FernUniversität in Hagen
Fakultät für Mathematik und Informatik
Lehrgebiet Informationssysteme und Datenbanken

Kommunikationstechnologien für webbasierte Client-Server-Anwendungen

Diplomarbeit im Diplomstudiengang Informatik II

vorgelegt am 29.09.2011

von Oleg Varaksin
Matrikelnummer 5752558

Betreuer: Prof. Dr. Gunter Schlageter

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur mit den angegebenen Hilfsmitteln angefertigt habe. Wörtlich oder inhaltlich übernommene Literaturquellen wurden besonders gekennzeichnet.

Oleg Varaksin

D-78112 St. Georgen, Berliner Str. 6
den 22. September, 2011

Inhaltsverzeichnis

Abkürzungsverzeichnis

AJAX	Asynchronous JavaScript and XML
APE	Ajax Push Engine
API	Application Programming Interface
ARP	Asynchronous Request Processing
BOSH	Bidirectional-streams Over Synchronous HTTP
CPU	Central processing unit
DOM	Document Object Model
DWR	Direct Web Remoting
FTP	File Transfer Protocol
GAE	Google App Engine
GUI	Graphical User Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identifier
IO	Input / Output
Java EE	Java Enterprise Edition
JMS	Java Message Service
JSF	JavaServer Faces
JSON	JavaScript Object Notation
JSP	JavaServer Pages
JVM	Java Virtual Machine
Kbps	Kilobits per second
Mbps	Megabits per second
NIO	New non-blocking Input / Output
RIA	Rich Internet Applications
SMTP	Simple Mail Transfer Protocol
SVG	Scalable Vector Graphics
TCP/IP	Transmission Control Protocol / Internet Protocol
XMPP	Extensible Messaging and Presence Protocol
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VML	Vector Markup Language
W3C	World Wide Web Consortium
WWW	World Wide Web
XHR	XMLHttpRequest
XML	Extensible Markup Language

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation und Problemstellung

In der Vergangenheit war die Interaktion zwischen einem Benutzer und einer Web-Applikation sehr einfach gestaltet. Das Web bestand aus einer Sammlung von Dokumenten mit einer Eins-zu-Eins-Beziehung zwischen einer Seite und einer URL. Jede Seite, die vollständig geladen wurde, existierte nur in einem einzigen statischen Zustand. Heutzutage basieren moderne Web-Anwendungen auf komplexen Seitenstrukturen, die ihren Zustand und Informationsbestand dynamisch ändern können. Häufig muss der Server dem Client neue Ereignisse in Echtzeit mitteilen. Die Möglichkeit, Daten bidirektional mit Software-Systemen auszutauschen, bringt erhebliche Vorteile mit sich. Echtzeit-Kollaborationen sind dadurch möglich. Kollaborative Werkzeuge und Anwendungen ermöglichen eine bessere, effizientere Arbeit und Kommunikation innerhalb verteilter Gruppen oder eines Teams. Man kann beispielsweise mit Freunden in Echtzeit chatten, deren Aktivitäten verfolgen und gemeinsame Dokumente von mehreren Personen gleichzeitig bearbeiten. Im Bereich E-Learning entstehen kollaborative Lernumgebungen, browsergestützte Whiteboards und Mind-Maps, die eine enge Zusammenarbeit zwischen Studenten und Lehrern fördern. Als Auszug weiterer wichtiger Einsatzgebiete für die bidirektionale Kommunikation können Online Multiplayer-Spiele, Newsticker-Systeme und Web-Anwendungen für die Lieferung von Echtzeitdaten, wie Aktien-, Börsen- und Währungskurse genannt werden.

Die Grundlage für die bidirektionale Kommunikation zwischen Client und Server stellt das Remote Scripting zusammen mit den Server-Push-Technologien dar. Die konventionelle Kommunikation im WWW (*World Wide Web*) ist nicht bidirektional, weil sie auf dem HTTP (*Hypertext Transfer Protocol*) basiert - einem zustandslosen Einweg-Protokoll. Der Client initiiert eine Anfrage und der Server kann die entsprechenden Daten nur für diese Anfrage generieren und zurücksenden. Dieser Beschränkung des HTTP-Protokolls wird durch ausgeklügelte und intelligente Server-Push-Techniken entgegengesetzt, welche die Aufrechterhaltung einer aktiven Verbindung zu einem Webserver und das Warten auf eingehende Daten ermöglichen. Mit Hilfe dieser Techniken meldet sich der Server nun selbst, wenn neue Aktualisierungen verfügbar sind und schiebt die vorliegenden Daten durch eine offen gehaltene Verbindung zum Client. Die Informationen über den Fortschritt einer serverseitigen Verarbeitung lassen sich dadurch an einen Browser (Client) übermitteln. Web-Technologien für die bidirektionale Kommunikation fallen unter den Begriff „Web 2.0“, der die heutige Entwicklung des Internet prägt. Eines der Hauptziele des „Web 2.0“ Paradigma ist es, Web-Anwendungen in ihrer Bedienbarkeit und Reaktivität mit Desktop-Anwendungen gleichzusetzen. Die bidirektionale Kommunikation erhöht die Interaktion zwischen Nutzern und Web-Anwendungen und verbessert die Benutzbarkeit von solchen Software-Systemen enorm. Sie gibt dem Web neue Interaktionsmuster, die an lokale Anwendungsprogramme erinnern.

Die vorliegende Diplomarbeit ist den webbasierten Kommunikationstechnologien im Client-Server-Umfeld gewidmet. Der Kern der Diplomarbeit behandelt die bedeutendsten Technologien und Implementierungen für die bidirektionale Kommunikation. Sie reichen

von dem einfachen Polling-Verfahren, bei dem Anfragen vom Browser in sehr kurzen Abständen an den Server abgeschickt werden, bis hin zum neuen HTML5 WebSocket-Standard, der einen echten, bidirektionalen Kommunikationskanal im Browser bereitstellt. Das Ziel der Diplomarbeit besteht in der Untersuchung von existierenden Lösungen für die bidirektionale Kommunikation im Web. Ein weiterer Aspekt besteht in der Ausarbeitung einer Antwort auf die Leitfrage, welche der hier in der Arbeit behandelten Technologien für welchen Einsatzzweck geeignet sind und worin sich diese Technologien unterscheiden. Dafür sind verschiedene Vergleichskriterien zu erarbeiten. Eine Gegenüberstellung anhand dieser Kriterien soll auch die Vor- und Nachteile von unterschiedlichen Technologien herausstellen.

Im praktischen Teil der Diplomarbeit wird ein asynchrones, kollaboratives Online-Whiteboard prototypisch implementiert. Die Web-Anwendung beinhaltet eine interaktive Tafel, welche das gleichzeitige Zeichnen und Schreiben von mehreren Benutzern unterstützt. Des Weiteren gehören zur eigentlichen Implementierung die systematisch aufgearbeitete Spezifikation, Modellierung und Design der Applikation. Für die Umsetzung der Beispielanwendung wird ein passendes Framework ausgewählt, das unterschiedliche Kommunikationsprotokolle unterstützen kann. Es muss dabei begründet werden, warum man dieses Framework für die Umsetzung nimmt. Die Beispielanwendung wird mit mehreren bidirektionalen Technologien entwickelt, um sie analysieren und vergleichen zu können. Die Ergebnisse werden ausführlich diskutiert.

1.2 Aufbau der Arbeit

Nach der Einleitung werden die Grundlagen der Client-Server-Kommunikation in *Kapitel 2* beschrieben. Dieses Kapitel ist die Voraussetzung für das Verständnis der synchronen und asynchronen Datenübertragung. Asynchrone Kommunikation mit AJAX wird in *Kapitel 3* behandelt. AJAX stellt die Grundlage für die bidirektionale Client-Server-Kommunikation dar. *Kapitel 4* befasst sich mit den Basismechanismen für die bidirektionale Kommunikation und gibt einen umfassenden Überblick über die zur Zeit existierenden Protokolle und Techniken. *Kapitel 5* stellt den Kern der Diplomarbeit dar. Die vorgestellten bidirektionalen Technologien werden nach zuvor definierten Kriterien bewertet und beurteilt. Das Ziel dieses Kapitels ist die Beantwortung der Fragen, wann und wo die vorgestellten Technologien ihren sinnvollen Einsatz finden. *Kapitel 6* befasst sich mit den diversen Anforderungen für die Beispielanwendung „Whiteboard“. Die Beispielanwendung wird näher spezifiziert und modelliert. Anschließend findet die Vorstellung der Gesamtarchitektur statt. *Kapitel 7* beantwortet zuerst die Frage, welche Anforderungen von modernen Applikationsservern zu erfüllen sind, um die bidirektionale Kommunikation zu unterstützen. Es gliedert sich dann in die Entwicklung der client- und serverseitigen Modelle. Weiterhin wird die Web-Anwendung auf Basis drei verschiedener Implementierungsmöglichkeiten realisiert. Die Ergebnisse werden in *Kapitel 8* präsentiert. In einer Online-Demonstration wird ein Mehrbenutzerbetrieb simuliert. Dort werden Reaktionszeiten des aufgestellten Online-Whiteboards, neben anderen Messdaten, für die unterschiedlichen Server-Push-Ansätze ausgewertet. *Kapitel 9* schließt die Arbeit mit einer Zusammenfassung und dem Ausblick ab.

2 Client-Server Kommunikation

Bevor man sich mit den Kommunikationstechnologien für webbasierte Client-Server-Anwendungen befasst, sollte man die Grundlagen der Client-Server-Architektur kennen. In diesem Kapitel werden wichtige Begriffe des Client-Server-Konzeptes vorgestellt. Es wird speziell auf das grundlegende HTTP-Protokoll eingegangen, das eine besondere Bedeutung für die Interaktion webbasierter Systeme besitzt. Bei der Kommunikation zwischen Client und Server müssen in der Regel textuelle Nachrichten oder multimediale Inhalte ausgetauscht werden. Der Datenaustausch kann dabei synchron oder asynchron stattfinden. Diese zwei Arten des Datenaustauschs bilden den Gegenstand der letzten Unterkapitel, da sie eine wichtige Rolle für das weitere Verständnis der bidirektionalen Kommunikation spielen.

2.1 Definitionen

Das Client-Server-Modell beschreibt ein Standardparadigma zur Modellierung und Realisierung von verteilten Anwendungen [FRE10]. Das Client-Server-Modell ist das Konzept für die Verteilung von Aufgaben und Dienstleistungen innerhalb eines Netzwerkes. Aufgaben und Dienstleistungen werden mittels Server auf verschiedene Rechner verteilt und können bei Bedarf von mehreren Clients zur Lösung ihrer eigenen Aufgaben angefordert werden. Im Client-Server-Modell wird eine Aufgabe als Dienst bezeichnet. Typische Internet-Dienste sind z.B. World Wide Web, E-Mail, FTP, Newsgroups usw. Was sind aber genau ein Server und ein Client?

Ein Server ist ein Programm (Prozess), welches einen Dienst (Service) anbietet. Server sind somit Programme, die permanent darauf warten, dass eine Anfrage eintrifft, die ihren Dienst fordern. Ein Web-Server wartet z.B. auf Anfragen, Web-Seiten dem Anfragenden (Client) auszuliefern. In der Praxis laufen Server meist auf bestimmten Computern - sogenannten Hostrechnern, die anderen Rechnern ihre Dienste anbieten. Aus diesem Grund hat es sich eingebürgert, diese Rechner selbst als Server zu bezeichnen. Man spricht dann von Mail-Server, Web-Server, Datenbank-Server, File-Server usw.

Clients sind dagegen Software-Programme, die typischerweise Daten von Servern anfordern. Ein Web-Browser ist beispielsweise ein Client. Ein Klick auf einen Verweis führt dazu, dass der Browser, der als Client fungiert, eine Anfrage an den entsprechenden Server stellt. Der Server, der auf einem entfernten Hostrechner läuft, wertet die Anfrage aus und sendet die gewünschten Daten - hauptsächlich in Form einer HTML-Seite - zurück an den Browser. Im Rahmen des Client-Server-Konzepts nutzt der Client also einen Dienst des Servers.

Die Kommunikation zwischen Client und Server ist abhängig vom Dienst, d. h. der Dienst bestimmt, welche Daten zwischen beiden Programmen ausgetauscht werden. Wie bereits erwähnt wurde, fordert ein Client normalerweise die Daten bei einem Server an. Ein Client kann aber auch die Daten an einen Server schicken: zum Beispiel, wenn man per FTP eine Datei auf den Server-Rechner hochladet. Andere Beispiele sind das Versenden einer E-Mail und das Abschicken eines ausgefüllten Formulars im Web. Der Client „drängt“ dem Server

diverse Daten auf. Diese Aktionen bezeichnet man als Client-Push. Ein entgegengesetztes Szenario ist, wenn der Server aktiv wird und dem Client etwas ohne dessen Anforderung zuschickt. Diesen Prozess nennt man Server-Push. Die Server-Push-Technologien sind insoweit interessant, als sie zahlreiche Broadcasting-Dienste ermöglichen sollten. Ein Client kann für ihn interessante Daten abonnieren und sie regelmäßig empfangen, ohne diese ständig anzufordern.

Clients und Server können als Programme auf verschiedenen Rechnern oder auf demselben Rechner ablaufen. Wichtig ist dabei jedoch, die Bezeichnungen Client und Server getrennt bzw. in verschiedenen Kontexten zu betrachten. In einem Peer-to-Peer-Netzwerk - welches einen anderen Netzwerktyp darstellt - sind alle Computer gleichberechtigt und können sowohl Dienstanbieter als auch Dienstnutzer sein. In dem Peer-to-Peer-Modell stellt ein beteiligtes Programm gleichzeitig einen Client und einen Server dar. Die nachfolgenden Abbildungen veranschaulichen die Unterschiede [PPURL].

Abbildung 1: Client-Server-Modell

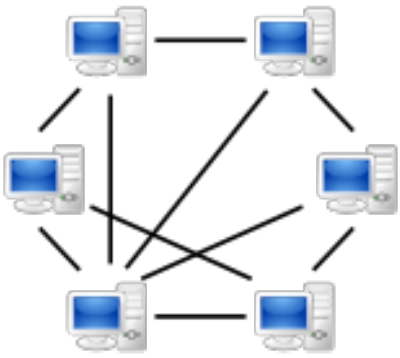

Abbildung 2: Peer-to-Peer-Modell

2.2 Datenübertragung mit HTTP-Protokoll

Um die Kommunikation zwischen Clients und Servern zu regeln, gibt es entsprechende Protokolle. Die Client-Server-Kommunikation im Web regelt das HTTP-Protokoll. Ein solches Protokoll läuft oberhalb des TCP/IP-Protokolls (*Transmission Control Protocol / Internet Protocol*) ab [WON00]. TCP/IP ist der Standard-Protokoll-Stack auf jedem Betriebssystem für die Anbindung an das Internet. HTTP ist das am häufigsten verwendete Web-Protokoll, dessen Ziel der lokalisierte Datentransfer dank einer sogenannten URL Zeichenkette ist. Das HTTP-Protokoll ist zustandslos. Mehrere HTTP-Anfragen werden grundsätzlich als voneinander unabhängige Transaktionen behandelt. Insbesondere werden sie ohne Bezug zu früheren Anfragen behandelt und es werden keine Sitzungsinformationen ausgetauscht und/oder verwaltet. Ein zuverlässiges Mitführen von Sitzungsdaten eines Benutzers kann erst in der Anwendungssoftware durch eine Sitzungs-ID (*session identifier*) implementiert werden. Die Sitzungs-ID ist ein Identifikationsmerkmal, um mehrere zusammengehörige Anfragen eines Benutzers zu erkennen und einer Sitzung (stehende Verbindung eines Clients mit einem Server) zuzuordnen.

Die Kommunikationseinheiten in HTTP werden als Nachrichten bezeichnet. Es gibt zwei unterschiedliche Arten von Nachrichten: die Anfrage (engl. *Request*) vom Client und die Antwort (engl. *Response*) des Servers. Jede HTTP-Anfrage besteht dabei aus drei Teilen.

- *Anfragezeile*. Diese Zeile gliedert sich in drei Teile: die anzuwendende HTTP-Methode, die URL und die vom Client verwendete Protokollversion. Es gibt insgesamt acht HTTP-Methoden [GOY02, 53ff]. Die drei für diese Diplomarbeit wichtigsten Methoden sind in der untenstehenden Tabelle aufgeführt.

GET	Mit ihr wird eine Ressource unter Angabe einer URL vom Server angefordert. Die Client-Daten sind ein Teil der URL. Die Länge der URL ist begrenzt.
POST	Mit ihr werden unbegrenzte Mengen an Daten zur weiteren Verarbeitung an den Server geschickt. Diese werden als Inhalt der Nachricht übertragen.
HEAD	Mit ihr wird, wie bei GET, eine Ressource unter Angabe einer URL vom Server angefordert. Es wird jedoch nur noch der Nachrichtenkopf (HTTP-Header) und nicht der eigentliche Dokumenteninhalt an den Client gesendet.

- *Nachrichtenkopf der Anfrage* (engl. *Message Header* oder auch *HTTP-Header* genannt). Der Nachrichtenkopf enthält Informationen über den Nachrichtenkörper wie etwa verwendete Kodierungen oder den Inhaltstyp, damit dieser vom Empfänger korrekt interpretiert werden kann. Auch zusätzliche Informationen, wie der Browsertyp und das Betriebssystem können darin enthalten sein.
- *Nachrichtenkörper der Anfrage* (engl. *Message Body*). Der Nachrichtenkörper enthält die Nutzdaten, wie beispielsweise Formulardaten, die mit einem POST-Befehl zum Server übertragen werden.

Ein Beispiel einer GET-Anfrage:

```
GET http://fractalsoft.net HTTP/1.0
Accept : text/html
If-Modified-Since : Saturday, 30-January-2011 14:37:11 GMT
User-Agent : Mozilla/4.0 (compatible; MSIE 6.0; Windows XP)
```

Jede HTTP-Antwort besteht ebenso aus drei Teilen.

