is used as underlying transport. "*The protocol is designed to overcome the client/server nature of the internet in general and specifically of HTTP to allow asynchronous messaging between all participants.*"[22]. Bayeux enables server initiated information push towards the client (web application) porting the concept of reactive programming to the web. Bayeux is an emerging protocol, nevertheless existing implementations for servers and web applications already exist.

In order to combine the space as a coordination platform and Bayeux as underlying transport, a new protocol called XVSMP/Bayeux was developed in this theses, defining the different operations an application can perform on the space. The XVSM is extended by a server component implementing the XVSMP/Bayeux protocol and using an existing Bayeux server implementation, to provide clients with access. On the client side a Bayeux implementation provided by the Dojo toolkit is used. On top of the Bayeux client a JavaScript API implementing the XVSMP/Bayeux protocol was created. The JavaScript API provides web application developers with a simple interface to interact with the space. Using the JavaScript API, the developer is relieved from knowing about the XVSMP/Bayeux protocol, the underlying Bayeux transport or the server component embedding the XVSM.

As a proof of concept two examples of use have been developed, a chat and a monitoring application. The chat application allows users to group together and exchange messages in different chat rooms. It is representing a distributed multi-user application, which is greatly benefiting from the reactive programming possibilities. User generated events (e.g. a chat message) can be directly pushed from the space to the recipients (e.g. users in the same chat room). Any polling overhead is avoided and the latency between event creation and event delivery is minimized. The monitoring application allows the user to display the state and content of the XVSM space. It is also possible to create snapshots of the spaces content at different times, which shall help in the development and debugging of space based applications. Finally a benchmark is presented, showing that asynchronous communication results in a very low and constant latency when delivering events to the client compared to traditional polling.

Appendices

A Code Organization

The source code of the XVSMP/Bayeux system is split into five directories:

- XVSMP-Bayeux Server

  This directory contains the source code of the XVSMP/Bayeux server implementation. It uses the software project management tool Maven[1] to declare its dependencies, compilation and build options. Following the Maven directory structure the source code is found in "src/main/java" including the sources of the JSON[2] project (org.json.* package) , the source of the Bayeux server (org.mortbay.* and dojox.cometd.* packages) and the XVSMP/Bayeux implementation (org.xvsm.* package). The "src/main/resources" directory contains the logging settings and the "src/main/webapp" directory the index.html file, which links to the sample applications. Copying the content of the four other XVSMP-Bayeux directories into the "src/main/webapp" directory and running Maven to build the XVSMP/Bayeux server results in a single .war file containing the whole XVSMP/Bayeux system. The "doc" directory contains the JavaDoc-generated API of the XVSMP/Bayeux server.

- XVSMP-Bayeux JavaScript API

  The XVSMP-Bayeux JavaScript API directory contains the Dojo toolkit (the dijit, dojo, dojox and util directory) and the JavaScript API (the xvsm directory). The content of the API's subdirectories is explained in more detail in section 4.3.

[1] http://maven.apache.org/
[2] http://www.json.org

- XVSMP-Bayeux Chat Application

  The content of the Chat Application's directory is explained in detail in section 5.1.3. In order to work properly it is important that the paths to the the JavaScript API within the index.html file and the URL to the XVSMP-Bayeux server within the chat.js file are set correctly (connect method).

- XVSMP-Bayeux Viewer Application

  The content of the Viewer Application's directory is explained in detail in section 5.2.2. In order to work properly it is important that the paths to the the JavaScript API within the index.html file and the URL to the XVSMP-Bayeux server within the viewer.js file are set correctly (connect method).

- XVSMP-Bayeux Performance Test

  This directory contains the four files used to perform the benchmark presented in section 6.3. In order to start the benchmark the space must be empty and the first web browser must open the comet.html file. When the second web browser opens the poll.html file, the first poll request initiates the benchmark (a different web browser or a web browser with a higher "max-persistent-connections-per-server" value is recommended due to the 2-connections problem, see section 6.2).

B Javascript API

connect(url)

description	Connects the JavaScript API to the XVSM space.
parameter	**url**: String The url of the XVSM System.

createNamedContainer(op, cb)

description	This method creates a named container on the XVSM system.
parameter	**op**: xvsm.jsapi.json.CreateNamedContainer Contains all the information needed to create a named container. **cb**: xvsm.jsapi.interfaces.CreateNamedContainerInterface This parameter must implement the interface CreateNamedContainer-Interface. It's methods are called upon reception of an answer from the XVSM system to the request.

clearSpace(cb)

description	Clears the content of the space.
parameter	**cb**: xvsm.jsapi.interfaces.ClearSpaceInterface This parameter must implement the interface ClearSpaceInterface. It's methods are called after the operation was performed.