- *Statuszeile*. Diese Zeile gliedert sich in drei Teile: Version des angewendeten Protokolls, Statuscode und Erläuterung des Statuscodes. Eine Liste der HTTP-Statuscodes ist fest definiert und gut bekannt [SCURL]. Die nachfolgende Tabelle zeigt exemplarisch einige gängige Statuscodes mit Erläuterungen.

200 OK	Die Anfrage wurde erfolgreich bearbeitet und das Ergebnis der Anfrage wird in der Antwort übertragen.
304 Not Modified	Der Inhalt der angeforderten Ressource hat sich seit der

	letzten Abfrage des Clients nicht verändert und wird deshalb nicht übertragen.
401 Unauthorized	Die Anfrage kann nicht ohne gültige Authentifizierung durchgeführt werden.
404 Not Found	Die angeforderte Ressource wurde nicht gefunden.
500 Internal Server Error	Ein unerwarteter Serverfehler ist aufgetreten.

- *Nachrichtenkopf der Antwort.* Dieser Teil besteht aus einer wahlfreien Zeilenfolge mit zusätzlichen Angaben über die Antwort und den Server.
- *Nachrichtenkörper der Antwort.* Dieser Teil enthält das eigentliche Dokument.

Ein Beispiel einer HTTP-Antwort:

```
HTTP/1.0 200 OK
Date: Fri, 30 Jan 2011 15:12:48 GMT
Last-Modified: Tue, 10 Jan 2011 11:18:20 GMT
Content-Language: de
Content-Type: text/html; charset=utf-8

<html>
  <head>
    <title>Hypertext Transfer Protocol</title>
  </head>
  <body>
    Now that both HTTP extensions and HTTP/1.1 are stable
    specifications, W3C has closed the HTTP Activity...
  </body>
</html>
```

In diesem Unterkapitel sei noch anzumerken, dass die Elemente einer URL fest vorgegeben sind. Eine URL kann in ihre Einzelteile zerlegt werden. Das sind Schema, Rechnername, Portnummer, Pfadangabe, Query-String und Fragment. Das nachfolgende Beispiel zeigt die Einzelteile einer URL.

```
http://example.org:8080/demo/example.cgi?land=de#geschichte
  |          |       |       |                |           |
Schema Rechnername Port  Pfadangabe  Query-String Fragment
```

2.3 Synchroner Datenaustausch

Der Prozessfluss einer traditionellen Web-Anwendung ist in den meisten Fällen synchronisiert. Bei einem synchronen Datenaustausch werden Anfragen und Antworten jeweils vollständig und nacheinander abgearbeitet. Sender und Empfänger von Daten synchronisieren die Kommunikation, d.h. sie warten, bis der Request-Response-Zyklus vollständig abgeschlossen ist. Bei einer synchronen Client-Server-Kommunikation ist der Client solange blockiert, bis der Server den Request angenommen und bearbeitet hat und die

Server-Response beim Client angekommen ist. Verzögert sich die erforderliche Antwort des Servers, entstehen unweigerlich längere Wartezeiten oder sogar Abbrüche im Ablauf der Anwendung. Als Beispiel sei das Hochladen einer Datei über eine Web-Anwendung, die im Browser bedient wird, genannt. Ist die zu übertragende Datenmenge sehr groß, muss ein Benutzer bei synchroner Kommunikation bis zum Abschluss des Datentransfers warten, um weitere Aktionen auf der Seite zu tätigen.

Nachteile einer synchronen Kommunikation liegen auf der Hand. Aufgrund der Tatsache, dass der Webserver bei jeder Anfrage seitens des Nutzers eine neue Webseite erzeugen und übermitteln muss, erscheint die Web-Anwendung, im Vergleich zu einer gewöhnlichen Desktop-Anwendung, insgesamt als träge und wenig intuitiv. Die Vorteile dieses Kommunikationskonzepts liegen bei Anwendungen, die eine Nutzung gemeinsamer Ressourcen erfordern und mehrere Zugriffe synchronisieren müssen. Denn bei der synchronen Kommunikation muss zuerst eine Rückmeldung vom Server vorliegen, bis weitere Aktionen vom Client erfolgen können. Außerdem wird so jede Anfrage immer hinsichtlich ihrer Korrektheit verifiziert.

In der nachfolgenden Grafik [NAURL] werden clientseitige Benutzerinteraktionen und serverseitige Reaktionen für eine klassische Web-Anwendung ohne AJAX-Technologie dargestellt (AJAX wird im Kapitel 3 beschrieben). Der Ablauf ist synchron. Daher muss der Benutzer bis zur Verarbeitung der Server-Antwort warten, um weiterarbeiten zu können.

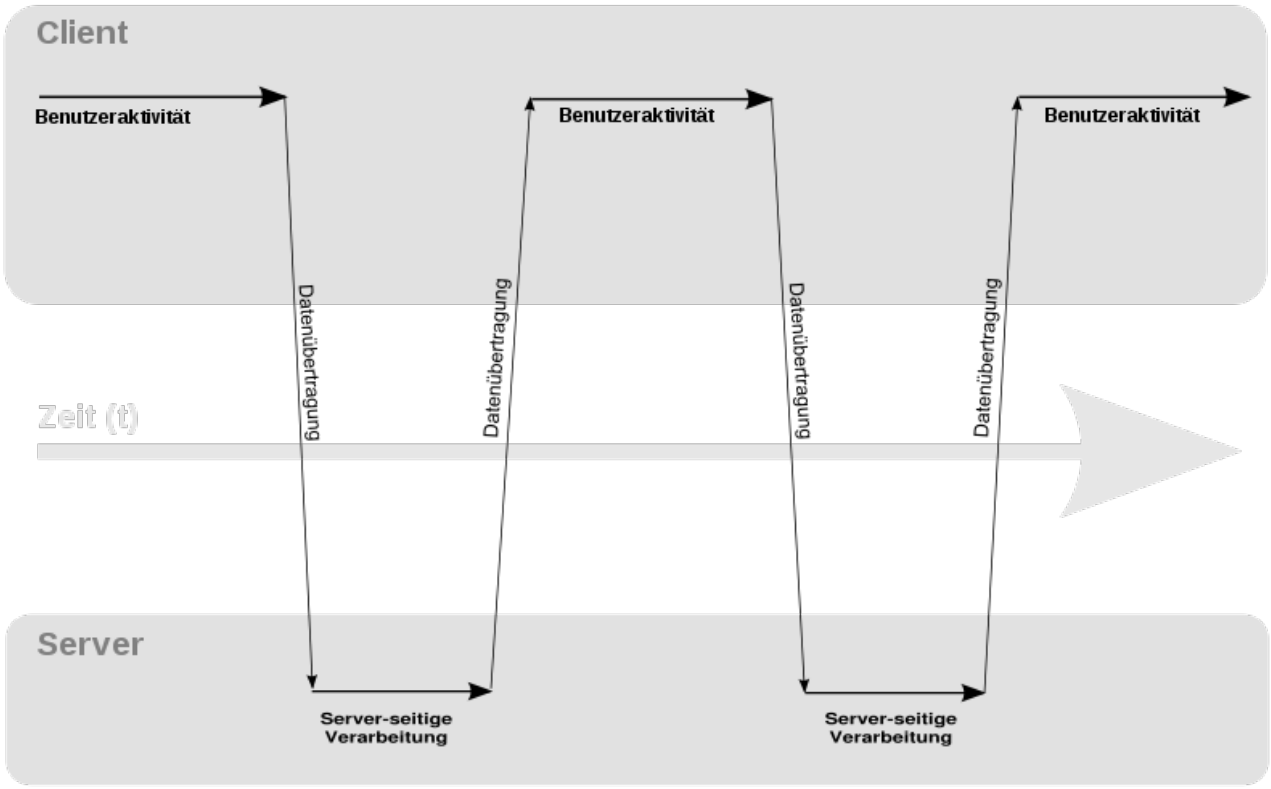

Abbildung 3: Synchrone Interaktion

Die synchrone Kommunikation stammt aus den Anfängen des Web und wurde früher ausgiebig eingesetzt, weil die nötigen Mechanismen für den asynchronen Datenaustausch fehlten.

2.4 Asynchroner Datenaustausch

Mit dem Beginn von Web 2.0 und RIA (*Rich Internet Applications*) rückte die asynchrone Kommunikation in den Vordergrund. Asynchroner Datenaustausch bringt mehr Dynamik, indem Anfragen asynchron verarbeitet werden und ein clientseitiges Skript, welches die Anfrage initiiert hat, somit nicht warten muss, bis die Anfrage beantwortet wurde. Die Webseite blockiert nicht die Interaktion des Benutzers mit dem Browser und ermöglicht es ihm so weiterzuarbeiten. Ein asynchroner Prozess ermöglicht dadurch eine flüssige und benutzerfreundliche Arbeitsweise. Technisch gesehen kapselt der Browser jede Anfrage an den Server in einem eigenen *Thread* (Teil oder Ausführungsstrang eines Prozesses) und blockiert somit die Webseite nicht, wie das nachfolgende Bild 4 veranschaulicht [NAURL].

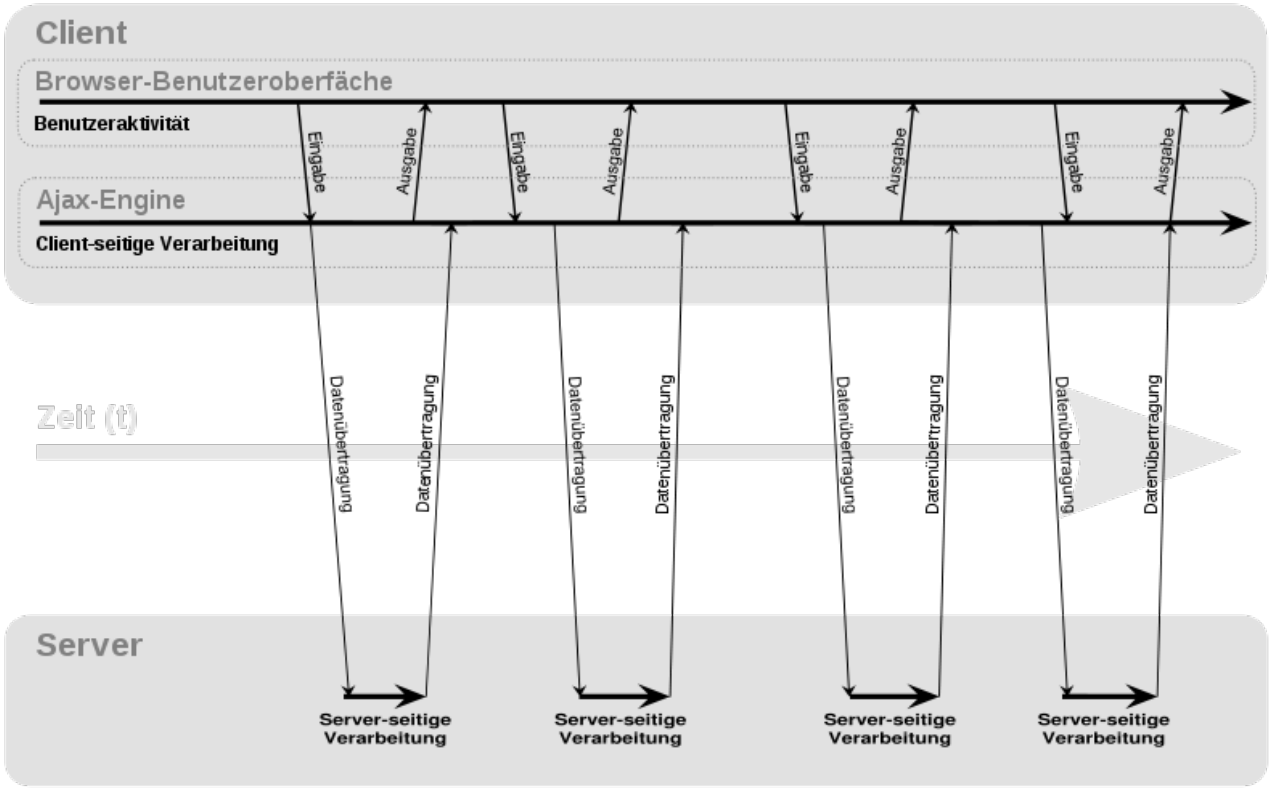

Abbildung 4: Asynchrone Interaktion

Eine Möglichkeit, wie man die asynchrone Kommunikation realisieren kann, besteht in der Benutzung von AJAX (*Asynchronous JavaScript and XML*). AJAX bezeichnet ein enorm wichtiges Konzept der asynchronen Datenübertragung zwischen einem Client (Browser) und einem Server. Eine ausgeklügelte AJAX Engine (spezielle JavaScript-Bibliothek) sendet und empfängt die Daten asynchron zu den entsprechenden Benutzerinteraktionen. Inzwischen existieren sehr viele ausgereifte AJAX-Frameworks [AFURL]. Webdienste, wie Google Suggest, Google Mail und Google Maps machen Gebrauch von den Möglichkeiten

der Web-Technik AJAX. Betrachten wir beispielweise Google Suggest - eine Erweiterung der Internet-Suchmaschine Google, bei der während des Tippens eines Suchworts bereits Vorschläge in Echtzeit erscheinen. Bei der Datenübertragung ist kein erneuter Seitenaufruf im Browser nötig. Die eingetippten Buchstaben des Suchwortes werden unmittelbar im Hintergrund zum Server via AJAX übertragen. Das Ergebnis wird vom Server zurückgeschickt und in Echtzeit in die bestehende Seite eingefügt. Dieser Vorgang wiederholt sich bei jedem Tastenanschlag und die Vorschläge werden während des Tippens angepasst [Abbildung 5]. Noch weiter geht Google Instant, bei der die Suchmaschine mit einer neuartigen Echtzeit-Funktion beschleunigt wird. Noch während ein Anwender den Suchbegriff eintippt, erscheinen Suchergebnisse zu den eingegeben Wortfragmenten. Die Zeitverzögerung ist nicht zu merken und man kann schon während des Tippens gleichzeitig Suchergebnisse überblicken. Weitere typische Einsatzzwecke asynchroner Kommunikation mit AJAX sind permanente Validierung von Eingabefeldern, Nachladen von Tooltips, Lesen von RSS-Feeds, Editieren von Texten ohne explizites Absenden (*Instant Edit*).

Abbildung 5: Google Suggest

Im nächsten Kapitel betrachten wir die AJAX-Technik wegen ihrer enormen Bedeutung etwas detaillierter mit Codefragmenten und stellen zahlreiche Möglichkeiten zum dynamischen Nachladen von Ressourcen vor.

3 Asynchrone Kommunikation mit AJAX

Das erste Unterkapitel gibt eine kleine Einführung in das Entstehen von AJAX und unterstreicht seine wichtige Bedeutung. Das nächste Unterkapitel befasst sich mit der AJAX-Architektur und beschreibt das `XMLHttpRequest` Objekt. Die Technik basierend auf `XMLHttpRequest` ist nicht die enzige zum Nachladen von Seitenfragmenten und für die bidirektionale Kommunikation. Alternative Techniken bilden den Gegenstand des dritten Unterkapitels. Die Grenzen von AJAX und die Kritikpunkte werden bei der Bewertung der Kommunikationstechnologien (Kapitel 5) aufgezeigt.

3.1 Geschichtliche Entwicklung und Bedeutung von AJAX

AJAX bereichert das Web mit neuen Interaktionsmustern: Nebenläufigkeit, ausgeklügelte visuelle Interaktionen und asynchrone Aktualisierung von einzelnen Seitenbestandteilen ohne Neuladen der kompletten HTML Seite. Die Möglichkeit partieller Aktualisierung einer Seite macht auf AJAX basierende Techniken einzigartig. AJAX-Anwendungen erwecken den Eindruck, dass sie komplett auf dem Computer des Anwenders ausgeführt werden. AJAX ist aus technischer Sicht eine Art Bestandteil von Web 2.0. RIA haben dank AJAX einen großen Schwung erlebt, weil das Web mit AJAX dynamischer wurde. AJAX ist auch die Grundlage für Server-Push-Technologien, wie es noch weiter in der vorliegenden Arbeit zu sehen sein wird.

Den Begriff AJAX hat Jesse James Garrett (Mitarbeiter der Agentur *Adaptive Path*) in seinem Aufsatz "Ajax: A New Approach to Web Applications" [NAURL] eingeführt. Nach dem veröffentlichten Artikel wurde AJAX durch den erhöhten Einsatz bei großen, bekannten Firmen noch weiter popularisiert und etabliert. AJAX an sich ist keine neue Technologie, sondern ein Sammelbegriff für bestimmte Techniken zur effizienten Ausnutzung bestehender Funktionalitäten im World Wide Web. Der andere Begriff für AJAX, der schon vor dem AJAX-Hype bekannt war, ist Remote Scripting. Technologische Grundlagen und zugrundeliegende Ideen für AJAX / Remote Scripting waren bereits seit etwa 1998 bekannt und haben hauptsächlich auf dem `XMLHttpRequest` Objekt [XRURL] basiert. So hat sich hinter existierenden Webseiten bereits eine große Portion der AJAX-Technologie verborgen, ohne dass dies dem Betrachter unmittelbar bewusst wurde. Somit sind für AJAX keine Browser-Plugins oder gar neue Browser erforderlich.

`XMLHttpRequest` (Abkürzung: `XHR`) ist ein JavaScript-Objekt, das ursprünglich von *Microsoft* stammt, von *Mozilla* weiterentwickelt wurde und jetzt durch das W3C (*World Wide Web Consortium*) standardisiert ist. Über das `XMLHttpRequest`-API (*Application Programming Interface*) können beliebige Daten über das HTTP(S) Protokoll transportiert werden. Auch andere Protokolle, wie z.B. FTP, werden unterstützt. Eine Besonderheit von `XMLHttpRequest` besteht darin, dass Anfragen asynchron verarbeitet werden können. Dank dieser Eigenschaft bildet `XMLHttpRequest` den Kern nahezu jeder AJAX-Anwendung. Das Format für den Datenaustausch beschränkt sich nicht nur auf XML. Daten als Klartext (*plain text*) oder im JSON-Format [JNURL] können ebenso übertragen werden. Das JSON-Format wird bei der Modellierung der Beispielanwendung “Whiteboard”

vorgestellt [Kapitel 6] und bei der Implementierung des Whiteboards eingesetzt [Kapitel 7].