destroyNamedContainer(name, cb)

description	This function removes a container identified by it's name on the XVSM space.
parameter	**name**: String The name of the container to be removed. **cb**: xvsm.jsapi.interfaces.DestroyNamedContainerInterface This parameter must implement the interface DestroyNamedContainer-Interface. It's methods are called upon reception of an answer from the XVSM system to the request. If this parameter is left out (or set null) answers to this request are ignored.

getNamedContainer(name, cb)

description	Searches the XVSM space for a container with a specific name.
parameter	**name**: String The name of the container to be searched for. **cb**: xvsm.jsapi.interfaces.getNamedContainerInterface This parameter must implement the interface getNamedContainerInter-face. It's methods are called upon reception of an answer from the XVSM system to the request.

createContainer(op, cb)

description	This method creates a container on the XVSM system.
parameter	**op**: xvsm.jsapi.json.CreateContainer Contains all the information needed to create a container. **cb**: xvsm.jsapi.interfaces.CreateContainerInterface This parameter must implement the interface createContainerInterface. It's methods are called upon reception of an answer from the XVSM system to the request.

destroyContainer(cref, cb)

description	Removes a container at the XVSM space.
parameter	**cref**: String The reference of the container to be destroyed. **cb**: xvsm.jsapi.interfaces.DestroyContainerInterface This parameter must implement the interface DestroyContainerInterface. It's methods are called upon reception of an answer from the XVSM system to the request. If this parameter is set to null, answers to this request are ignored.

getContainerProperties(op, cb)

description	This function receives the properties of a container at the XVSM system.
parameter	**op**: xvsm.jsapi.json.GetContainerProperties Contains all the properties to be fetched. **cb**: xvsm.jsapi.interfaces.GetContainerPropertiesInterface This parameter must implement the interface GetContainerPropertiesInterface.It's methods are called upon reception of an answer from the XVSM system to the request.

setContainerProperties(op, cb)

description	This function sets the properties of a container on the XVSM system.
parameter	**op**: xvsm.jsapi.json.SetContainerProperties Contains all the properties to be set. **cb**: xvsm.jsapi.interfaces.SetContainerPropertiesInterface This parameter must implement the interface SetContainerPropertiesInterface. It's methods are called upon reception of an answer from the XVSM system to the request. If this parameter is set to null, answers to this request are ignored.

read(op, cb)

description	This method reads entries from a container on the XVSM system.
parameter	**op**: xvsm.jsapi.json.ReadTakeDestroy Contains all the information necessarry to read entries from a container. **cb**: xvsm.jsapi.interfaces.ReadTakeDestroyInterface This parameter must implement the interface ReadTakeDestroyInterface. It's methods are called upon reception of an answer from the XVSM system to the request.

take(op, cb)

description	This method takes entries from a container on the XVSM system (consuming read).
parameter	**op**: xvsm.jsapi.json.ReadTakeDestroy Contains all the information necessary to take entries from a container. **cb**: xvsm.jsapi.interfaces.ReadTakeDestroyInterface This parameter must implement the interface ReadTakeDestroyInterface. It's methods are called upon reception of an answer from the XVSM system to the request.

destroy(op, cb)

description	This method removes entries from a container on the XVSM system.
parameter	**op**: xvsm.jsapi.json.ReadTakeDestroy Contains all the information necessary to remove entries from a container. **cb**: xvsm.jsapi.interfaces.ReadTakeDestroyInterface This parameter must implement the interface ReadTakeDestroyInterface. It's methods are called upon reception of an answer from the XVSM system to the request. If this parameter is set to null answers to this request are ignored.

write(op, cb)

description	This method writes entries to a container on the XVSM system.
parameter	**op**: xvsm.jsapi.json.WriteShift Contains all the information necessary to take entries from a container. **cb**: xvsm.jsapi.interfaces.WriteShiftInterface This parameter must implement the interface xvsm.jsapi.interfaces.WriteShiftInterface. It's methods are called upon reception of an answer from the XVSM system to the request. If this parameter is set to null answers to this request are ignored.

shift(op, cb)

description	This method shifts entries to a container on the XVSM system (overwriting write).
parameter	**op**: xvsm.jsapi.json.WriteShift Contains all the information necessary to shift entries to a container. **cb**: xvsm.jsapi.interfaces.WriteShiftInterface This parameter must implement the interface xvsm.jsapi.interfaces.WriteShiftInterface. It's methods are called upon reception of an answer from the XVSM system to the request. If this parameter is set to null answers to this request are ignored.

createNotification(op, cb)

description	This function creates a Notification on the XVSM System.
parameter	**op**: xvsm.jsapi.json.CreateNotification A CreateNotification object including all the details of the notification to be created. **cb**: xvsm.jsapi.interfaces.NotificationInterface This parameter must implement the interface xvsm.jsapi.interfaces.NotificationInterface. It's methods are called upon reception of an answer from the XVSM system to the request.

cancelNotification(id, cb)

description	This function cancels an existing notification on the XVSM system.
parameter	**id**: String The Id of the notification to cancel. **cb**: xvsm.jsapi.interfaces.CancelNotificationInterface This parameter must implement the interface xvsm.jsapi.interfaces.CancelNotificationInterface. It's methods are called upon reception of an answer from the XVSM system to the request. If this parameter is set to null answers to this request are ignored.

createTransaction(timeout, cb)

description	This function creates a transaction.
parameter	**timeout**: Number Specifies the amount of time the system tries to create a transaction before giving up and issuing a TimeoutException. **cb**: xvsm.jsapi.interfaces.CreateTransactionInterface This object methods are called upon reception of an answer from the XVSM system to the request.

commitTransaction(id, cb)

description	This function commits an existing transaction on the XVSM system.
parameter	**id**: String The Id of the transaction to commit. **cb**: xvsm.jsapi.interfaces.TransactionInterface It's methods are called upon reception of an answer from the XVSM system to the request. If this parameter is set to null, answers to this request are ignored.

rollbackTransaction(id, cb)

description	This function rollbacks an existing transaction on the XVSM system.
parameter	**id**: String The id of the transaction to rollback. **cb**: xvsm.jsapi.interfaces.TransactionInterface It's methods are called upon reception of an answer from the XVSM system to the request. If this parameter is set to null, answers to this request are ignored.

listContainers(cb)

description	This function retrieves the references of all containers in the space.
parameter	**cb**: xvsm.jsapi.interfaces.ListContainersInterface This parameter must implement the interface ListContainersInterface. It's methods are called upon reception of an answer from the XVSM system to the request.

listNotifications(cref, cb)

description	This function retrieves the references of all notifications on a specific container in the space.
parameter	**cref**: String The reference of the container. **cb**: xvsm.jsapi.interfaces.ListNotificationsInterface This parameter must implement the interface ListNotificationsInterface. It's methods are called upon reception of an answer from the XVSM system to the request.

C Source Code Sample - Chat Application

Listing C.1: index.html

```
<!--
 -  Filename:    index.html
 -  Author:    Lukas Lechner
 -  Created:     5.3.2007
 -  Description:  The main file of the XVSM Chat application.
 -->

<html>
  <head>
    <title>XVSM Chat</title>
      <style type="text/css">
          @import "../jsapi/dijit/themes/tundra/tundra.css";
      </style>
    <script type="text/javascript">
      /* If isDebug is set to true, a debug window is added. When
       * setting the id of an element (e.g. a Tab) to the
       * debugContainerId value, this element is used as debug window.
       */
      var djConfig = {  isDebug: true,
                debugContainerId : "debug",
                parseOnLoad: true
              };
    </script>
    <!--
       The src attribute must point to the dojo installation (relative
       or absolute url) including the  JavaScript API.
    -->
    <script type="text/javascript" src="../jsapi/dojo/dojo.js"></script>
    <script type="text/javascript" src="./chat/chat.js"></script>
  </head>
  <body class="tundra">
    <div id="TabContainer" dojoType="dijit.layout.TabContainer"
      style="width:100%; height: 100%;" selectedChild="maintab">
      <div dojoType="dijit.layout.LayoutContainer" id="maintab"
        title="main">
        <div dojoType="dijit.layout.ContentPane" layoutAlign="client"
          style="margin: 5px; border-right: 1px solid #6290d2">
          <table>
            <tr><td>Username</td>
              <td><div id="userName" dojoType="dijit.form.TextBox"></div></td>
              <td><div dojoType="dijit.form.Button" id="loginButton"
                onclick="loginlogout();">login</div></td>
            </tr>
            <tr><td>Channel</td>
              <td><div id="channelName" dojoType="dijit.form.TextBox"></div></td>
              <td><div dojoType="dijit.form.Button" id="createChannelButton"
                onclick="createChannel();" disabled>create channel</div></td>
```

```
            </tr>
          </table>
        </div>
        <div dojoType="dijit.layout.ContentPane" layoutAlign="right"
          style="border: 1px solid grey; margin-right: 5px; margin-top: 5px;
            margin-bottom: 5px;">
          <table border="0" height="100%" width="150">
          <tr><td style="background-image:url(chat/images/button.gif);">
            Channels
          </td></tr>
          <tr height="100%" valign="top"> <td>
            <select id="Channels" style="height:100%; width: 100%;
              border: 0px solid #6290d2;" multiple> </select>
          </td></tr>
          <tr height="20"><td align="center">
            <div dojoType="dijit.form.Button" id="joinChannelButton"
              onclick="joinChannel();" disabled>Join</div>
          </td></tr>
          </table>
        </div>
      </div>
    </div>
  </body>
</html>
```