3.2 AJAX-Architektur

Ein typischer Aufbau einer AJAX-Anwendung wird in vielen Büchern beschrieben [BER05], [GRO07], [CRA07]. Wie schon erwähnt wurde, ist `XMLHttpRequest` ein spezielles Objekt, das heutzutage in allen modernen Browsern verfügbar ist. Drei Schritte sind notwendig, um eine Anfrage (HTTP-Request) abzusetzen.

1. `XMLHttpRequest` instanziieren.
2. Callback-Funktion bereitstellen, die immer dann aufgerufen wird, wenn sich der Bearbeitungszustand der aktuellen Anfrage (Transaktion) ändert.
3. Anfrage senden.

Der erste Schritt besteht aus einem Aufruf:

```
var xhr = new XMLHttpRequest();
```

Im Internet Explorer 6 und älteren Versionen dieses Browsers wurde die `XHR`-Funktionalität durch ein `ActiveX` Objekt implementiert, so dass wir hier eine spezielle Behandlung für den Fall brauchen. Wir werden eine plattformunabhängige Funktion zur Erzeugung eines `XMLHttpRequest`-Objekts schreiben, die wie folgt aussieht:

```
function createXMLHttpRequest() {
  var i, xhr, activeXids = [
    'MSXML2.XMLHTTP.3.0',
      'MSXML2.XMLHTTP',
      'Microsoft.XMLHTTP'
  ];

  if (typeof XMLHttpRequest === "function") { // native XHR
    xhr =  new XMLHttpRequest();
  } else { // IE before 7
    for (i = 0; i < activeXids.length; i += 1) {
      try {
        xhr = new ActiveXObject(activeXids[i]);
        break;
      } catch (e) {}
    }
  }

  return xhr;
}
```

Es werden hier nacheinander die verschiedenen Möglichkeiten zur Erzeugung des `XMLHttpRequest`-Objekts probiert, wobei die `ActiveX`-Varianten jeweils in einen `try/catch`-Block für die Behandlung von Exceptions eingeschlossen sind. Damit sieht der

erste Schritt jetzt so aus:

```
var xhr = createXMLHttpRequest();
```

In dem zweiten Schritt wird eine Callback-Funktion für Statusänderungen in der Variablen `onreadystatechange` eingetragen.

```
xhr.onreadystatechange = handleResponse;
```

Die angegebene Funktion `handleResponse` wird für jede Statusänderung des `xhr` Objekts aufgerufen. Die Eigenschaft `readyState` des `XMLHttpRequest`-Objekts gibt Aufschluss über den aktuellen Status der Transaktion beim Aufruf dieser Callback-Funktion. So lassen sich einzelne Phasen der Datenübertragung unterscheiden.

```
function handleResponse() {
  if (xhr.readyState !== 4) {
    return false;
  }

  if (xhr.status !== 200) {
    alert("Error, status code: " + xhr.status);
    return false;
  }

  document.body.innerHTML +=
    "<pre>" + xhr.responseText + "<\/pre>";
};
```

Die oben stehende Callback-Funktion überprüft die Eigenschaft `readyState.` Es gibt vier mögliche Werte dieser Eigenschaft – von 0 bis 4. Der Wert 4 bedeutet, dass die Bearbeitung der aktuellen Anfrage abgeschlossen ist. Für alle anderen Werte macht die Callback-Funktion nichts und wartet nur auf das nächste `readystatechange` Event. Wenn die Bearbeitung der Anfrage abgeschlossen ist, wird die Eigenschaft `status` überprüft. Diese Eigenschaft stimmt mit dem HTTP Status Code überein - zum Beispiel 200 (OK) oder 404 (Not found). In dem Beispiel sind wir nur an dem Response Code 200 interessiert und melden alle anderen Status-Codes als Fehler (*alert box*). Mögliche Werte für `XMLHttpRequest.readyState` sind in der nachfolgender Tabelle aufgelistet.

Wert	Bezeichnung	Bedeutung
0	UNINITIALIZED	Das Objekt wurde noch nicht initialisiert, es erfolgte noch kein Aufruf der Funktion `open`.
1	LOADING	Das Request-Objekt wurde initialisiert, aber der Request noch nicht mittels `send` abgesetzt.
2	LOADED	Der Request wurde mittels der Funktion `send` abgesetzt.
3	INTERACTIVE	Teile der Antwort sind bereits verfügbar. Über das Feld `responseText` kann auf die empfangenen Daten

		zugegriffen werden.
4	COMPLETED	Die Bearbeitung des Requests ist beendet.

Der letzte Schritt besteht in dem Senden des Requestes mittels zwei Methoden `open()` und `send()`.

```
xhr.open("GET", "http://myhost.org/myapp/mypage.php", true);
xhr.send();
```

Die Methode `open()` setzt die HTTP(S) Methode für den Request (z.B. GET, POST) und die URL fest. Die Methode `send()` übergibt beliebige POST Daten oder gar keine Parameter im Falle von GET. Der letzte Parameter der Methode `open()` spezifiziert, ob der Request asynchron ist (`true`) oder nicht (`false`). Bei asynchronen Requests setzt der Browser die Ausführung des laufenden Skripts nach den Aufrufen von `open()` und `send()` fort, ohne den Ablauf zu blockieren. Der letzte Parameter der `open()` Methode sollte - falls keine anderen Gründe dagegen sprechen - immer auf `true` gesetzt sein. Dies bewirkt eine Steigerung der Benutzerfreundlichkeit, da der Benutzer auf keine Server-Antwort warten muss.

3.3 Remote Scripting ohne XMLHttpRequest

Moderne Browser bieten zahlreiche Möglichkeiten zum dynamischen Nachladen von Ressourcen. Im allgemeinen Fall wird das Attribut `src` (source) eines (I)Frame-, Script-, oder sogar Image-Elements, wie `frame`, `iframe`, `script`, `img`, zur Laufzeit von einer JavaScript-Funktion verändert. Um einen GET-Request an einen Webserver abzusetzen, reicht das Setzen einer URL in diesem Attribut aus. Für einen POST-Request ist eine Erzeugung eines Web-Formulars erforderlich, das dann als Basis für den Request verwendet wird.

Die einfachste Form für das Remote Scripting ist es, wenn wir Daten mit einer Anfrage an einen Webserver schicken und keine Antwort erwarten oder nur an kleinen Datenmengen interessiert sind, die in HTTP-Cookies (kleine Textinformationen vom Webserver, die im Speicher des Browsers gespeichert werden) übertragen werden. In diesem Fall können wir ein Image erzeugen und das `src` Attribute auf eine URL verweisen. Auf diese Weise kann man das Absetzen eines Requests erzwingen.

```
new Image().src = "http://myhost.org/myapp/mypage.php";
```

Als Anwort kann der Server ein 1x1 px Bild schicken oder aber eine Antwort mit dem Statuscode „204“ erzeugen. Der Statuscode „204“ bedeutet „No Content“ und der Dokumenteninhalt wird nicht verändert. Es findet also keine Aktualisierung der Anzeige statt. Auf diese Weise lässt sich der Wert eines Cookies verändern, weil die Header-Felder einer Antwort (HTTP-Response) durchaus aktualisierte Informationen tragen können. Auf Nachteile dieser und nachfolgender Lösungen wird in dem nächsten Kapitel eingegangen.

Die zweite Möglichkeit zum dynamischen Nachladen von Ressourcen ist die Manipilation

des `src` Attributes eines `script` Elementes. Mit `XHR` ist es nicht ohne weiteres möglich, externe Daten mit AJAX aufzurufen. Dabei gilt es, die jeweiligen Sicherheitsrichtlinien der Browser zu beachten. Würden die Browser keine Überprüfung vornehmen, wäre es leicht möglich, durch das Einbeziehen von fremden Inhalten kritische Aktionen herbeizurufen. Im Gegensatz zu `XHR` ist das Nachladen mit einem `script` Element nicht auf den selben Host (Domain) eingeschränkt. Es ist erlaubt mit dieser Technik, die Daten von fremden Hosts einzubeziehen. Obwohl beliebige Datentypen in einem Response unterstützt werden, wird am meisten JSON verwendet. In der URL wird eine Callback Funktion spezifiziert, die mit den zurückgelieferten Daten eingespeist und aufgerufen wird. Ein Beispiel:

```
http://myhost.org/myapp/mypage.php?callback=myHandler
```

Die URL wird in ein statisches oder dynamisches `script` Element hereingeladen.

```
var script = document.createElement("script");
script.src = url;
document.body.appendChild(script);
```

JSON Daten werden als Parameter an die zuvor definierte Callback Funktion übergeben und das Endergebnis resultiert in einem Funktionsaufruf, wie z.B.

```
myHandler({"hello": "world"});
```

Ein sehr verbreitetes Verfahren für das asynchrone Nachladen von Ressourcen basiert auf `iframe`-Elementen (sogenannte *inline frames*). Der Elementtyp `iframe` ist in der Transitional HTML-Variante definiert und dient zur Einbettung von Ressourcen innerhalb eines HTML-Dokumentes. Die Ressource wird mit dem Attribut `src` referenziert, das mittels JavaScript manipuliert werden kann. Man erzeugt normalerweise einen nicht sichtbaren, randlosen `iframe` mit einer Ausdehnung von 0 Pixeln irgendwo im Rumpf des HTML-Dokuments. Das Nachladen einer Ressource wird über ein externes Ereignis angestoßen – wie z. B. die Betätigung eines Buttons oder das Klicken auf einen Textabschnitt. Zurückgegeben wird normalerweise ein `script`-Element mit einem beliebigen JavaScript-Code. Es wird dabei eine JavaScript-Funktion aufgerufen, die im Hauptdokument definiert wurde. Dadurch lassen sich Callback-Funktionen realisieren, ähnlich wie mit `XMLHttpRequest`. Der folgende Beispielcode veranschaulicht das eben beschriebene:

```
<html>
  <head>
  <script type="text/javascript">
    function myHandler(result) {
      alert(result);
    }

    function loadUrl(element, url) {
      document.getElementById(element).src=url;
    }
```

```
  </script>
  </head>
  <body>
    <iframe id="ifr" name="ifr" src="#"
            style="width:0px; height:0px; border: 0px">
    </iframe>
    <input type="button"
           onclick="loadUrl('ifr', 'mypage.html')"/>
  </body>
</html>
```

Der Button verfügt über ein Attribut `onclick`, in dem die Funktion `loadUrl` aufgerufen wird. Als Argumente werden der eindeutige Bezeichner des versteckten `iframe` und die URL der nachzuladenden Ressource übergeben.

```
<script type="text/javascript">
  window.parent.myHandler('Hello World!');
</script>
```

Im Response auf die Anfrage der Seite `mypage.html` wird eine Callback-Funktion `myHandler` aufgerufen, die im Hauptdokument definiert wurde. Das Ergebnis serverseitiger Operation wird als Argument an diese Funtion übergeben. Das Ergebnis kann z.B. eine Zusammenstellung von Texten (Einträgen) für eine Auswahlliste darstellen - wie im Beispiel von Google Suggest gezeigt [Abbildung 5].

4 Basismechanismen für die bidirektionale Kommunikation

Eine klassische Interaktion zwischen einem Benutzer und einer Web-Applikation läuft in der Regel folgendermaßen ab: der Benutzer startet einen Browser und tippt eine URL ein, wenn er ein Dokument sehen will. Will er einen Blog-Eintrag verfassen oder einen Kommentar hinterlassen, muss er dasselbe machen – Browser starten, ein Formular (HTML Form) ausfüllen und den Submit-Button drücken. Unternimmt der Benutzer nichts, bleibt die aktuelle Webseite unverändert und so besteht die Gefahr, dass die Daten schnell veraltet sind, ohne dass der Benutzer davon erfährt. Heutzutage hat sich das Paradigma geändert – von einem „*website-centric*“ Modell, bei dem die Webseite im Mittelpunkt aller Interaktionen stand, zu einem „*user-centric*“ Modell [BRU10, 2ff]. Alle Interaktionen starten und enden jetzt bei dem Benutzer. In der heutigen, modernen Welt ist es immer wichtiger, seine Daten und deren Veränderung in Echtzeit abfragen und analysieren zu können. Echtzeitlösungen sind gefragt und werden derzeit intensiv diskutiert. Solche Echtzeitlösungen basieren auf bidirektionaler Kommunikation zwischen Client und Server und werden durch neuartige Webtechnologien - z.B. Server-Push-Technologien - implementiert. Das serverseitige Senden der Daten an den Client (Browser) widerspricht zwar dem klassischen Web-Paradigma vom Request-Response-Ablauf, eröffnet aber neue Wege für die reichhaltige Interaktivität [MOS09].

In den nachfolgenden Kapiteln werden die grundlegenden Mechanismen und Architekturen für die bidirektionale Kommunikation aufgezeigt, ohne sie zu vergleichen und zu bewerten. Eine tiefgehende Diskussion über die Vor- und Nachteile jeder einzelnen Technik findet in dem Kapitel 5 statt.

4.1 HTTP Polling

HTTP Polling ist eine recht einfache und populäre Methodik, den Server in regelmäßigen Abständen abzufragen. Mit Polling fragt eine AJAX-Anwendung den Server jedes Mal ab, wenn der Inhalt einer Seite eine Aktualisierung (*Update*) benötigt. Eine Chat-basierte Anwendung wird beispielsweise den Server regelmäßig im 10-sekunden Takt abfragen, um Nachrichtenaktualisierungen abzurufen. Technisch bedeutet das, dass der Browser jedes Mal eine Verbindung zum Server öffnet, sobald er den Client mit neuen Informationen versorgen will. Das nachfolgende Bild 6 veranschaulicht den Request-Response-Ablauf bei dieser Technik [NEURL].

Es gibt auch eine optimierte Variante des HTTP Pollings, wenn der Ereignisabstand bekannt ist. In diesem Fall kann die Poll-Distanz auf den Abstand der Ereignisse festgelegt werden. Diese Polling-Variante heißt *PredicatedPoll* [ROC09, 20f]. Mit *PredicatedPoll* wird davon ausgegangen, dass nur die wirklich notwendigen HTTP-Anfragen durchgeführt werden müssen.

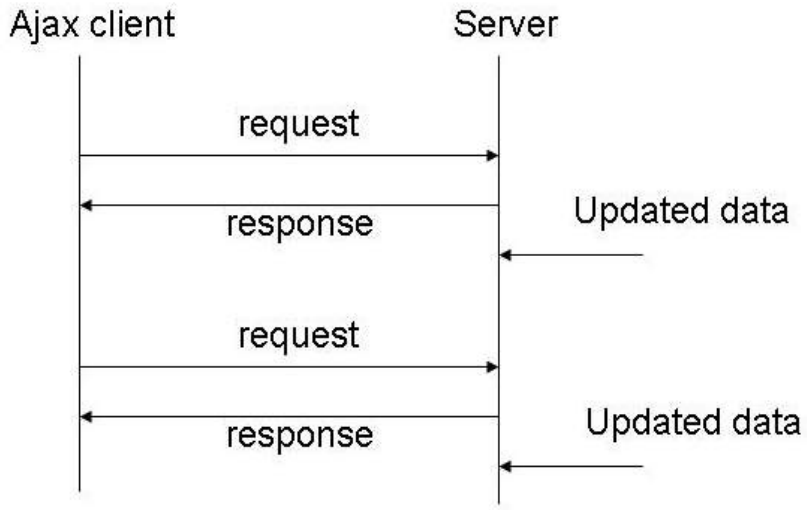

Abbildung 6: HTTP Polling

Das Polling-Verfahren stellt keine richtige Push-Technlogie dar. Eine Webseite wird aus Sicht des Benutzers zwar aktualisiert, rein technisch gesehen handelt es sich aber um eine Simulation der Push-Technlogie [EBG07].

4.2 Comet (Reverse AJAX)

Unter Comet (auch bekannt als Reverse AJAX, AJAX-Push) versteht man eine Technik, die es dem Server erlaubt, zeitnah aktuelle Informationen an den Client zu senden, wenn diese vorliegen [CRA08]. Bei der Comet-Technik unterscheidet man zwischen zwei Verfahren - Long Polling und Streaming. Technisch gesehen blockiert Comet einen *Thread* des Servers bis er aufgeweckt wird, weil neue Daten verfügbar sind oder eine Zeitspanne abgelaufen ist. Manche Authoren bezeichnen Comet als eine Herangehensweise, um durch kleine *Hacks* dem HTTP-Protokoll eine Zwei-Wege-Kommunikation zu ermöglichen [QRY09]. Es gibt auch Authoren, die Comet und Server-Push gleichstellen [SEURL]. In der vorliegenden Arbeit unterscheiden wir jedoch zwischen diesen Begriffen und fassen Server-Push als einen Oberbegriff auf.

4.2.1 HTTP Long-Polling

Bei Long-Polling wird ein Request zum Server geschickt und für einen bestimmten Zeitraum offen gehalten. Der Server beendet den offen gehaltenen Request, sobald neue Daten vorliegen. Wenn keine neuen Daten ermittelt werden konnten, wird der Request ebenfalls nach einem konfigurierbaren *Timeout*-Intervall geschlossen. Die ankommende Response wird im Browser verarbeitet. Anschließend wird ein neuer Request von einer AJAX-Engine gesendet, der auf weitere Daten wartet. Die nachstehende Abbildung 7 veranschaulicht, wie der Client über serverseitige Ereignisse benachrichtigt wird und daraufhin gleich wieder die Verbindung herstellt [NEURL].