Listing C.2: chat/chat.js

```
/**
 *  Filename:     chat.js
 *  Author:     Lukas Lechner
 *  Created:    5.3.2007
 *  Description:  This file contains the logic of the chat-client.
 */

  /* The custom widgets of the chat-application are registered within the
   * dojo packaging system, so that they can be accessed the same way
   * than the widgets provided with the dojo toolkit.
   */
  var loc = document.location.href;
  loc = loc.replace("index.html","");
  dojo.registerModulePath("chat", loc + "chat");

  dojo.require("dojox.layout.FloatingPane");
  dojo.require("dojox.collections.ArrayList");
  dojo.require("dijit.layout.ContentPane");
  dojo.require("dijit.layout.LayoutContainer");
  dojo.require("dijit.layout.ContentPane");
  dojo.require("dijit.layout.TabContainer");
  dojo.require("dijit.form.Button" );
  dojo.require("dijit.form.TextBox");
  dojo.require("xvsm.jsapi.JSapi");
  dojo.require("chat.widget.Channel");

  dojo.require("dojo.parser");

  /* This method is executed when the internet browser finished loading the
   * chat application. It initializes and connects the JavaScript API.
   */
  dojo.addOnLoad( function() {
    jsapi = new xvsm.jsapi.JSapi();
    jsapi.connect("../../cometd");
  });

  /* If the web application is closed the shutdown method will be executed.
   */
  dojo.addOnUnload( function() {
     shutdown();
  });
```

```
// The names of the two named channels for usernames and channelnames.
var channels = "channels";
var users = "users";

var channelsCref = null;
var channelNotificationId = null;
var usersCref = null;
var openChannels = new dojox.collections.ArrayList();

/*  Searches for a named container. If it doesn't exist it will be
 *  created. In both cases the doProcess method of the cb parameter will
 *  be called with the cref of the container.
 */
function getOrCreateContainer(name, cb) {
  var x = new xvsm.jsapi.interfaces.GetNamedContainerInterface();
  x.doProcess = function(cref){
    cb.doProcess(cref);
  };

  x.unknownContainerNameException = function(){
    var y = new xvsm.jsapi.interfaces.CreateNamedContainerInterface();
    y.doProcess = function(cref) {
      cb.doProcess(cref);
    }
    jsapi.createNamedContainer(new CreateNamedContainer(name, null,
      "RANDOM"), y);
  }
  jsapi.getNamedContainer(name, x);
}

/*  Used to retrieve all currently available channels.
 */
function getChannels() {
  if (channelsCref === null) {
    getOrCreateContainer(channels, new setChannelsCref());
  } else {
    var read = new ReadTakeDestroy(channelsCref, 0, null,
      [new RandomSelector(-1)]);
    jsapi.read(read, new setChannelsResponse());
  }
}

/* Setup for the channelContainer. The container properties are set to
 * the coordination type "key". If the channel already exists this is an
 * overhead. However it doesn't harm the application, because this method
 * is only executed once at startup. A notification is created to listen
 * for creation or deletion of channels.
 */
function setChannelsCref() {
  this.doProcess = function(cref) {
    channelsCref = cref;

    var cp = new SetContainerProperties();
    cp.setContainerRef(cref);
    cp.addProperty(new ContainerProperty(
      ContainerProperties.COORDINATION_TYPES, "KEY", "SET"));
    jsapi.setContainerProperties(cp);

    var c = new CreateNotification(channelsCref, -1,
      NotificationTarget.WRITE ,
    NotificationMode.INFINITE, true);
    jsapi.createNotification(c, getChannelListener());

    setChannels();
  }
}

/* Returns the handler for the notification listening for changes in
```

```
 * the channel container. When the notification fires, the setChannels
 * method is called to retrieve anew the list of channels.
 */
function getChannelListener() {
  var x = new xvsm.jsapi.interfaces.NotificationInterface();
  x.setNotificationId =  function(id) {
    channelNotificationId = id;
  };
  x.handleNotification = function(){
    setChannels();
  };
  return x;
};

/*  Retrieves the list of currently available channels.
 */
function setChannels() {
  var read = new ReadTakeDestroy(channelsCref, 0, null,
    [new RandomSelector(-1)]);
  jsapi.read(read, new setChannelsResponse());
}

/*  Displays the names of the channels within the chat-application.
 */
function setChannelsResponse() {
  this.doProcess = function(list) {
    var channelwindow = dojo.byId("Channels");
    while (channelwindow.length > 0) {
      channelwindow.remove(0);
    }
    for (var i = 0; i < list.length; i++) {
      var t = document.createElement("option");
      t.text = list[i].value[0].value;
      t.value = list[i].value[1].value;
      // This is a browser issue. Some browsers only accept one
      // of the following versions of the add method.
      try {
        channelwindow.add(t, null);
      } catch (ex) {
        channelwindow.add(t);
      }
    }
    if (list.length > 0) {
      dijit.byId("joinChannelButton").setDisabled(false);
    }
  }
  dijit.byId("createChannelButton").setDisabled(false);
}

/*  Creates a new channel.
 */
function createChannel(name) {
  var box = dijit.byId("channelName");
  var name = box.textbox.value;
  if (name !== null) {
    jsapi.createContainer(new CreateContainer(null,
      CoordinationType.RANDOM), getChannelCreated(name));
  };

  /* Every channel needs its own container. The container is set to
   * the coordination type "key". Its reference and its name are stored
   * to the named container "channel".
   */
  function getChannelCreated(name) {
    var x = new xvsm.jsapi.interfaces.CreateContainerInterface();
    x.name = name;

    x.doProcess = function(cref) {
      var cp = new SetContainerProperties();
```

```
      cp.setContainerRef(cref);
      cp.addProperty(new ContainerProperty(
        ContainerProperties.COORDINATION_TYPES, "KEY", "SET"));
      jsapi.setContainerProperties(cp);

      var e = new Entry();
      e.setString(name);
      e.setContainer(channelsCref);

      var e2 = new Entry();
      e2.setCref(cref);
      e2.setContainer(channelsCref);

      var t = new Tuple();
      t.addEntry(e);
      t.addEntry(e2);
      t.addSelector(new KeySelector("username", ValueTypes.STRING,
        this.name));
      t.setContainer(channelsCref);

      var w = new WriteShift();
      w.addEntry(t);
      w.setTimeout(0);
      jsapi.write(w);
    }

    x.fatalException = function() {
      alert("Could not create channel");
    }
    return x;
  };
}

/* Joins the channel currently selected from the list of channels.
 * If the channel doesn't exist or the user already joined the selected
 * channel, the operation is aborted. Otherwise a new channel widget is
 * created and notifications are created to listen for users joining,
 * leaving or posting to the channel.
 */
function joinChannel() {
  var ch = dojo.byId("Channels");
  var channel  = ch.options[ch.selectedIndex].text;
  var cref = ch.options[ch.selectedIndex].value;
  if (channel !== null) {
    var tab = dijit.byId("TabContainer");
    if (openChannels.contains(cref)) return;
    var ele = document.createElement("div");
    var props= {
            id: channel,
            channelCref: cref,
            title: "Channel: " + channel,
            closable: "true",
            onClose: tabClosed
          };

    var newtab = new chat.widget.Channel(props, ele);

    newtab.channelName = channel;
    newtab.channelCref = cref;

    tab.addChild(newtab);
    tab.selectChild(newtab);

    var cN = new CreateNotification(cref, -1,
      NotificationTarget.WRITE, NotificationMode.INFINITE, true);
    jsapi.createNotification(cN, getUserAddListener(newtab));

    var cN = new CreateNotification(cref, -1,
      NotificationTarget.DESTROY, NotificationMode.INFINITE, true);
```

```
    jsapi.createNotification(cN, getUserRemoveListener(newtab));

    jsapi.createContainer(new CreateContainer(),
      getJoinChannel2(newtab));