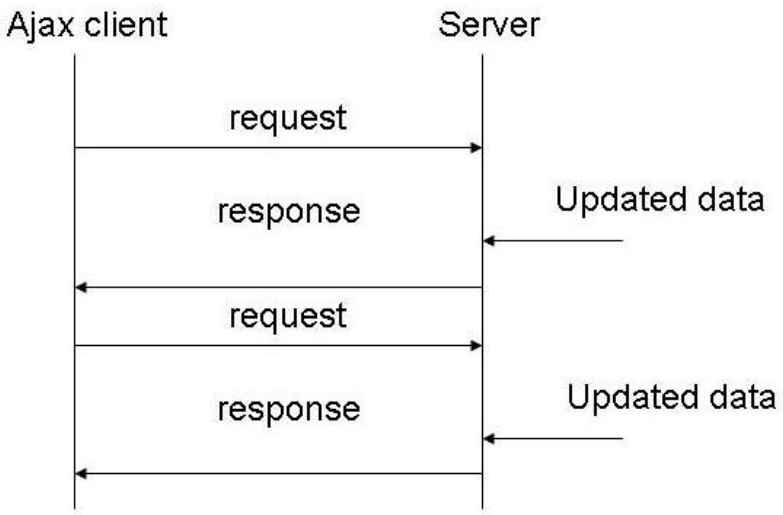

Abbildung 7: HTTP Long-Polling

4.2.2 HTTP Streaming

HTTP Streaming ähnelt dem Long-Polling mit einer Ausnahme, dass die Verbindung nie geschlossen wird. Der Client schickt einen einzigen Request, der niemals vom Server beendet wird. Sobald neue Daten auf dem Server verfügbar sind, werden diese nach und nach in die Response geschrieben (*HTTP chunked transfer*). Die offene Verbindung wird bei dieser Technik wiederverwendet, wie die nächste Grafik zeigt [NEURL].

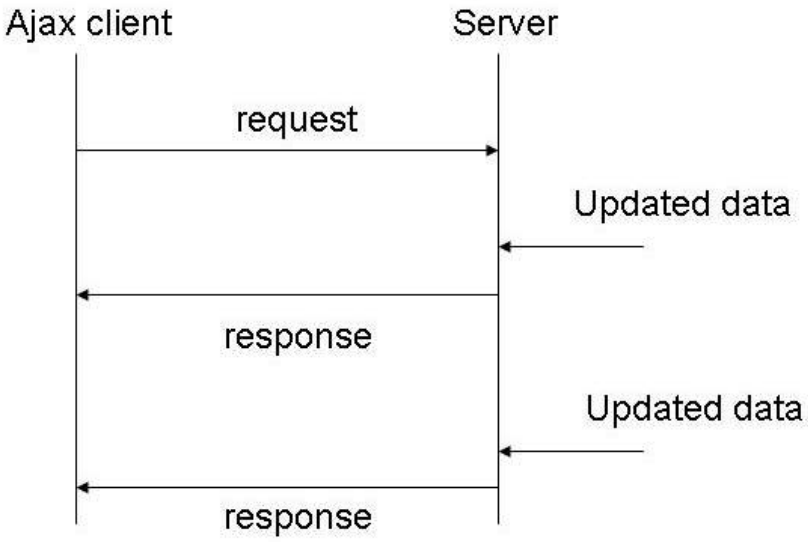

Abbildung 8: HTTP Streaming

Technisch gesehen wird beim HTTP Streaming der Status, der sogenannte „*readystate*“ des `XMLHttpRequest` Objektes (siehe das Kapitel 3.1), ausgelesen. Man macht sich zu Nutze, dass der Status auslesbar ist und ab „*readystate*“ 3 („*Loading*“) bereits Daten zur

Verfügung stehen. Sobald sich die HTTP-Verbindung in diesem Status befindet, werden eingetroffene Daten regelmäßig abgefragt und neue Ereignisse verarbeitet.

4.2.3 Bayeux-Protokoll

Bayeux ist ein Protokoll mit geringer Latenzzeit für den Transport von asynchronen Nachrichten zwischen einem Web-Server und einem Web-Client. Das Protokoll ist primär für den Datenaustausch via HTTP ausgelegt. Die Nachrichten werden über benannte Kanäle (*named channels*) zugestellt [BEURL]. Das Hauptziel von Bayeux ist die bidirektionale Interaktion zwischen Web-Clients und dem Web-Server. Das Protokoll setzt auf Comet auf und profitiert von den oben genannten Verbindungstypen: Long-Polling, Streaming, Callback-Polling. Das Callback-Polling hat exakt dasselbe Prinzip wie das Long-Polling, mit dem Unterschied, dass die Anfragen und Antworten von Client und Server nicht zwingend von derselben Domain ausgehen müssen. Das Callback-Polling ist also eine Cross-Domain Form eines regulären Long-Pollings.

Die Zustellung von Nachrichten erfolgt in Bayeux über benannte Kanäle in folgende drei Richtungen

- von Server zu Client
- von Client zu Server
- von Client zu Client (über einen Server)

Bayeux-Nachrichten sind im JSON-Format [JNURL] spezifiziert. Das nachfolgende Beispiel zeigt eine typische Nachricht

```
{
    "channel": "/some/name",
    "clientId": "83js73jsh29sjd92",
    "data": { "myapp" : "specific data", value: 100 }
}
```

Die asynchrone Kommunikation ist ereignisgesteuert und folgt dem Publish/Subscribe-Paradigma. Das Publish/Subscribe-Paradigma führt zu einer weitgehenden Entkopplung von Ereignisproduzenten und Ereigniskonsumenten. Der Client meldet sich beim Server an, indem er einen benannten Kanal im JSON-Objekt angibt. Die Namen der Kanäle sind in der Form einer absoluten Pfadangabe einer URI. In dem oben genannten Beispiel heißt der Kanal `"/some/name"`. Es können ebenfalls Wildcards benutzt werden, wie z.B. `/news/*`, wobei * eine beliebige Zeichenkette sein kann. Somit schließt `/news/*` z.B. `/news/italy` und `/news/italy/lazio` ein. Der Server kann anschließend JSON-Daten auf den benannten Kanälen zurückschicken.

Der Datenaustausch beginnt mit einem Handshake-Verfahren. Der Client und der Server tauschen sich die Informationen über den Verbindungstyp, die Authentifizierung usw. aus. Danach kann der Client sowohl bestimmte Kanäle abonnieren als auch seine Daten herausgeben, indem er Subscribe- bzw. Publish-Nachrichten verschickt. Wenn er die Daten herausgibt, bekommen sie interessierende Abonnenten (*Subscribers*) automatisch zugestellt. Die Daten können ebenso durch serverseitige Ereignisse an alle Kanal-Abonnenten gesendet werden.

Abbildung 9: Bayeux Nachrichten

Die mit Abstand bekannteste Implementierung der Bayeux-Spezifikation ist CometD [CDURL] - ein Projekt bei der *Dojo Foundation*.

4.3 Piggyback

Piggyback ist eine intelligente Art der Umsetzung von Server-Push. Diese Technik aktualisiert die Webseite nicht sofort, wenn der Server über neue *Updates* verfügt. Stattdessen wird solange gewartet, bis der Client eine neue Anfrage schickt. Das kann z.B. dann passieren, wenn der Benutzer neue Eingaben macht und ein Formular abschickt oder auf einen Link klickt. Wenn der Server nun auf diese Anfrage antwortet, fügt er auch die *Updates* der Antwort hinzu. D.h. Daten, die er vorher zwischengespeichert hat, werden mit der aktuellen Response transferiert.

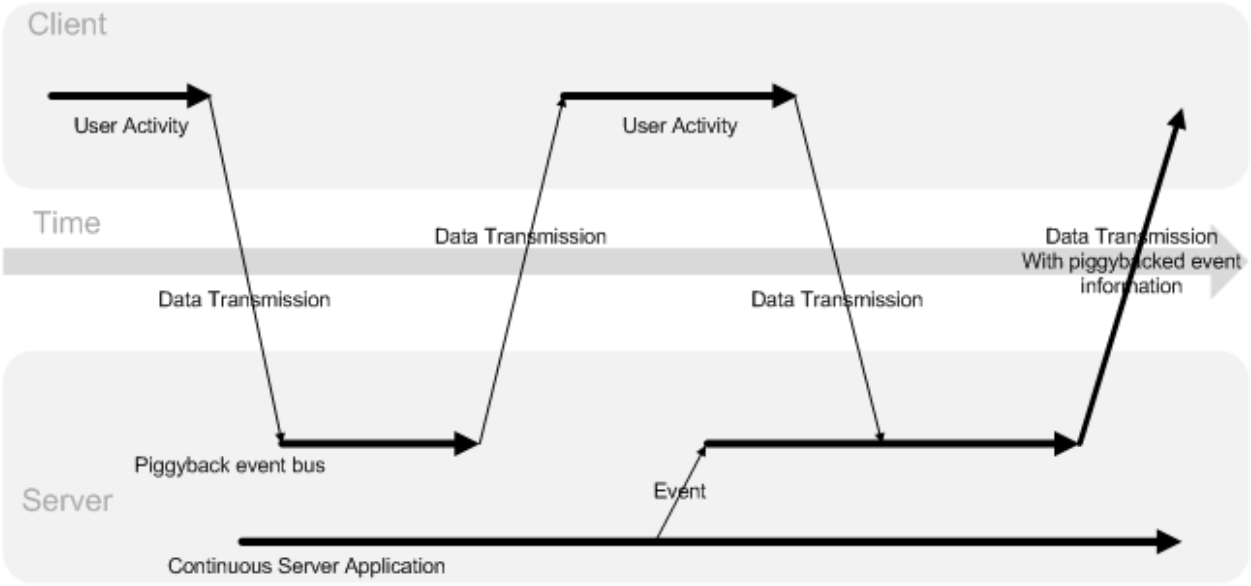

Abbildung 10: Piggyback-Technik

Wie der Name schon sagt, handelt es sich hier um ein Huckepack-Tragen - das Tragen einer Nachricht „auf dem Rücken“ einer anderen. Anstatt einer direkten und sofortigen Antwort über die neuen *Updates*, werden die anstehenden *Updates* als Zusatzinformationen mit der nächstmöglichen Response zurückgeschickt. In dieser Hinsicht handelt es sich um ein „passives“ Server-Push-Verfahren. Die obenstehende Abbildung 10 soll die Piggyback-Technik verdeutlichen [EXURL].

4.4 OpenAjax-Push-Protokoll

Das OpenAjax Protokoll für die Server-Push-Technologie ist im Rahmen der OpenAjax Initiative entstanden [OAURL]. Die OpenAjax Alliance, das Konsortium renommierter IT Unternehmen (*Adobe*, *Google*, *IBM*, *Microsoft* u.a.), hat sich das Ziel gesetzt, die Verwendung von AJAX zu standardisieren, um AJAX-Anwendungen robust, austauschbar und zukunftssicher zu machen. Die OpenAjax Alliance hat die Spezifikation und eine Open-Source-Implementierung von OpenAjax Hub 2.0 veröffentlicht. Hub 2.0 spezifiziert Javascript-APIs mit dem Ziel, sichere Mashups (Erstellung neuer Medieninhalte durch die nahtlose Kombination bereits bestehender Inhalte) über verschiedene Hersteller hinweg zu ermöglichen. So sollen Widgets (grafische Komponenten) und Mashups besser vor Angriffen geschützt werden. Dazu isoliert Hub 2.0 Widgets von Dritten in sicheren Sandboxen und regelt die Kommunikation verschiedener Widgets untereinander.

Das OpenAjax-Push-Protokoll [OPURL] verfolgt einen anderen Ansatz als Bayeux für die bidirektionale Kommunikation. Der gesamte Datenverkehr vom Server an den Browser wird bei den Comet-Techniken (HTTP Long-Polling / Streaming) durch die Nutzung blockierter HTTP-Verbindungen bewerkstelligt. Eine blockierte HTTP-Verbindung lässt leider keinen weiteren Datenverkehr zu. Sie bleibt in dem sogenannten „client read state“. Die meisten Browser können maximal 2 bis 8 Verbindungen gleichzeitig zu einer Domain aufbauen. So können freie Browser-Verbindungen in einer AJAX-Push / Comet Applikation sehr schnell aufgebraucht werden. Das ist insbesondere dann der Fall, wenn mehrere Web-Applikationen in verschiedenen Browser-Tabs geöffnet sind. Das OpenAjax-Push-Protokoll beschreibt ein Verfahren für die gemeinsame Nutzung einer blockierten HTTP-Verbindung durch alle Clients aus derselben Domain in einem einzigen Browser-Fenster. Dabei wird eine einzige Verbindung unter vielen Clients für reine Benachrichtigungen (*notifications*) geteilt („*geshared*“). Die eigentlichen Daten werden über andere Verbindungen / Kanäle geliefert, und zwar durch ganz normale, reguläre AJAX-Requests. Dafür ist die jeweilige Web-Applikation selbst zuständig. Das OpenAjax-Push geht somit von der Annahme aus, dass AJAX-Applikationen für den Datentransport bereits gut geeignet sind und erforderliche Push-Funktionen via Benachrichtigen neben den existierenden AJAX-Techniken beigefügt werden können.

Eine detaillierte Beschreibung der soeben beschriebenen Technologie wird nachfolgend aufgeführt. Ein GET- oder POST-Request zu der URL

```
http://<host>:<port>/openajax-push
```

wird blockiert und liefert eine “&”-separierte Liste von Client-IDs (Identifikatoren), für die neue Inhalte (*Updates*) verfügbar sind. Einzelne Clients innerhalb des Browsers rufen dann

die neuen Inhalte in ihrer gewöhnlichen Art und Weise ab (hängt von dem verwendeten Web-Framework ab). Als Beispiel auf diese Anfrage

```
POST /openajax-push HTTP/1.1
Accept: */*
Cookie: BROWSERID=D8921099ED29BA291A4E1D02C9118CBA
Content-Type: application/x-www-form-urlencoded;charset=UTF-8
Content-Length: 0
Connection: keep-alive
Host: localhost:8080
```

wird eine Benachrichtigung gesendet, dass die Inhalte für Clients mit IDs “1”, “3”, “5” und “7” vorhanden sind.

```
HTTP/1.1 200 OK
Content-Type: application/x-www-form-urlencoded;charset=UTF-8
Content-Length: 7
Date: Thu, 19 Apr 2007 16:09:53 GMT
Server: Apache-Coyote/1.1

1&3&5&7
```

Daraufhin werden im Browser dazugehörige Rückruffunktionen (*callbacks*) für die Clients “1”, “3”, “5” und “7” aufgerufen, so dass diese Clients ihre Daten / Nachrichten über eigene nicht blockierte Mechanismen holen können. Das OpenAjax-Push-Protokoll beschreibt somit nur noch das Format der Benachrichtigung über das Vorliegen neuer Daten / Nachrichten auf dem Server. Zu bemerken ist noch, dass die Rückruffunktionen von einem Browser-Fenster zu dem anderen (für eine und dieselbe Browser-Instanz) propagiert werden müssen. Diese Art der Kommunikation zwischen Browser-Fenstern kann durch Cookies realisiert werden, weil sie Browser-Fenster übergreifend sind.

4.5 Ajax-on-Demand

Der Begriff Ajax-on-Demand wurde von Alexander und Sergey Smirnov (Exadel Inc.) eingeführt [DEURL]. Ajax-on-Demand stellt eine interessante Alternative zu Polling, Comet und OpenAjax dar und behebt deren wesentliche Nachteile, die im Kapitel 5 beleuchtet werden. Diese Technik ähnelt dem HTTP Polling, verbessert es jedoch wesentlich in Punkten der Ressourcenoptimierung während des Verbindungsaufbaus und der Latenzzeit. Ajax-on-Demand ist sozusagen eine „leichtgewichtige“ Variante des Polling-Verfahrens. Die Idee kam von einem Unix-Hilfsprogramm namens „biff“. Das kleine Programm prüft im Hintergrund, ob neue E-Mails eingetroffen sind und benachrichtigt alle Empfänger, die dann Ihre E-Mails ganz normal abholen können. Eine Analogie gibt es auch in der Java-EE-Welt. Durch Clients gesendete Nachrichten können in eine spezielle Nachrichten-warteschlange (*message queue*) gestellt werden und werden dort durch sogenannte *Message Driven Beans* überwacht und abgeholt.

Abgesehen von den technischen Details, wird in der webbasierten Client-Server-Umgebung eine kleine, ressourcensparende Prüfung auf neue Daten (*Updates*) periodisch ausgeführt.

Die Prüfung wird durch den Client (AJAX-Engine in Browser) initiiert. Liegt ein *Update* auf dem Server vor, wird er vom Client auf einem regulären und zuverlässigen Weg geholt. Die Implementierung sollte sich nach einem Zwei-Phasen-Modell richten. In der ersten Phase schickt der Client kurz laufende Requests zu einem speziellen Servlet, welches Informationen über verfügbare Nachrichten-Identifikatoren sammelt. Solche HTTP Requests können z.B. HEAD-Requests sein. Im Vergleich zu POST- oder GET-Requests verlangen HEAD-Requests keine Autorisierung, Wiederherstellung der Benutzer-Sessions oder andere ressourcenintensive Operationen. Das Ziel der kurz laufenden Requests ist die Verifizierung auf neu angekommene *Updates*. In der zweiten Phase kommt ein normaler, regulärer AJAX-Request an die Reihe und aktualisiert die Webseite entsprechend den neuen Daten. Die Abbildung 11 zeigt den Ablauf schematisch [DEURL].