    openChannels.add(cref);
  }

  /* Every user has a container within a channel. This method writes
   * the username and the reference of the user's container for this
   * channel into the container of the channel. A Keyselector is used
   * to verify that every user can only join a channel once.
   */
  function getJoinChannel2(tab) {
    var x = new xvsm.jsapi.interfaces.CreateContainerInterface();
    x.tab = tab;
    x.doProcess = function(cref) {
      this.tab.userCref = cref;

      var e = new Entry();
      e.setString(user);
      e.setContainer(this.tab.channelCref);

      var e2 = new Entry();
      e2.setCref(cref);
      e2.setContainer(this.tab.channelCref);

      var t = new Tuple();
      t.addEntry(e);
      t.addEntry(e2);
      t.addSelector(new KeySelector(user, ValueTypes.STRING, user));
      t.setContainer(this.tab.channelCref);

      var w = new WriteShift(0);
      w.addEntry(t);
      jsapi.write(w);

      var cN = new CreateNotification(cref, -1,
        NotificationTarget.WRITE, NotificationMode.INFINITE, false);
      jsapi.createNotification(cN, getMessageListener(cref, this.tab));
    }
    return x;
  }
}

/* This method returns the handler for the notification (listening for users
 * joining the channel) on the userlist of a channel. If it fires the list
 * of users in the channel will be retransmitted.
 */
function getUserAddListener(tab) {
  var x = new xvsm.jsapi.interfaces.NotificationInterface();
  x.tab = tab;
  x.setNotificationId = function(id) {
    this.tab.userAddNotificationId = id;
  };
  x.handleNotification = function() {
    getUserList(this.tab);
  };

  x.notifyEntries = function(list) {
    getUserList(this.tab);
  };
  return x;
}

/* This method returns the handler for the notification (listening for
 * users leaving the channel) on the userlist of a channel. If it fires
 * the list of users in the channel will be retransmitted.
 */
```

```
function getUserRemoveListener(tab) {
  var x = new xvsm.jsapi.interfaces.NotificationInterface();
  x.tab = tab;

  x.setNotificationId = function(id) {
    this.tab.userRemoveNotificationId = id;
  };

  x.handleNotification = function() {
    try {
      getUserList(this.tab);
    } catch (e) {
      console.debug(e);
    }
  };
  return x;
}

/* This method returns the handler for the notification listening for
 * new messages on the channel. If it fires the new available message
 * will be read from the XVSM.
 */
function getMessageListener(cref, tab) {
  var x = new xvsm.jsapi.interfaces.NotificationInterface();
  x.cref = cref;
  x.tab = tab;

  x.setNotificationId = function(id) {
    x.tab.messageNotificationId = id;
  };

  x.handleNotification = function() {
    var read = new ReadTakeDestroy(x.cref);
    jsapi.take(read, new getMessage());
    function getMessage() {
      this.doProcess = function(list){
        tab.addMessage(list[0].getValue());
      }
    };
  }
  return x;
}

/*  This method retrieves the list of users in the channel from the
 *  XVSM space.
 */
function getUserList(tab){
  var read = new ReadTakeDestroy(tab.channelCref, 0, null,
    [new RandomSelector(-1)]);
  jsapi.read(read, new setUserList(tab));

  function setUserList(tab) {
    this.tab = tab;

    this.doProcess = function(list){
      try {
        this.tab.clearUserlist();
        if (list != null) {
          for (var i = 0; i < list.length; i++) {
            this.tab.addUser(list[i].value[0].value);
          }
        }
      } catch (e) {
        console.debug(e);
      }
    };
  }
}
```

```
/*  This method sends a new message to the channel. New messages
 *  are stored in the containers of the users on the channel.
 */
function newMessage(c, msg) {
  var read = new ReadTakeDestroy(c, 0 );
  read.addSelector(new RandomSelector(-1));

  jsapi.read(read, new messageRead());

  function messageRead(){
    this.doProcess = function (list){
      for (var i = 0; i < list.length; i++) {
        var userContainer = list[i].value[1].value;
        var e = new Entry();
        e.setString(user + ": " + msg)
        e.setContainer(userContainer);

        var w = new WriteShift(-1, null, [e]);
        jsapi.write(w);
      }
    };
  }
}

/*  Perform login/logout events from the login/logout button.
 */
function loginlogout(){
  var button = dijit.byId("loginButton");
  if (button.containerNode.innerHTML == "login") {
    login();
  } else {
    logout();
  }
}

/*  Perform a login on the system with the chosen username.
 *  Retrieves the user container, then tries to write register the
 *  username.
 */
function login(){
  var name = dijit.byId("userName").textbox.value;
  if ((name !== null) && (name.length !== 0)) {
    if (usersCref == null) {
      getOrCreateContainer(users, new registerUser(name));
    } else {
      var r = new registerUser(name);
      r.doProcess(usersCref);
    }
  } else {
    alert("Username not set");
  }
}

/*  Sets the coordination type for the named container "user"
 *  and writes the username to it.
 */
function registerUser(name) {
  this.name = name;
  this.doProcess = function(cref) {
    usersCref = cref;

    var cp = new SetContainerProperties();
    cp.setContainerRef(cref);
    cp.addProperty(new ContainerProperty(
      ContainerProperties.COORDINATION_TYPES, "KEY", "SET"));
    jsapi.setContainerProperties(cp);
    var w = new WriteShift(0);
    var e = new Entry();
    e.setContainer(cref);
```

```
    e.setString(name);
    e.addSelector(new KeySelector(name, ValueTypes.STRING, name));
    w.addEntry(e);
    user = name;
    jsapi.write(w, userRegistered());
  }
};

/*  If the writing of the username was succesful, retrieve the list
 *  of available channels.
 */
function userRegistered() {
  var x = new xvsm.jsapi.interfaces.WriteShiftInterface();
  x.doProcess = function(){
    dijit.byId("userName").textbox.disabled = "disabled";
    dijit.byId("loginButton").setLabel("logout");
    getChannels();
  };

  x.timeoutExpiredException = function() {
    alert("This Username already exists. Please choose " +
        "another one");
  }
  return x;
};

/* Call shutdown to close all channel windows and disable buttons
 * to join channels. Clear the list of channels. Allow the user
 * to edit the nickname field and enable the login button.
 */
function logout(){
    shutdown();
    dijit.byId("loginButton").setLabel("login");
    dijit.byId("userName").textbox.disabled = "";
    dijit.byId("joinChannelButton").setDisabled(true);
    dijit.byId("createChannelButton").setDisabled(true);
    user = null;
    var channelwindow = dojo.byId("Channels");
    while (channelwindow.length > 0) {
      channelwindow.remove(0);
    }
}

/* Closes and destroys the channel window and removes notifications
 * and entries of the users container in this channel.
 */
function tabClosed(x,y) {
  jsapi.cancelNotification(y.messageNotificationId);
  jsapi.cancelNotification(y.userAddNotificationId);
  jsapi.cancelNotification(y.userRemoveNotificationId);
  var destroy = new ReadTakeDestroy(y.channelCref, -1, null,
    [new KeySelector(user, ValueTypes.STRING, user)]);
  jsapi.destroy(destroy);
  jsapi.destroyContainer(y.userCref);
  openChannels.remove(y.channelCref);
  x.removeChild(y);
  y.destroy();
}

/*  Closes all open channel windows. Cancels the notification
 *  listening for changes on the named container "channels".
 */
function shutdown() {
  var tab = dijit.byId("TabContainer");
  var childs = tab.getChildren();
  for (var i = 0; i < childs.length; i++) {
    var title = childs[i].title;
    if (title != "main") {
      tabClosed(tab, childs[i]);
```

```
      }
    }
    jsapi.cancelNotification(channelNotificationId);
    var e = new ReadTakeDestroy();
    e.addSelector(new KeySelector(user, ValueTypes.STRING, user));
    e.setContainerRef(usersCref);
    jsapi.destroy(e);
  }
```