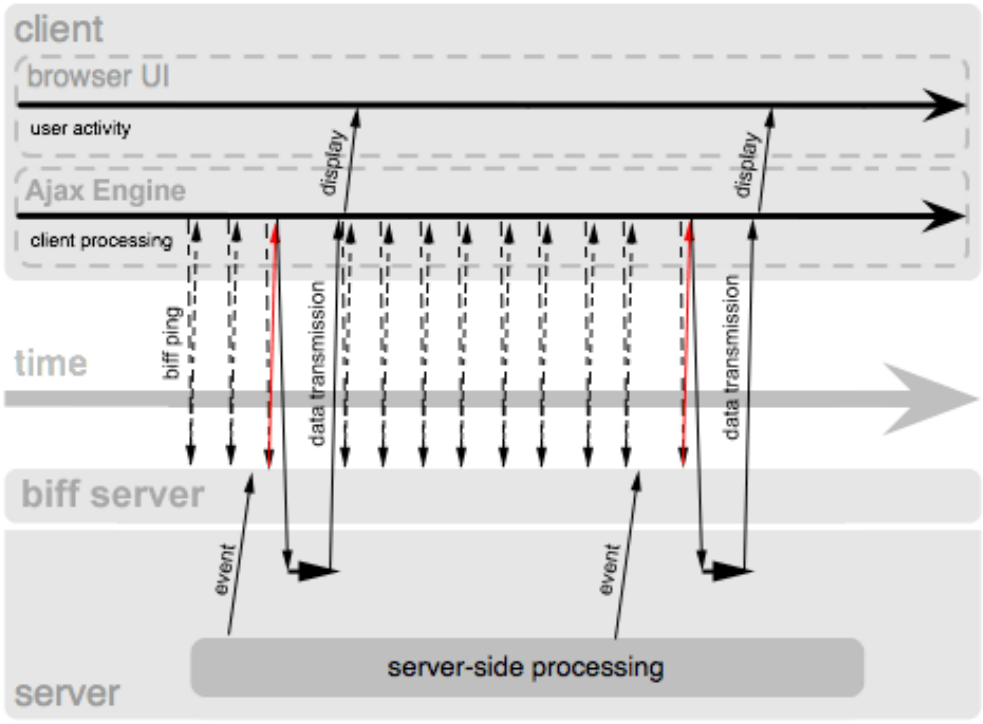

Abbildung 11: Ajax-on-Demand

4.6 Erweiterbare Protokolle zur Nachrichtenübermittlung

„*Bidirectional-streams Over Synchronous HTTP*“ oder kurz BOSH ist ein Transport Protokoll, das eine bidirektionale Kommunikation (*Streams*) in der Client-Server-Umgebung emuliert. Das Protokoll macht Gebrauch von der Long-Polling-Technik. BOSH wurde im Rahmen des XMPP-Protokolls entworfen [BIURL]. In diesem Unterkapitel werden die Grundlagen der beiden Protokolle sowie die darauf basierende neue Channel-Schnittstelle von Google erläutert.

4.6.1 Extensible Messaging and Presence Protocol (XMPP)

XMPP steht für „*Extensible Messaging and Presence Protocol*“ und erlaubt eine Client-

Client (über einen XMPP-Server) oder Server-Server Kommunikation. Obwohl das Protokoll primär auf „Instant Messaging“ ausgelegt ist, ist es vielseitig einsetzbar. Mit XMPP kann unter anderem Chatten (bekannteste Chat-Anwendungen sind IRC und ICQ), Telefonieren, Dateien austauschen, gemeinsam Dokumente bearbeiten, RSS-, Atom-Feeds transportieren. Für den Betrieb eines XMPP-Netzwerks ist mindestens ein XMPP-Server nötig. Über das Internet kann dieser Server zu anderen XMPP-Servern Verbindungen herstellen. Die Netz-Architektur erinnert dabei an SMTP (*Simple Mail Transfer Protocol*). Für den Nachrichtenaustausch von Client zu Client müssen sich beide Clients auf einem XMPP-Server einloggen. Der Benutzername, auch Jabber ID genannt, hat das Format einer E-Mail Adresse. Ein Beispiel - „alice@jabber.example.com“, wobei „alice“ der Benutzername und „jabber.example.com“ der XMPP-Server ist. In der nachfolgenden Abbildung [WEURL] sendet Alice eine Nachricht an Bob. Die Nachricht wird erst zu dem XMPP-Server „jabber.example.com“ gesendet, an dem Alice angemeldet ist. Alice's XMPP-Server schickt die Nachricht weiter an Bob's Server, indem das XMPP-Protokoll verwendet wird. Der Server von Bob verwendet anschließend das XMPP kompatible Protokoll OSCAR, um die Nachricht an Bob's ICQ Client weiterzuleiten.

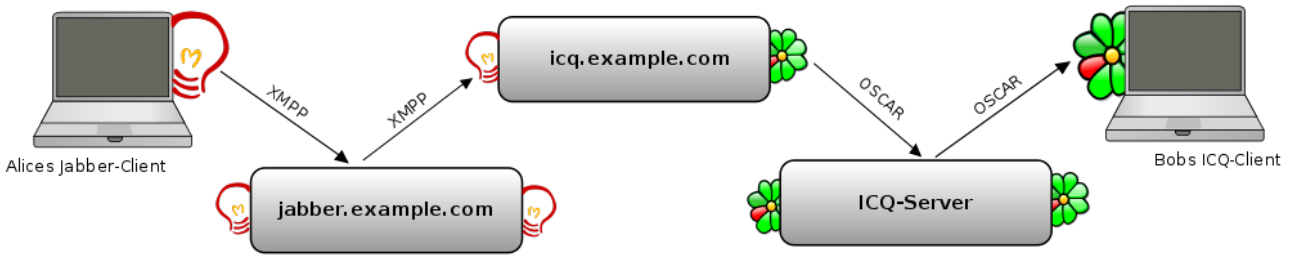

Abbildung 12: XMPP-Transport

Auf XMPP basieren eine Reihe bekannter Dienste, wie z.B. Google Talk / Wave, Facebook Chat, Apple iChat Server, Oracle Beehive Collaboration, LiveJournal, Microblogging-Dienst Twitter.

4.6.2 Bidirectional-streams Over Synchronous HTTP (BOSH)

BOSH ermöglicht XMPP-Pakete über das HTTP zu schicken. Das bedeutet, dass XMPP direkt über den Browser benutzt werden kann. Mit BOSH lassen sich also XMPP-Applikationen für den Browser realisieren. Neben Client und Server gibt es in der BOSH-Architektur einen sog. „Connection Manager“, der eine serverseitige Implementierung des BOSH-Protokolls darstellt und von dem eigentlichen Server logisch getrennt ist. Der „Connection Manager“ übersetzt transparent die Kommunikation zwischen dem HTTP-Protokoll des Clients und dem XMPP-Protokoll, welches der Server verwendet. Der „Connection Manager“ fungiert damit als eine Art "Übersetzungsproxy bzw. Übersetzungsdienst" zwischen BOSH HTTP-Clients und dem XMPP-Server. Der Client sendet per AJAX eine Anfrage an den „Connection Manager“, der solange nicht antwortet, bis ihm neue Ereignisse vorliegen. Beim Auftreten neuer Ereignisse überträgt der „Connection Manager“ diese sofort über den bestehenden, offenen Request und beendet die Verbindung. Stellt der Client eine neue Anfrage an den Server während parallel dazu eine Verbindung bereits offen ist, wird eine zweite Verbindung aufgebaut. Der Server antwortet

allerdings auf die Anfrage des Clients über die erste Verbindung, auch wenn es keine Daten zu senden gibt. Daraufhin wird die erste Verbindung geschlossen. Die jetzt noch offene zweite Verbindung steht dem Server ab sofort zur Verfügung, falls er aktuelle Daten aufgrund neuer Ereignisse an den Client schicken muss. Das heißt, dass die Rollen der zwei Verbindungen mit jeder Anfrage des Clients wechseln. Eine solch geschickte Handhabung von zwei Verbindungen bringt Vorteile gegenüber dem Standard Long-Polling. Diese werden im Kapitel 4 erläutert.

4.6.3 Channel API für Google App Engine

Google hat seine eigene Push-Technik erfunden, die für die GAE (*Google App Engine*) konzipiert wurde – Channel API [GAURL]. Google App Engine ist eine Plattform zum Entwickeln und Hosten von Webanwendungen auf den Servern von Google. Das Channel API ermöglicht dauerhafte bidirektionale Verbindungen zwischen den im Browser und den auf dem Server laufenden Teilen / Artefakten einer Anwendung. Die Channel-Schnittstelle ähnelt zwar dem HTML5 WebSocket [Kapitel 4.9.2], indem sie die Ereignisse (*events*) wie `open`, `message`, `error` und `close` definiert, setzt aber komplett auf das XMPP-Protokoll. Der Hauptunterschied der Google's Push-Technik zum WebSocket-Standard besteht auch in der Nutzung von sogenannten, auf dem Server erzeugten *Channels* anstatt einer URL. Der Client benutzt ein unsichtbares `iframe`, um mit der Google App Engine kommunizieren zu können und verarbeitet die oben erwähnten Ereignisse durch bereitgestellte JavaScript Handler-Funktionen (noch *Listeners* oder *Callbacks* genannt).

Der Client beginnt die Kommunikation und fordert ein „Token“ für sich. Der Server generiert das „Token“, schickt es an den Client und erzeugt einen Kommunikationskanal (*Channel*) für ihn. Der Client kann nun den Kanal permanent nutzen, um über neue Ereignisse benachrichtigt zu werden. Angenommen, ein weiterer Client sendet nun einen POST-Request an den Server. Die übermittelten Daten des POST-Requests sind für den ersten Client bestimmt. Der Server verarbeitet die Nachricht und leitet sie über den zuvor eingerichteten Kanal an den ersten Client weiter. Bei dem ersten Client wird der `onmessage` Handler ausgeführt, der die Nachricht des zweiten Clients in dem übergebenen Parameter bekommt.

Der nachfolgende JavaScript Code zeigt das Warten auf neue Nachrichten mit dem `onmessage` Handler, nachdem das „Token“ auf der Client-Seite bekannt ist.

```
<script src='/_ah/channel/jsapi'></script>

<script>
  var channel = new goog.appengine.Channel(<token>);
  var socket = channel.open();
  socket.onmessage = function(evt) {
    evt.data  // do something with the data
  };
</script>
```

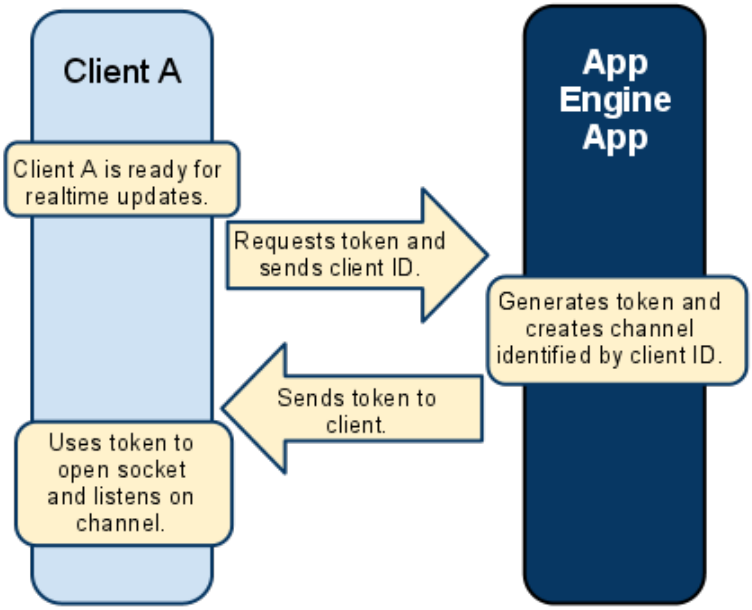

Abbildung 13: Google's API - Channel erzeugen

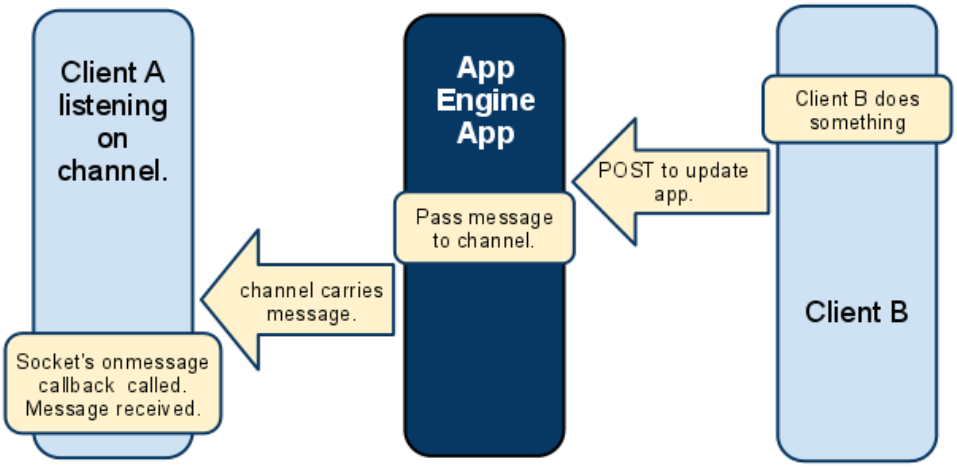

Abbildung 14: Google's API - Nachrichten senden

4.7 Reverse HTTP

Reverse HTTP ist ein experimentelles Protokoll, vorgeschlagen von Mark Lentczner und Donovan Preston im März 2009, dass Vorteile von dem sogenannten HTTP/1.1 Upgrade-Header nutzt [REURL]. Wenn die Anfrage einen Upgrade-Header enthält, der für den Server akzeptabel ist, wird hiermit das Ändern des Protokolls bestätigt und die Richtung der Datenübertragung kann sich ändern. Die Einweg-Kommunikation vom Client zum Server lässt sich damit reversieren. Wenn der Client einen Request mit dem *Upgrade-Header: PTTH/0.9* an den Server stellt, kann der Server mit dem *Upgrade-Header: PTTH/1.0* antworten. Danach tauschen sie die Rollen, d.h. der Server beginnt das HTTP-Socket als Client zu benutzen und der Client benutzt das HTTP-Socket als Server. Im Einzelnen kann

die Ablaufreihenfolge folgendermaßen aussehen.

1. Client → Server Request

```
POST / HTTP/1.1
Host: localhost:9999
Accept-Encoding: identity
Upgrade: PTTH/0.9
```

2. Server → Client Response

```
HTTP/1.1 101 Switching Protocols
Content-type: text/plain
Upgrade: PTTH/0.9
Date: Tue, 13 May 2008 20:14:45 GMT
Content-Length: 0
```

3. Server → Client Request

```
GET / HTTP/1.1
Host: 127.0.0.1:65331
Accept-Encoding: identity
accept: text/plain;q=1,*/*;q=0
```

4. Client → Server Response

```
HTTP/1.1 200 OK
Content-type: text/plain
Date: Tue, 13 May 2008 20:14:45 GMT
Content-Length: 15
!dlrow ,olleH
```

Man sieht ab Punkt 3, dass der Server als Client und der Client als Server fungieren. Falls der Client oder der Server unfähig sind, diese Art der HTTP-Kommunikation zu unterstützten, kann ein Fallback-Protokoll in dem Poll- / Comet-Stil benutzt werden. Der Server kann in dem Fall eine Nachrichtenwarteschlange führen (*event queue*), die durch den Client periodisch abgefragt wird. Nachfolgend ist ein Beispiel aufgeführt, das auf dem JSON-Format basiert.

1. Client → Server Request

```
POST / HTTP/1.1
Host: localhost:9999
Accept-Encoding: identity
```

2. Server → Client Response

```
HTTP/1.1 200 OK
Content-type: application/json
Date: Tue, 13 May 2008 20:14:45 GMT
Content-Length: 210
{'message-id': 'xxx', 'method': 'GET', 'request-uri':
```

```
'/', 'http-version': 'PTTH/0.9', 'headers': [['Host',
'127.0.0.1:65331'], ['Accept-Encoding', 'identity'],
['Accept','text/plain;q=1,*/*;q=0']], 'body': }
```

3. Client → Server Request

```
POST / HTTP/1.1
Host: localhost:9999
Accept-Encoding: identity
Content-type: application/json
Content-length: 193
{'message-id': 'xxx', 'http-version': 'PTTH/0.9',
'status': 200, 'reason': 'OK', 'headers': [['Content-
type', 'text/plain'], ['Date', 'Tue, 13 May 2008 20:14:45
GMT']], 'body': '!dlrow ,olleH'}
```

4. Server → Client Response

```
HTTP/1.1 200 OK
Content-type: application/json
Date: Tue, 13 May 2008 20:14:45 GMT
Content-Length: 2
{}
```

Die Response `{}` bedeutet, dass keine serverseitigen Ereignisse innerhalb einer gewissen Zeit aufgetreten sind und der Client erneut eine Poll-Anfrage stellen soll.

4.8 Neue HTML5 Kommunikationsstandards

Die Entstehung von HTML5 beginnt im Juni 2004, initiiert vom W3C. *Mozilla* und *Opera* forderten eine Weiterentwicklung von HTML, DOM und CSS beim W3C als Basis für Web-Applikationen der Zukunft. Im Jahr 2010 wurde HTML5 zu dem Schlagwort (Buzzword) in der Webentwickler-Szene. Große Firmen wie *Google*, *Apple* und *Microsoft* eigneten sich die neue Technologie an. Dazu trug auch die heftig geführte Debatte zwischen *Apple* und *Adobe* bei, ob HTML5 das Ende von Flash bedeuten würde oder nicht. Neben den klassischen Bestandteilen der Spezifikation wie Video, Audio, Canvas, intelligenten Formularen, Offline-Applikationen und Microdata, existieren auch Themen im unmittelbaren Umfeld von HTML5 – Geolocation, Web Storage, Web Workers, WebSockets und Server-Sent Events [OGG10]. Mit WebSockets und Server-Sent Events befinden sich Technologien in der Entstehung, welche es auch einem Server ermöglichen, Daten aktiv an einen Browser zu senden. Es handelt sich dabei um eine native Unterstützung der Server-Push-Technologie durch die spezifizierten Web-Standards.