Listing C.3: chat/widget/Channel.js

```
dojo.provide("chat.widget.Channel");

dojo.require("dijit.layout.ContentPane");
dojo.require("dijit.layout.LayoutContainer");
dojo.require("dijit.form.TextBox");

/*  This class implements the logic of the chat widget. The appearance is
 *  specified in the templates/Channel.html file. DojoAttachPoint attributes
 *  allow the direct manipulation of elements in the chat widget. DojoAttachEvents
 *  connect events with code blocks in this class.
 */
dojo.declare("chat.widget.Channel", [dijit.layout.LayoutContainer, dijit._Templated], {

    // only wors when the chat module is set. (see top of the chat.js file)
    templatePath: dojo.moduleUrl("chat", "./widget/templates/Channel.html"),

    // enables the usage of widgets in the template
    widgetsInTemplate: true,

    /*  With dojo custom widgets it is important to initalize variables.
     *  Otherwise they are recognised as static.
     */
    constructor: function() {
      this.channelName = "";
      this.channelCref = "";
      this.userCref = "";
      this.userAddNotificationId = "";
      this.userRemoveNotificationId = "";
      this.messageNotificationId = "";
    },

    /* Adds a user to the list of users on this channel.
     */
    addUser: function(name){
      this.user.innerHTML = this.user.innerHTML + "<br> " + name;
    },

    /* Clears the userlist.
     */
    clearUserlist: function() {
      this.user.innerHTML = "";
    },

    /* Adds a message to the channelwindow.
     */
    addMessage: function(msg) {
      this.display.setContent(this.display.domNode.innerHTML + "<br>" + msg);
    },

    /* Function called when clicking on the send button. Calls the send method
     * of the chat.js file.
     */
    send: function(){
      newMessage(this.channelCref, this.text.textbox.value);
    }
  }
);
```