4.8.1 Server-Sent Events

Konventionelle Web-Applikationen generieren clientseitige Ereignisse (*events*), z.B. führt ein Klick auf einen Link im Browser zum Laden einer neuen Webseite. Solche Ereignistypen, die sich vom Browser zum Server ausbreiten, nennt man gelegentlich Client-

Sent Events. Server-Sent Events dagegen erlauben dem Server einseitig und ohne Rückfrage beim Client Daten zu versenden und Funktionen auszuführen. Bei Server-Sent Events braucht man keinerlei offener *Sockets* (bidirektionale Software-Schnittstelle zur Netzwerk-Kommunikation) oder ständiger Anfrage bei einem anderen Server. Aus diesem Grund ist der neue HTML5-Standard gerade auf mobilen Geräten sinnvoll, um Energie zu sparen. Auch Browser-Ressourcen werden hier besser benutzt als mit `XMLHttpRequest` oder `iframe`. Obwohl eine endgültige Spezifikation des W3C Konsortiums für die Server-Sent Events noch nicht verabschiedet ist [SSURL], werden sie bereits in der ersten Vorabversion von Firefox 6 unterstützt. Damit lassen sich schon jetzt DOM-Ereignisse vom Server aus auslösen und mit Daten versehen.

Die neue JavaScript Programmierschnittstelle `EventSource` bietet im Rahmen des HTML5-Standarts die Fähigkeit, ereignisbasierte Push-Nachrichten von einem Server zu empfangen. Das `EventSource`-Objekt erwartet einen Parameter mit der URL zu dem Host, mit dem eine persistente Verbindung aufgebaut werden soll. Über diese Verbindung werden dann Daten an den Client geliefert. Man kann eine Methode `onmessage` auf dem `EventSource`-Objekt registrieren, die für den Datenempfang zuständig ist.

```
<script type='text/javascript'>
  var src = new EventSource('http://example.com/server.php');
  src.onmessage = function (event) {
    event.data // do something with the data
  };
</script>
```

Jede serverseitig generierte Nachricht enthält den Mime-Type `text/event-stream`, und besteht aus einer Liste von Nachrichtendaten.

```
Content-Type: text/event-stream
data: This is the first message.

Content-Type: text/event-stream
data: This is the second message, it
data: has two lines.

Content-Type: text/event-stream
data: This is the third message.
```

4.8.2 WebSocket - Vollduplex-Kommunikation

Für die Entwicklung des WebSocket-Protokolls gab es einige Gründe [OGG10]. Eine HTML-Seite, die ein Stylesheet und fünf Bilder referenziert, benötigt zum Laden sieben Verbindungen. Das bedeutet, siebenmal findet ein Verbindungsaufbau statt, um Meta-Daten und Nutzdaten zu übertragen. In der HTTP Version 1.1 wurde dieses Verhalten zwar durch die Keep-Alive-Funktion insoweit verbessert, als neue TCP-Verbindungen nicht jedes Mal aufgebaut werden müssen, die Meta-Informationen werden aber immer noch für jedes

Objekt extra übertragen. Das andere Problem besteht in der Verfolgung einer Benutzersitzung, weil HTTP diese Funktion nicht hat. Abhilfe schaffen hier Hilfsmittel wie Session und Cookies. Das WebSocket-Protokoll, welches eine Ergänzung zu HTTP darstellt und das zugrundeliegende TCP/IP-Netzwerkprotokoll nutzt, stellt eine wesentliche Verbesserung dar. Das neue Protokoll transportiert Daten ohne Meta-Informationen in einem konstanten Strom, und zwar gleichzeitig vom Server zum Client und umgekehrt (Vollduplex). Die bidirektionale Kommunikation zwischen Server und Client läuft über genau einen Kommunikationskanal ab. Sobald die WebSocket-Verbindung aufgebaut ist, können Server und Client gleichzeitig miteinander kommunizieren.

Das WebSocket-Protokoll kann auf der Webseite des W3C Konsortiums als Editor's Draft begutachtet werden [WAURL]. Eine Client-Server-Verbindung wird mittels Handshake-Verfahrens aufgebaut [WSURL]. Die Verbindung wird nach dem Austausch eines Handshakes im Gegensatz zu HTTP aufrechterhalten. Der Client sendet einen Header in der folgenden Form:

```
GET <path> HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Sec-WebSocket-Key: <random token>
Host: <hostname>:<port>
Origin: http://<host>[:<port>]
```

Der Server antwortet mit

```
HTTP/1.1 101 Web Socket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
WebSocket-Origin: http://<hostname>[:<port>]
WebSocket-Location: ws://<hostname>:<port>/
```

Der WebSocket-Client gibt an, dass er ein Upgrade auf das WebSocket-Protokoll durchführen möchte. Der `Sec-WebSocket-Key` Header stellt eine Base64-enkodierte Zeichenkette dar, die der Server nutzt, um den Verbindungsaufbau zu akzeptieren. Der WebSocket-Server antwortet mit dem HTTP-Statuscode 101 und signalisiert damit, dass er den Upgrade-Wunsch akzeptiert. Der Handshake dient in erster Linie den Sicherheitsaspekten, um z. B. Cross-Domain-Zugriffe zu kontrollieren. Nun können beide Parteien die Daten über einen einzigen bidirektionalen Socket-Kanal austauschen. Die Daten werden in Form sogenannter Datenframes ausgetauscht. Anschließend sollte der Socket-Kanal anhand des Handshake-Verfahrens geschlossen werden. Es gibt zwei Arten von WebSocket-Datenframes – Textframes and Binärframes. Freigegeben sind momentan nur noch Textframes. Jede textuelle Nachricht fängt mit einem `0x00`-Byte an und endet mit einem `0xFF`-Byte. Zwischen den zwei Bytes befinden sich die Nutzdaten, kodiert im UTF-8 Format.

```
0x00 <UTF-8 Textdaten> 0xFF
```

Jede binäre Nachricht hat eine andere Struktur:

```
0x80-0xFF <Länge> <Binäre Daten>
```

Die Abbildung 15 soll den Lebenszyklus einer WebSocket-Verbindung demonstrieren.

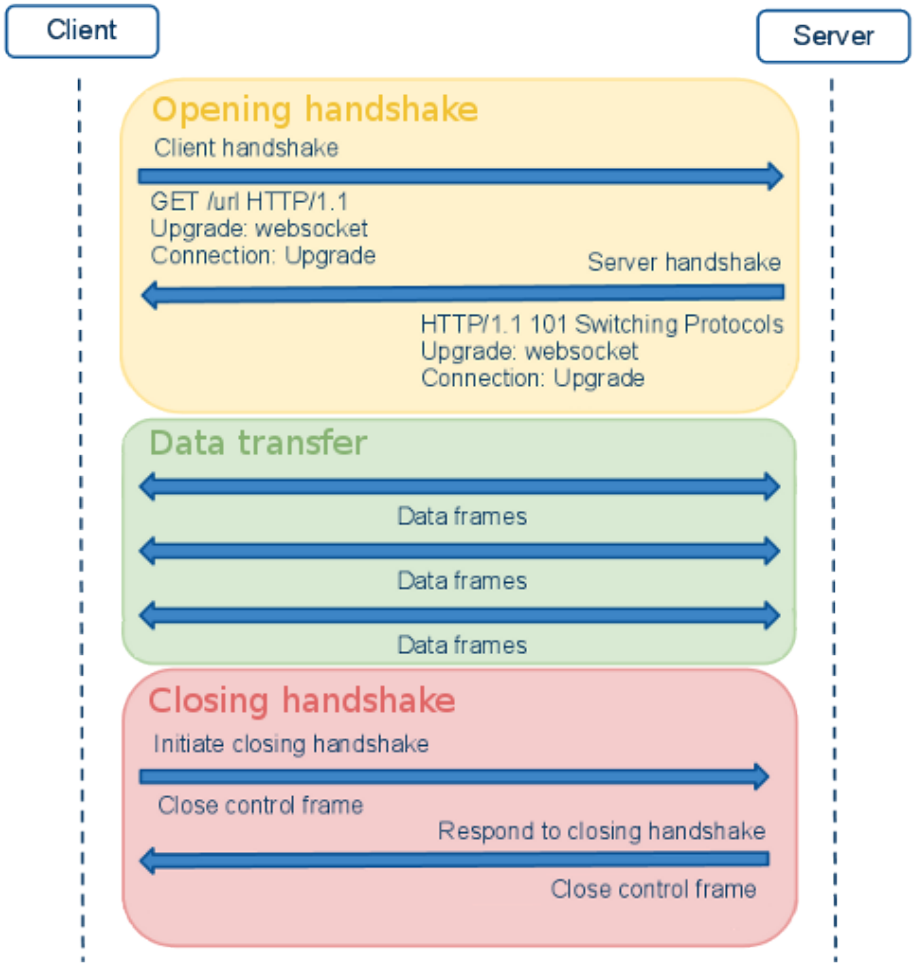

Abbildung 15: WebSocket-Verbindung

Um die Verbindung zu einem WebSocket-Server aufzubauen, wird ein JavaScript-Objekt erzeugt. Der Konstruktor nimmt die URL des Servers entgegen. Anschließend lassen sich verschiedene Handler-Funktionen definieren.

```
// connect to a web socket located at ws://127.0.0.1:8080/
var socket = new WebSocket("ws://127.0.0.1:8080/");

// define listeners
socket.onopen = function() {
  alert("Connection to the socket is now open");
}
socket.onmessage = function(e) {
  alert("Message received: " + e.data);
}
socket.onclose = function(e) {
```

```
  alert("Connection to the socket is closed");
}

// send message to the server
socket.send("Hallo Server!");

// close WebSocket
socket.close();
```

Der `onopen`-Listener wird nach der Herstellung der Verbindung aufgerufen, der `onclose`-Listener wird aufgerufen, sobald die Verbindung vom Client oder vom Server geschlossen wurde und der `onmessage`-Listener nimmt die ankommenden Daten entgegen. Mittels der `send()` Methode sendet der Client Nachrichten über den bidirektionalen Kommunikationskanal.

5 Bewertung bidirektionaler Kommunikationstechnologien

Die Bewertung der Technologien für die bidirektionale Kommunikation hilft einen besseren Überblick für die Auswahl eines geeigneten Server-Push-Frameworks zu verschaffen, das einer konkreten Client-Server-Landschaft am besten gerecht werden kann. Doch welche Vergleichskriterien müssen zur Bewertung herangezogen werden? Bei der Vielzahl von verschiedenen Server-Push-Technologien fällt es zugegebenermaßen schwer, gemeinsame Kriterien zu finden. Nicht alle Kriterien sind technologieübergreifend und können auf alle Techniken gleichermaßen angewendet werden. Sie sind aber trotzdem nennenswert und werden bei der Bewertung berücksichtigt.

Im Rahmen der Bewertung der Kommunikationstechnologien werden zuerst die Grenzen von AJAX diskutiert, die inhaltsmäßig in dieses Kapitel gehören. Danach wird zur Erarbeitung von Vergleichskriterien und zum eigentlichen Vergleich übergegangen. Eine Zusammenfassung mit der Übersichtstabelle einzelner Technologien und untersuchter Kriterien schließt die Bewertung ab.

5.1 Grenzen von AJAX

Wie wir bereits gesehen haben, hat AJAX viele Vorteile und findet eine breite Unterstützung durch aktuelle Browser ohne zusätzliche Plugins. Alle AJAX Bestandteile sind offen und standardisiert, die verwendeten Techniken sind praxiserprobt und gut dokumentiert. Wie jede andere Technologie, hat aber auch die AJAX-Technologie ihre Grenzen. Die nachfolgende Auflistung soll meist bekannte Schwachstellen dieser Technologie aufzeigen.

- Nachladen von Ressourcen mittels `img` Elementes stellt keine echte AJAX-Technik dar, weil damit nur kleine Datenmengen übertragen werden können und jegliche Flexibilität und Kontrolle fehlen. Kleine Datenmengen werden in Cookies übertragen, was die notwendige Überwachung des Browserzustandes in kurzen Abständen mit sich zieht.
- Beim Nachladen von Ressourcen mittels `script` Elementes ist Vorsicht geboten, weil dies die Kommunikation mit fremden Webseiten (*third-party sites*) ermöglicht und zu Sicherheitslücken führen kann.
- Beim Nachladen von Ressourcen mittels `iframe` Elementes kommt es zu Klickgeräuschen und Ladebalken wenn ein `iframe` seinen Inhalt neu lädt. Aufgrund der speziellen Kommunikation zwischen dem Hauptdokument und dem `iframe` ist die Handhabung zudem schwieriger und der ganze Ablauf ist fehleranfälliger.
- Mit der Manipilation des `src` Attributes ist kein POST-Request möglich. Ein GET-Request ist auf ca. 2 KB begrenzt. Alle Techniken, die auf der Manipilation des `src` Attributes basieren, können somit wegen der Obergrenze an übertragbaren Datenmengen zu Problemen führen. Außerdem werden Werte bei einem GET-Request der URL hinzugefügt. Das ist unschön, weil man alle Werte in der Adresszeile des Browsers sieht.
- Remote Scripting ohne `XMLHttpRequest` ergibt das Problem der koordinierten Verarbeitung der Antwort des Servers. Ohne `XMLHttpRequest` besteht keine

vollständige Kontrolle über die HTTP-Anfrage. Rückmeldungen über den ändernden Status einer laufenden HTTP-Anfrage können nicht beschafft werden. Andererseits wird das Remote Scripting ohne `XMLHttpRequest` besser von älteren Browsern unterstützt.

- Einige AJAX-Anwendungen verändern das Verhalten des Zurück-Buttons des Browsers. Wenn der Benutzer auf einen Hyperlink klickt und eine ganz neue Seite aufgebaut wird, wird ein neuer Eintrag in der Browser-History erzeugt. AJAX-Anwendungen erzeugen aber keine neuen Seiten. Stattdessen aktualisieren sie den Inhalt innerhalb einer existierenden Seite. Wenn jetzt der Benutzer den Zurück-Button anklickt, springt er zur vorher geladenen Seite zurück oder sogar aus der AJAX-Anwendung heraus. Nur bei der Verwendung von IFrames bewahrt der Zurück-Button seine normale Funktion.
- Häufige Kritikpunkte an AJAX sind die höhere Serverlast, die von den zahlreichen Mikro-Transaktionen (AJAX-Request, -Response) hervorgerufen wird. Ein weiterer Punkt sind diverse Sicherheitsprobleme durch die verstärkte Nutzung von JavaScript und ActiveX [BER05].
- AJAX hat auch Auswirkungen auf die Barrierefreiheit von Webseiten [BER05]. Visuelle Effekte basieren vorwiegend auf der Manipulation des DOM-Baums. Die transportierte Information lässt sich damit nicht besonders gut auf einem linearen Ausgabegerät wie z. B. Einem Screenreader oder einer Braillezeile wiedergeben. Der Nutzerkreis wird damit eingeschränkt.
- AJAX an sich garantiert keine Auslieferung von Daten ohne explizite Browser-Anfrage. Chat-Anwendungen, Online Spiele, Live-Daten vom Server sind damit nicht ohne weiteres möglich. Eine Echtzeit-Auslieferung von Daten vom Server zum Client nach dem Prinzip „don't call us, we call you“ ist nur auf Basis einer Server-Push-Architektur möglich.

5.2 Vergleichskriterien

Latenzzeit und Performance

Latenzzeit, auch Verzögerungszeit und Reaktionszeit genannt, ist der Zeitraum zwischen einer Aktion und dem Eintreten einer verzögerten Reaktion. Bei den meisten Server-Push-Technologien bleibt die Aktion verborgen und wird erst durch die Reaktion deutlich. Die Latenz ist eine häufige Problemquelle jeder AJAX-Anwendungen. Je nach angewendeter Technik besteht ein deutlich bemerkbares Latenzproblem oder nicht. Die Latenz kann beispielsweise durch ein länger eingestelltes Abfrageintervall höher ausfallen. Firewalls und Proxy-Server können die Latenzzeit bei manchen Verfahren ebenso erhöhen. Die Latenz und die Performance sind zwei verwandte Begriffe und fallen daher unter ein Vergleichskriterium. Mit der Performance wird in der Informatik das Zeitverhalten von Software bezeichnet. Performanceprobleme existieren in AJAX-Anwendungen genauso wie in „herkömmlichen“ Webanwendungen. Hinsichtlich des in diesem Kapitel beschriebenen Vorhabens, sind solche Technologien herauszuarbeiten, welche die Latenzzeit bei der Auslieferung von Nachrichten an den Client reduzieren und gleichzeitig die Performance der Anwendung optimieren.

Serverlast
Die Serverlast entsteht hauptsächlich durch die Erzeugung neuer *Threads* auf dem Server mit zunehmender Anzahl der HTTP-Requests. Der Server kann bei zu vielen HTTP-Requests einfach zu Grunde gehen, da verfügbare Ressourcen schnell verbraucht werden. Der zweite Fall, der zu einer höheren Serverlast führen kann, ist die zunehmende Anzahl von gleichzeitig offenen Verbindungen. Bei einigen Technologien ist dies gerade der Fall und somit muss die Server-Software, durch einen rapide ansteigenden Verwaltungsaufwand für offene Verbindungen, einer hohe Belastung standhalten. Zusammenfassend kann man sagen, dass je mehr die Technologie serverseitige Prozesse beansprucht, desto mehr ist die Serverbelastung. Man spricht auch von der CPU-Auslastung, die den Zustand eines oder mehrerer Hauptprozessoren eines Computers beschreibt (prozentual). Die CPU-Auslastung steigt mit der benötigten Rechenzeit für die Bewältigung laufender Prozesse.

Speicherauslastung
Da ein HTTP-Request einen *Thread* startet, wird dafür serverseitig Speicher allokiert. Je nachdem wie viele kleinere Verbindungen bei der asynchronen Kommunikation geöffnet werden, folgt daraus die Anforderung nach einer besseren Speicherauslastung. Eine Lösung besteht darin, mehr als einen Prozess mit mehreren *Threads* für die Verarbeitung zu verwenden. Es wird dafür immer eine definierte Anzahl von *Threads* zur Verfügung gestellt, um schnell auf Anfragen reagieren zu können. Man spricht von einem *Thread-Pool*. Nicht mehr verwendete *Threads* kehren in den Pool zurück. Die Speicherauslastung kommt nicht nur durch *Threads* zustande, sondern kann auch durch die genutzte Server-Software beeinflusst werden. Wie sieht es aber auf der Client-Seite aus? Hier ergibt sich ein Problem, wenn die Verbindung zu lange bestehen bleibt und sich somit Unmengen von Daten ansammeln können. Da heutzutage die wenigsten Browser mit bidirektionaler Kommunikation gerechnet haben, wird dies schnell zu einem Problem im Browser-Speicher, der den Dienst mit einem unschönen "*Out of memory*" quittieren kann.

Skalierbarkeit
Die Skalierbarkeit, im Kontext von Server-Push-Technologien, beschreibt die Anzahl von gleichzeitigen Anfragen an den Server und die damit auszuhaltende Last. Mit steigender Anzahl an gleichzeitigen Anfragen können Kapaziätsprobleme auftreten. Die Skalierbarkeit beschreibt auch die Anpassungsfähigkeit und die Erweiterbarkeit einer Software. In Bezug auf die bidirektionalen Kommunikationstechnologien ist es interessant zu wissen, ob man bestimmte Funktionen mit zunehmender Last auf zusätzliche Server verteilt werden kann.

Firewalls und Proxy-Server Toleranz
Manche bidirektionale Kommunikationen werden durch Firewalls und Proxy-Server blockiert. Die zugrunde liegenden Technologien bezeichnet man in diesem Fall als nicht Firewall- und Proxy-freundlich. Wie kommt das zustande? Grundsätzlich startet der *Roundtrip* eines Requests im Browser des Benutzers, passiert verschiedenste Netzwerke, Proxies und Gateways, bevor er beim Server landet, der die Antwort auf ähnlichem Weg zurücksendet. Firewalls und Proxy-Server können nicht nur die Latenzzeit erheblich erhöhen, wie oben erwähnt wurde, sondern auch die ganze Response zwischenspeichern und sie abbrechen. Der populärste Proxy-Server *Squid* hat z.B. standardmäßig ein *Timeout* von 15 Minuten. D.h. wenn keine neuen Response-Daten innerhalb dieser Zeit eingelesen

wurden, wird der langlaufende Request beendet und die dazugehörige Verbindung geschlossen. Diese Aussage trifft allerdings nicht auf alle Server-Push-Techniken zu und muss daher extra als ein Vergleichskriterium betrachtet werden.

Datenvolumen
Das Datenvolumen bezeichnet die Menge an Daten, die zur Übertragung der Informationen benötigt werden. Einige Client-Server-Kommunikationsverfahren setzen voraus, dass mehrere vollständige HTTP-Requests, inklusive HTTP-Header, an den Server geschickt werden. Der eigentlichen Übertragung der Anwendungsdaten kann auch ein Handshake vorausgehen, bei dem mehrere Bytes übertragen werden. Das verursacht unnötige Daten, die über die Netzwerkleitungen gehen. Die anderen Verfahren dagegen kommen mit schlanken HTTP-Requests aus. Intelligente Server-Push-Techniken versuchen möglichst viele Nutzdaten in eine Antwort zu packen und besitzen insofern einen höheren Informationsgehalt einer Nachricht, was zum wesentlich reduzierten Datenvolumen führt. Das Datenvolumen ist ein wichtiges Kriterium jeder Web-Technologie. Es beeinflusst nicht nur den Kostenfaktor, wenn man z.B. einen DSL Volumentarif hat, sondern hängt auch eng mit der Latenzzeit und der Bandbreitenlast zusammen.

Zustellsicherheit der Daten und Datenverlust
Eine verlustfreie Datenübertragung ist für geschäftskritische Anwendungen unerlässlich. Zustellsicherheit der Daten gibt die Garantie, dass die Daten während der Übertragung nicht verloren gehen. Je nach angewendeter Technik fällt die Garantiequote höher oder geringer aus. Allgemein können längere Abfrageintervalle und langlebige HTTP-Anfragen zu Datenverlusten führen. Das Problem besteht darin, dass eine Verbindung aus beliebigem Grund unterbrochen und wieder aufgenommen werden kann. Eine unterbrochene Verbindung verursacht meistens Datenverluste. Um dieses Manko zu überbrücken, führen manche Frameworks spezielle Mechanismen für eine langfristige Aufbewahrung nicht zugestellter Daten ein und garantieren zugleich eine spätere Zustellung. Man spricht in dem Fall von sogenannten „*delayed messages*“.

Datenaustauschformat
Unter einem Datenaustauschformat verstehen wir textuelle und binäre Repräsentationen der Daten. Bei den meisten bidirektionalen Technologien werden die Nachrichten in textueller Form übertragen. Es gibt allerdings fortgeschrittene Technologien, die diese Einschränkung aufheben und die Übertragung von binären Daten in der HTTP-Response erlauben. Binäre Daten sind beispielsweise Dokumente, Bilder, Videos. Die Fähigkeit der Übertragung von binären Nachrichten hängt von dem zugrundeliegenden Protokoll ab. Bei dem synchronen Datenaustausch stellt die Übertragung von binären Daten kein Problem dar.

Umgang mit dem Verbindungslimit
Die HTTP-Spezifikation schränkt die maximal parallel laufenden Verbindungen zu einem Host auf zwei ein. Das ist darin begründet, dass die meisten Server nur zwei offene Verbindungen zu einem Client zulassen. Moderne Browser, wie z.B. neue Firefox and Internet Explorer erhöhen zwar diese Zahl geringfügig, das Verbindungslimit für Anfragen an die gleiche Domain kann aber trotzdem schnell erreicht werden. Bleibt ein Browser-Request lange offen, kann der Browser keine neuen Verbindungen mehr aufbauen. Die

Arbeit in mehreren Browser-Tabs oder mit mehreren bidirektionalen Frameworks auf einer Seite (z.B. eine eigene Framework-Instanz in jedem Frame) ist dann gar nicht möglich. Diese Schwachstelle weisen allerdings nicht alle bidirektionalen Kommunikationsprotokolle auf. Wie wir im Kapitel 4 gesehen haben, gibt es Protokolle, die eine problemlose bidirektionale Arbeit in mehreren Browser-Tabs ermöglichen. Eine Nachricht kann hier auch an mehrere Browser-Tabs (Applikationsinstanzen) geliefert werden. An dieser Stelle sei noch zu erwähnen, dass die Verwendung von künstlichen, virtuellen Subdomains oder Hosts eine mögliche Abhilfe für das Verbindungslimit schafft.

Implementierungsaufwand
Der Aufwand bei der Implementierung eines Frameworks für eine konkrete bidirektionale Kommunikationstechnologie ist ein weiteres wichtiges Vergleichskriterium. Bei manchen Technologien ist der Aufwand recht groß und bei anderen dagegen sehr gering, weil die jeweilige Technologie bereits die gesamte Infrastruktur / Laufzeitumgebung (*software environment*) mit sich bereitstellt oder sogar *native* ist und durch Browser automatisch unterstützt wird. Die Einfachheit der Implementierung hängt auch davon ab, inwieweit das zugrundeliegende Kommunikationsprotokoll einfach ist.

Verbreitung und Unterstützung in Frameworks
Die Verbreitung einer Technologie ist ein Indiz dafür, dass sie sich erfolgreich weiterentwickelt und bekannt ist. Eine verbreitete Technologie, die auch in vielen Frameworks unterstützt wird, findet mehr Anwendungsgebiete als diejenige, die weniger unterstützt wird. Einige Techniken für die bidirektionale Kommunikation haben nur eine theoretische oder akademische Bedeutung und sind nicht für den praktischen Einsatz geeignet. Es gibt auch Ansätze, die in dieser Arbeit nicht aufgeführt wurden, da sie heutzutage keine Relevanz für den praktischen Einsatz haben und noch nicht in Form eines Frameworks implementiert wurden. Die verfügbaren Frameworks werden in dem Kapitel 7.1 besprochen.

5.3 Gegenüberstellung und Vergleich

5.3.1 HTTP Polling

Latenzzeit und Performance
Das Polling-Verfahren besitzt eine hohe Latenzzeit, die sich aus dem Pollingintervall ergibt. Je höher das Pollingintervall ist, desto höher ist die Latenzzeit. HTTP Polling garantiert keine zeitnahen *Updates*. In der Abbildung 8 sieht man, dass manche Responses informationslos („leer“) sind, wenn es keine *Updates* zur Zeit der Abfrage gab. Die Reaktionszeit auf das Vorhandensein neuer Daten ist die schlechteste von allen hier betrachteten Verfahren (abgesehen von Piggyback). Es gibt mittlerweile eine optimierte Variante *PredictedPoll,* bei der der Ereignisabstand bekannt ist und nur die wirklich notwendigen Anfragen durchgeführt werden müssen. Trotzdem besteht hier ein weiteres Problem, wenn die Poll-Abfrage kurz vor dem Ereignis eintrifft. Die Nachrichtenlaufzeit liegt in dem Fall nicht bei der Signallaufzeit plus die Verarbeitungszeit am Server, sondern es kommt zusätzlich die Distanz zum nächsten Poll-Ereignis hinzu.

Serverlast
Das HTTP-Polling verursacht eine hohe Serverlast. Wenn das Pollingintervall z.B. mit einer Sekunde definiert wurde, müssen jede Sekunde bei 100 gleichzeitigen Besuchern 100 Anfragen gestellt werden, auch wenn es keine Änderungen gibt.

Speicherauslastung
Wegen vielen und häufigen HTTP-Requests ist die serverseitige Speicherauslastung sehr hoch, weil mehrere *Threads* parallel geöffnet werden (normalerweise ein *Thread* pro Request). Die Speicherbelegung auf dem Server steigt bei vielen Benutzern enorm hoch. Die clientseitige Speicherauslastung stellt kein Problem dar – es werden ganz normale AJAX-Request gesendet. Für jede Aufgabe wird ein eigener HTTP-Request aufgebaut, verarbeitet und dann das Ergebnis verworfen.

Skalierbarkeit
Beim Polling entsteht zwar eine hohe Serverlast, basierende auf dem Polling Lösungen sind aber gut skalierbar. Die HTTP-Requests unterscheiden sich nicht von den herkömmlichen und man kann somit bestimmte Funktionen auf zusätzliche Server verteilen. Systeme für *Load Balancing* (Verfahren für die Einhaltung der Kapazitätsgrenzen bei der Speicherung, dem Transport und der Verarbeitung von Objekten), die zur Lastverteilung dienen, sind in diesem Fall gut einsetzbar.

Firewalls und Proxy-Server Toleranz
Das Polling-Verfahren hat keine Nachteile in Bezug auf Firewalls und Proxy-Server, weil sich der Request / Response-Lebenszyklus nicht von dem herkömmlichen unterscheidet. Es gibt keine dauerhaft offen gehaltenen Verbindungen, so dass das Problem mit dem Caching oder dem Response-Abbruch hier nicht auftritt.

Datenvolumen
Periodische HTTP-Requests, -Responses weisen keine Besonderheiten auf. Anfragen sind ganz normale, nicht optimierte POST-Requests mit allen Informationen in HTTP-Header und HTTP-Body. Antworten sehen ebenso keine Optimierungen bezüglich der Nutzdaten vor. Das Datenvolumen ist daher hoch im Vergleich zu den anderen Technologien.

Zustellsicherheit der Daten und Datenverlust
Wenn das Abfrageintervall relativ lang ist, kann es zu Datenverlusten führen. Wird die Verbindung zwischen zwei Abfragen unterbrochen, gehen neue Daten (Aktualisierungen), die dem Client noch nicht bekannt sind, möglicherweise verloren. Die Zustellsicherheit der aktualisierten Daten ist mit langen Abfrageintervallen nicht garantiert.

Datenaustauschformat
Es gibt kein Problem sowohl textuelle als auch binäre Nachrichten an den Client zu schicken, falls serverseitig neue Informationsbestände vorliegen.

Umgang mit dem Verbindungslimit
Das Verbindungslimit wird beim Polling umgangen, da keine offen gehaltenen und somit

blockierten Verbindungen existieren. Es sei denn, der Benutzer öffnet mehrere Browser-Tabs, so dass jedes Tab die Polling-Requests gleichzeitig mit den anderen Tabs sendet. Parallel können dann nicht mehr als 2 bis 8 Requests an einen Host geschickt werden. Mehrere Anfragen werden nicht blockiert, sondern einfach sequentiell abgearbeitet.

Implementierungsaufwand
Die Polling-Technik ist einfach zu implementieren. Es sind keine speziellen Anforderungen an die Hardware / Software nötig. Das Polling kann mit üblichen Javascript-Mitteln realisiert werden, indem man die Funktion `setTimeout()` aufruft, die eine andere Funktion und die Zeit als Parameter erwartet. Diese andere Funktion wird immer wieder nach dem Ablauf der vorgegebenen Zeit aufgerufen. Man kann in der Funktion somit periodische AJAX-Anfragen abschicken, um beim Server nach aktualisierten Informationen zu fragen.

Verbreitung und Unterstützung in Frameworks
Das Polling-Verfahren ist dank seiner Einfachheit weit verbreitet und wird in fast jedem Framework als die einfachste Art der bidirektionalen Kommunikation implementiert. Es schlägt laut Studien deutlich alle anderen Verfahren in der Verbreitung, erfreut sich aber keiner großen Beliebtheit aufgrund der geschilderten Nachteilen. Man sollte sich bewusst sein, dass es sich bei dieser Technik um keine echte bidirektionale Kommunikation handelt.

5.3.2 Comet mit HTTP Long-Polling

Latenzzeit und Performance
Beim HTTP Long-Polling besteht zwar auch das Latenzproblem, allerdings ist die Latenz geringer als beim Polling, welches mit dem Abfrageintervall arbeitet. Andererseits weist das HTTP Long-Polling eine höhere Latenzzeit gegenüber dem HTTP Streaming auf.

Serverlast
Die Serverlast hängt beim Long-Polling davon ab, wie oft neue bzw. aktualisierte Informationsbestände vorliegen, über die der Client informiert werden muss. Nachdem neue Daten bekanntgegeben worden sind, wird der Request / Response-Lebenszyklus abgeschlossen und einen neuen Request gesendet (Abbildung 7). Dies führt zu erhöhter Serverlast bei häufigen Anfragen und mehreren Benutzern. Bezüglich der Serverlast, nimmt das Long-Polling eine mittlere Position zwischen normalem Polling und HTTP Streaming ein.

Speicherauslastung
Die Speicherauslastung hängt wieder davon ab, wie schnell neue *Updates* verfügbar sind. Auslieferung neuer *Updates* bedeutet das Senden neuer Request, was unter Umständen die Anzahl der *Threads* und den belegten Speicher serverseitig erhöht. Clientseitig gibt es keine nennenswerte Probleme damit.

Skalierbarkeit
Eine gute Skalierbarkeit beim Long-Polling ist nicht möglich. Es sind spezielle Server nötig, die auf die Skalierbarkeit ausgerichtet sind und mehrere tausend Verbindungen offen halten

können.

Firewalls und Proxy-Server Toleranz
Wenn einer Web-Applikation ein Firewall oder Proxy vorgeschaltet ist, gibt es meistens keine Probleme mit dem Caching oder dem Response-Abbruch. Comet mit dem Long-Polling gilt offiziell als proxy-freundlich. Die Ursache dafür liegt in der Natur der Datenübertragung. Die Datenübertragung an den Client setzt kein *Chunked Transfer Encoding* voraus und muss nicht in kleinen Datensegmenten (*chunks*) erfolgen. *Chunked Transfer Encoding* bedeutet, dass die Größe der Datenmenge (*Content-Length*) in HTTP-Header nicht gesetzt ist und der Server ohne das Wissen der Größe sofort mit der Übertragung dynamischer Informationen beginnen kann. *Chunked Transfer Encoding* ist für das HTTP Streaming erforderlich.

Datenvolumen
Der Long-Polling-Ansatz hat auch ein gewisser *Overhead* bei der Anzahl der übertragenen Datenmenge. Es werden mehrere vollständige Requests an den Server geschickt, inklusive HTTP Header. Dies verursacht unnötiges Datenvolumen, welches über die Datenleitung gesendet werden muss. Gegenüber dem einfachen Polling ist das zu übertragende Datenvolumen jedoch geringer.

Zustellsicherheit der Daten und Datenverlust
Die Zustellsicherheit der Daten ist höher als beim regulären Polling. Trotzdem können auch hier neue Daten zwischen einer Response und einem darauf folgenden neuen Request verloren gehen. Die Wahrscheinlichkeit des Datenverlustes innerhalb dieser kleinen Zeitspanne ist allerdings sehr gering.

Datenaustauschformat
Theoretisch sind sowohl textuelle als auch binäre Nachrichten möglich. Praktisch findet nur noch das textuelle JSON-Format seine Anwendung. Das Bayeux-Protokoll unterstützt lediglich Subscribe- bzw. Publish-Nachrichten im JSON-Format.

Umgang mit dem Verbindungslimit
Die Verbindung beim Long-Polling wird für einen bestimmten Zeitraum offen gehalten, bis neue Daten vorliegen. Innerhalb dieses Zeitraumes besteht das Verbindungslimit, da heutige Web-Browser im Normalfall nur eine begrenzte Anzahl von parallelen HTTP-Anfragen an den gleichen Host stellen können.

Implementierungsaufwand
Der Aufwand bei der Implementierung der Long-Polling Technik ist zwar höher als bei einem regulären Polling, bleibt jedoch geringer im Verhältnis zum HTTP Streaming. Im Verhältnis zu allen anderen bidirektionalen Technologien ist der Aufwand relativ hoch. Es gibt keine automatische Unterstützung für diese Comet-Technologie, wie dies z.B. beim WebSocket-Protokoll der Fall ist oder durch die Goolge App Engine gewährleistet wird.

Verbreitung und Unterstützung in Frameworks
Wie alle Comet-Technologien findet auch das Long-Polling eine breite Unterstützung in

existierenden Frameworks und Engines, beispielsweise in den Frameworks Atmosphere [ATURL], CometD [CDURL] und DWR [DIURL], um nur einige zu nennen.