Listing C.4: chat/widget/templates/Channel.html

```
<div dojoType="dijit.layout.LayoutContainer" style="width: 100%; height: 100%;">
  <div dojoType="dijit.layout.ContentPane" layoutAlign="right" style="width:150px;
    margin:5px; border: 1px solid grey;">
    <div style="top: 0px; text-align: center;">User</div>
    <div style="top: 20px; text-align: center;" dojoAttachPoint="user"></div>
  </div>
  <div dojoType="dijit.layout.ContentPane" layoutAlign="bottom" style="height:50px;
    margin:5px; border: 1px solid grey;">
    <table>
      <tr>
        <td><div dojoType="dijit.form.TextBox" dojoAttachPoint="text" size="50">
            </div>
        </td>
        <td><div dojoType="dijit.form.Button" dojoAttachEvent="onClick:send">Send
        </div>
        </td>
      </tr>
    </table>
  </div>
  <div dojoType="dijit.layout.ContentPane" layoutAlign="client" dojoAttachPoint="display"
    style="margin:5px; border: 1px solid grey;">
  </div>
</div>
```

List of Figures

Bibliography

[1] Ibm: Tspaces - intelligent connectionware. http://www.almaden.ibm.com/cs/TSpaces/. Last checked: Jan, 2008.

[2] Spacebasedcomputing.org. http://www.spacebasedcomputing.org. Last checked: 12.12.2007.

[3] Alessandro Alinone. Changing the web paradigm. http://www.lightstreamer.com/Lightstreamer_Paradigm.pdf, 2006. Last checked: 30.5.2007.

[4] Ryan Asleson and Nathaniel T. Schutta. *Foundations of AJAX*. Apress, Berkeley, CA, USA, 2006.

[5] Heri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[6] C. Bussler. A minimal triple space computing architecture. *in Procs. of the WIW'05 Workshop on WSMO Implementations*, 2005.

[7] Paolo Ciancarini, Andreas Knoche, Robert Tolksdorf, and Fabio Vitali. Pagespace: An architecture to coordinate distributed applications on the web. *in Proc. of Fifth International World Wide Web Conference*, 1996.

[8] Dave Crane and Eric Pascarello. *Ajax in Action*. Manning, Greenwich, CT, USA, 2006.

[9] D. Crockford. The application/json media type for javascript object notation (json). http://www.ietf.org/rfc/rfc4627.txt?number=4627, 2006. Last checked: 12.12.2007.

[10] Fielding et al. Hypertext transfer protocol - http/1.1. http://tools.ietf.org/html/rfc2616, 1999. Last checked: 02.12.2007.

[11] The Dojo Foundation. Dojo, the javascript toolkit. http://dojotoolkit.org/. Last checked: 15.11.2007.

[12] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice*. Pearson Education, 1999.

[13] Jesse James Garrett. A new approach to web applications. http://www.adaptivepath.com/publications/essays/archives/000385.php, 2005. Last checked: 29.4.2007.

[14] David Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[15] Inc. Internet Systems Consortium. Isc domain survey: Number of internet hosts. http://www.isc.org/ops/ds/host-count-history.php. Last checked: 2.12.2007.

[16] Sean Kelly. Speeding up ajax with json. http://www.developer.com/lang/jscript/article.php/3596836. Last checked: 24.5.2007.

[17] Eva Kühn, Johannes Riemer, and Geri Joskowicz. *XVSM (eXtensible Virtual Shared Memory) Architecture and Application*. Technical University of Technology, Vienna University of Technology, Austria, June 2005. Technical Report TU-Vienna.

[18] Eva Kühn, Johannes Riemer, and Lukas Lechner. Xvsmp/bayeux: A protocol for scalable space based computing in the web. *16th IEEE International*

Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, June 2007.

[19] Eva Kühn, Johannes Riemer, Richard Mordinyi, and Lukas Lechner. Integration of xvsm spaces with the web to meet the challenging interaction demands in pervasive scenarios. *Ubiquitous Computing and Communication Journal. Special Issue on Coordination in Pervasive Environments*, 2008.

[20] Markus Kühnel and Severin Ecker. *XVSM Core Architecture 0.7.* Technical Report E185/1, Vienna University of Technology, Austria, February 2007.

[21] Floyd Marinescu and Stefan Tilkov. Debate: Json vs. xml as a data interchange format. http://www.infoq.com/news/2006/12/json-vs-xml-debate, 2006. Last checked: 24.5.2007.

[22] Alex Russel, David Davis, Greg Wilkins, and Mark Nesbitt. Bayeux protocol – bayeux 0.1draft5. http://svn.xantus.org/shortbus/trunk/bayeux/bayeux.html, 2007. Last checked: 22.5.2007.

[23] Alex Russell. Comet - the next stage of ajax. http://www.irishdev.com/NewsArticle.aspx?id=2166, 2006. Last checked: 30.5.2007.

[24] Werner J. Schoenfeldinger. 'www meets linda' linda for global www-based transaction processing systems. *In Electronic Proc. 4th Int. World Wide Web Conference*, 1995.

[25] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems Principles and Paradigms.* Prentice Hall, Inc., Upper Saddle River, NJ, USA, 2002.

[26] P. Thompson. Ruple: an xml space implementation. *in Proc. of XML Europe 2002 Conference*, 2002.

[27] G.C. Wells. A tuple space web service for distributed programming. *in Proc. of 2006 Int. Conf. on Parallel & Distributed Processing Techniques and*

Applications, 2006.

[28] Greg Wilkins. Jetty continuations. http://docs.codehaus.org/display/JETTY/Continuations, 2006. Last checked: 22.5.2007.

[29] Greg Wilkins. Why ajax comet? http://www.webtide.com/downloads/whitePaperWhyAjax.html, 2006. Last checked: 22.5.2007.

[30] Dr. Phil Windley. Comet: Beyond ajax. http://www.irishdev.com/NewsArticle.aspx?id=2173, 2006. Last checked: 30.5.2007.

[31] Nicholas C. Zakas, Jeremy McPeak, and Joe Fawcett. *Professional Ajax*. Wiley Publishing, Inc., Indianapolis, IN, USA, 2006.