5.3.3 Comet mit HTTP Streaming

Latenzzeit und Performance
Das HTTP Streaming hat eine geringe Latenzzeit und eine gute Performance, weil die Auslieferung von Daten an den Client in kleinen Teilstücken, sogenannten *chunks*, über eine einzige offene Verbindung erfolgt. Als einziger Konkurrent kann nur der WebSocket-Standard die Streaming-Technologie bei der Performance übertreffen. Durch die Benutzung einer und derselben Verbindung fällt die Zeit für den zusätzlichen Aufbau und das Schließen einzelner HTTP-Verbindungen weg. Die Reaktionszeit ist deshalb hoch und das HTTP Streaming ist bei diesem Kriterium beinahe ideal. Trotzdem gibt es das Problem, dass diverse Firewalls und Proxy-Server die Latenzzeit erheblich erhöhen können, wenn sie die Response zwischenspeichern.

Serverlast
Die Serverlast ist sehr gering. Falls der Server bei der bidirektionalen Kommunikation ausgelastet wird, wird es empfohlen, die betroffene Web-Applikation auf das HTTP Streaming umzustellen.

Speicherauslastung
Die Speicherauslastung ist auch sehr gering, weil nicht zu viel Speicher für neue *Threads* (Requests) belegt wird. Dennoch kann das HTTP Streaming schnell zu einem Problem im Browser-Speicher werden. Der verbrauchte Speicher ist abhängig von den drei Faktoren - Menge, Komplexität und Intervall der serverseitigen Ereignissen.

Skalierbarkeit
Wie auch beim Long-Polling ist eine gute Skalierbarkeit hier nicht gewährleistet. *Load Balancing,* die zur Lastverteilung dient, ist mit langwierig offen gehaltenen Verbindungen problematisch.

Firewalls und Proxy-Server Toleranz
Dem HTTP Streaming liegt nach wie vor das HTTP-Protokoll mit allen seinen Schwächen zu Grunde. Wie bei der Latenzzeit bereits erwähnt wurde, stellt das Streaming-Verfahren ein echtes Problem in dem Zusammenhang mit Firewalls und Proxy-Servern dar. Die Auslieferung von Datensegmenten (*chunks*) in einer permanent aufrecht erhaltenen Client-Server-Verbindung kann durch Firewalls und Proxy-Server gecacht und zu einem späteren Zeitpunkt weitergeleitet werden. Im schlimsten Fall kann eine langlebige Verbindung durch Proxy-Server sogar unterbrochen werden, wenn z.B. ein *Timeout* auftritt. Deshalb definieren fast alle Streaming-Lösungen ein *Fallback*, indem sie auf das Long-Polling Verfahren ausweichen. Alternativ können auch SSL-Verbindungen aufgebaut werden, um die Response vor dem Zwischenspeichern zu schützen. Allerdings steigen dann die Latenzzeit (durch SSL-Handshakes) und die serverseitige Ressourcenverwalung. Verschiedene Firewalls und Proxy-Server werden typischerweise für einen Internet-Zugang eingerichtet. Innerhalb des Intranets (in einer geschlossenen Einheit, wie z.B. einer Firma) ist solch eine

Einrichtung meistens belanglos. Die Fausregel für den Einsatz vom HTTP Streaming lautet somit im Bezug auf das geschilderte Problem – empfohlen für Web-Anwendungen im Intranet und nicht wirklich empfohlen für das Internet.

Datenvolumen
Die Streaming-Technik hat den Vorteil, dass man über eine Verbindung beliebig viele Ereignisse übertragen und so den *Overhead* durch den Verbindungsauf-/abbau vermeiden kann. Das Datenvolumen ist somit nicht hoch. Allerdings gibt es auch Technologien, wie z.B. das WebSocket-Protokoll, die mit noch wenigen Bytes an Datenvolumen auskommen.

Zustellsicherheit der Daten und Datenverlust
Die Zustellsicherheit der Daten bei dieser Technik ist sehr hoch und wird als eine der besten von allen bidirektionalen Kommunikationsverfahren bewertet, denn die Response wird hier nicht geschlossen, wie bei den oben bewerteten Polling-Verfahren. Ein kleines Problem gibt es dennoch. Das `XMLHttpRequest`-Objekt der älteren Versionen des Internet Explorers hat die Eigenheit, erst eine Antwort zu liefern, wenn die Verbindung geschlossen wird. Bei den Versionen des Internet Explorers 5.5 und älter wurde auch berichtet, dass der Browser eine länger als 5 Minuten offene Verbindung als eine abgelaufene Verbindung auffasst, wenn keine Daten innerhalb dieser Zeit empfangen wurden [HSURL]. Als Folge zeigt er eine falsche Fehlermeldung.

Datenaustauschformat
Wie auch beim HTTP Long-Polling werden die Nachrichten im JSON-Format ausgetauscht, weil die Datenübertragung auf dem Bayeux-Protokoll basiert.

Umgang mit dem Verbindungslimit
Das Problem mit dem Verbindungslimit tritt beim HTTP Streaming vermehrt auf. Es wird oft bemängelt, dass es durch die Herstellung einer kontinuierlichen Verbindung zum Server unmöglich ist, gleichzeitig Daten über diese Verbindung hochzuladen. Um das Hochladen der Daten in einer Comet-Applikation zu realisieren, muss der Client parallel eine zweite Verbindung zum Server aufbauen, was bei steigender Benutzerzahl zu erheblichen Problemen führen kann. Außerdem macht der Internet Explorer wieder Schwierigkeiten, indem er geöffnete Verbindungen mit einem ewig drehenden Icon quittiert. Das Verhalten irritiert auf Dauer den Benutzer. Dieses Problem kann mit eingebetteten unsichtbaren IFrames umgegangen werden. Allerdings erkennt auch hier der Browser, dass `iframe` nicht komplett geladen wurde und die Seite als nicht fertig geladen anzeigt. Das zieht unerwünschte Konsequenzen nach sich, wie z.B. Ladebalken in der Statusleiste.

Implementierungsaufwand
Die Implementierung eines Frameworks für das HTTP Streaming ist mit dem erheblichen Aufwand verbunden, da diese Technik nur in Browsern funktioniert, die das `XMLHttpRequest`-Objekt W3C konform implementieren. In dem Kapitel 4.3.2 wurde erläutert, dass die Daten in „*readystate*“ 3 bereits vorhanden sein müssen. Beim Internet Explorer ist dies leider nicht der Fall. Er wechselt direkt von „*readystate*“ 2 nach „*readystate*“ 4, so dass man erst dann die Daten abfragen kann, wenn es schon zu spät ist. Allerdings implementieren nahezu alle anderen gängigen Browser das `XMLHttpRequest`-

Objekt W3C konform, sodass diese Technik sehr häufig Anwendung findet. Für den Internet Explorer wird dann eine eigene Technik genutzt. Dabei macht man es sich zu Nutze, dass der Javascript Code in einem `<script>`-Tag direkt nach dem Schließen des Tags bereits ausgeführt wird. Die Verbindung wird nun nicht mehr über das `XMLHttpRequest`-Objekt aufgebaut, sondern mit Hilfe eines unsichtbaren `iframe`. Der Webserver überträgt die Ereignisse dabei direkt als auszuführenden Javascript Code, der in `<script>`-Tags eingeschlossen ist. Sobald ein Ereignis übertragen wurde und der Browser ein abschließendes `</script>`-Tag empfangen hat, wird der enthaltene Javascript-Code automatisch ausgeführt und das Ereignis ausgewertet.

Verbreitung und Unterstützung in Frameworks
HTTP Streaming ist weit verbreitet und findet z.B. beim E-Mail-Dienst *Google Mail* [GMURL] seine Anwendung, um die Mail-Oberfläche bei neuen Nachrichten in Echtzeit zu aktualisieren und ihr das Aussehen einer Desktop-Applikation zu verleihen. Die Unterstützung in Frameworks ist ebenso hervorragend.

5.3.4 Piggyback

Latenzzeit und Performance
Das Piggyback-Verfahren besitzt eine unter Umständen sehr hohe Latenzzeit. Es hängt von den Benutzeraktivitäten ab, wie schnell neue Daten auf der Clientseite verfügbar sind. Wenn der Benutzer sich mit einer zeitintensiven Aufgabe beschäftigt und keine Aktivitäten lange Zeit macht, wie beispielsweise das Lesen eines Blogs, bleibt die Webseite nicht aktualisiert, obwohl neue Inhalte bereits verfügbar sind.

Serverlast
Die Serverlast unterscheidet sich nicht von der Serverlast konventioneller Web-Applikationen. Die Technik basiert auf ganz normallen HTTP-Requests, -Responses, schickt keine periodischen Poll-Abfragen und blockiert keine Verbindungen bzw. hält sie nicht offen über eine lange Zeit.

Speicherauslastung
Die Speicherauslastung gleicht fast der einer konventioneller Web-Applikation. Doch es gibt einen kleinen Unterschied. Der Speicher wird bei der Piggyback-Technologie insofern mehr belegt, als sich zu aktualisierende Daten (*Updates*) mit der Zeit auf der Serverseite kontinuierlich ansammeln können, falls neue HTTP-Anfragen von dem Client nur selten ankommen.

Skalierbarkeit
Web-Applikationen, die die Piggyback-Technologie einsetzen, sind gut skalierbar, da langlebige Verbindungen nicht berücksichtigt werden müssen. Man kann bestimmte Funktionen auf zusätzliche Server ohne jegliche Probleme verteilen, wenn die Benutzerzahl ansteigt.

Firewalls und Proxy-Server Toleranz
Beim Piggyback-Verfahren handelt es sich um die herkömmlichen Request / Response-

Lebenszyklen, so dass diese Technologie keine Nachteile mit Firewalls und Proxy hat.

Datenvolumen
Das Datenvolumen ist wahrscheinlich das geringste aus allen vorgestellten Verfahren, mit der Ausnahme der WebSocket-Kommunikation. Die Daten werden „transparent“, in dem normalen Request- / Response-Ablauf übertragen. Das Request-Response-Paradigma ändert sich hier gar nicht und beeinflusst daher nicht das Datenaufkommen.

Zustellsicherheit der Daten und Datenverlust
Die Zustellsicherheit der aktualisierten Daten ist nicht hundertprozentig garantiert. Je mehr Zeit zwichen zwei Requests verstrichen wird, umso höher ist die Wahrscheinlichkeit, dass die Daten verloren gehen können.

Datenaustauschformat
Es ist nur noch der Datenaustausch textueller Nachrichten möglich, weil mehrere Nachrichten unter Umständen zu einer großen Nachricht zusammengefasst werden müssen. Die andere Schwachstelle beim Piggyback, die oft bemängelt wird, besteht in dem Zusammenhang zwischen dem Response-Inhalt (*content of the response*) und der Request-Art (*nature of the request*). Der Response-Inhalt passt nicht genau auf den Request. Sinngemäß bzw. inhaltsmäßig stehen sie nicht unbedingt in einer Beziehung zueinander.

Umgang mit dem Verbindungslimit
Beim Piggyback besteht das Problem des Verbindungslimits nicht. Die Piggyback-Technik unterscheidet sich nicht vom Polling mit einem Abfrageintervall bei diesem Kriterium.

Implementierungsaufwand
Der Implementierungsaufwand ist größer als beim Polling und vermutlich sogar den Comet-Technologien. Die Erfassung der aktualisierten Daten und das Anwenden aller gesammelten Änderungen in der richtigen Reihenfolge ist keine triviale Aufgabe, wie es auf den ersten Blick scheinen mag.

Verbreitung und Unterstützung in Frameworks
Piggyback ist nicht so populär, wie das Polling oder die Comet-Techniken, weil es gewisse Benutzeraktivitäten erfordert, bevor eine Webseite die aktuellen Änderungen erfährt. Es findet auch kaum Unterstützung in Frameworks.

5.3.5 OpenAjax-Push

Latenzzeit und Performance
Da bei der OpenAjax-Push-Technologie eine einzige Verbindung unter vielen Clients für reine Benachrichtigungen geteilt wird, bekommen die Clients sehr schnell mit, wenn neue *Updates* verfügbar sind. Die Latenzzeit dafür ist sehr gering. Die *Updates* selbst müssen die Clients aber erst noch anfordern. Dafür schicken sie ganz normale HTTP-Requests. Dies bedeutet wiederum, dass die Latenzzeit größer als bei den meisten bidirektionalen Technologien ist. Als Ausnahmen können Polling, Piggyback und Ajax-on-Demand aufgelistet werden. Diese haben unter Umständen eine noch schlechtere Performance (es

komm auf das Abfrageintervall an).

Serverlast
Das OpenAjax-Push könnte man als einen Hybrid, bestehend aus dem Polling und den Comet-Technologien bezeichnen, so dass die Serverlast irgendwo dazwischen liegen sollte.

Speicherauslastung
Die Speicherauslastung liegt genauso in der Mitte zwischen dem Polling und den Comet-Technologien, wie auch die Serverlast. D.h. die CPU-Auslastung ist niedriger als bei dem Polling und höher als bei dem HTTP Long-Polling / Streaming.

Skalierbarkeit
Die Technologie wurde mit dem Gedanken einer effizienteren Skalierbarkeit entworfen. Serverseitiger Zustand (*state*) ist minimal, Benachrichtigungen werden im Normalfall nicht zwischengespeichert und das Protokoll stellt alle erforderlichen Informationen bei jeder erneuten Verbindung (*reconnect*) bereit, um das *Clustering* (Rechnerverbund von vernetzten Computern, die von außen in vielen Fällen als ein Computer gesehen werden können) zu unterstützen.

Firewalls und Proxy-Server Toleranz
Das OpenAjax-Push weist keine Probleme mit Firewalls und Proxy-Servern auf. Es gibt keine langlebigen Verbindungen, die gecacht (auf dem Proxy-Server zwischengespeichert) oder unterbrochen werden könnten.

Datenvolumen
Wegen einer Verbindung für die Benachrichtigungen und der anderen für den eigentlichen Datenaustausch ist das Datenvolumen höher als bei allen anderen Technologien, mit zwei Ausnahmen von Polling und Ajax-on-Demand.

Zustellsicherheit der Daten und Datenverlust
Die Zustellsicherheit ist zwar hoch, bleibt aber geringer als bei Comet-Technologien, weil ein *Roundtrip* nötig ist, um neue Daten abzuholen. Andererseits kann sie zugleich auch höher im Vergleich zu den anderen Technologien ausfallen, wenn Proxy-Server im Spiel sind. BOSH und WebSocket schneiden bei diesem Kriterium in jedem Fall besser ab.

Datenaustauschformat
Das Datenformat kann hier beliebig sein - entweder textueller oder binärer Natur. Der Server kann die Daten in allen möglichen Formaten im Rahmen einer herkömmlichen Response an den Client schicken.

Umgang mit dem Verbindungslimit
Das OpenAjax-Push-Protokoll hebt das Verbindungslimit auf, dass die Anzahl der gleichzeitigen Verbindungen von einem Client zum einem Host auf maximal 2 bis 8 (je nach Browser) beschränkt ist. Das Protokoll ermöglicht eine problemlose bidirektionale Arbeit in mehreren Browser-Tabs, da nur noch eine einzige Verbindung blockiert wird. Man kann somit mehrere Instanzen einer Web-Applikation in verschiedenen Browser-Tabs öffnen, was

bei Comet-Technologien gar nicht möglich ist.

Implementierungsaufwand
Ein Framework auf Basis des OpenAjax-Push-Protokolls kann relativ einfach implementiert werden. Der Aufwand dafür ist geringer als bei den Comet-Technologien und höher als bei den Google's Channel API und WebSocket, weil die letztgenannten all die Mittel bereits an die Hand geben, die man für die Implementierung braucht.

Verbreitung und Unterstützung in Frameworks
Das OpenAjax-Push ist noch nicht so weit verbreitet wie die Comet-Technologien. Die OpenAjax Alliance bemüht sich dabei, die Verbreitung durchzusetzen. Das wohl bekannteste Framework, das dieses Push-Protokoll implementiert, ist ICEPush von der ICEsoft Technologies [ICURL]. Eine bidirektionale Kommunikation auf Basis von ICEPush kann laut Hersteller mit vielen anderen Web-Frameworks integriert werden - JSP, JSF, Spring MVC, Grails, GWT, Wicket, JQuery und Prototype.

5.3.6 Ajax-on-Demand

Latenzzeit und Performance
Ajax-On-Demand stellt eine bessere Alternative zur regulären Polling-Technologie dar. Dies spiegelt sich in einer geringeren Latenzzeit wieder. Die geschätzte durchschnittliche Antwortzeit für die ressourcensparenden, kurz laufenden Anfragen aus der ersten Phase ist vergleichbar mit der Antwortzeit eines systemnahen TCP/IP Ping (ein Diagnose-Werkzeug, mit dem überprüft werden kann, ob ein bestimmter Host in einem IP-Netzwerk erreichbar ist). Ansonsten ist das Ajax-on-Demand immer noch ein reguläres Polling, wenn auch „leichtgewichtig“. D.h. die Latenzzeit und die Performance sind bei diesem Verfahren nicht besonders gut und unterlegen den anderen Verfahren, die in dieser Diplomarbeit betrachtet werden. Ein Grund dafür besteht darin, dass eine bidirektionale Kommunikation hier genauso wie bei einem normalen Polling nur noch simuliert wird.

Serverlast
Die ressourcensparenden Anfragen aus der ersten Phase mildern die Serverlast. Es wurde gemessen, dass ein Tomcat-Server 5.5 mit einem 3 GHz Prozessor und einem 10 MBits Datendurchsatz in der Lage ist, mehr als 10000 solcher HTTP-Anfragen gleichzeitig zu verarbeiten [DEURL]. Trotzdem steigt die Last auf dem Server rapide mit der zunehmender Anzahl von Benutzern an. Abgesehen von dem regulären Polling ist die Last im Durchschnitt höher als bei den meisten anderen Verfahren.

Speicherauslastung
Mehrere parallel geöffnete *Threads* sollten hier im Allgemeinen keine Probleme bereiten. Die Anzahl der Requests ist zwar hoch, aber die meisten davon leben sehr kurz und schließen somit geöffnete *Threads* auch schnell.

Skalierbarkeit
Der Ansatz mit dem Ajax-on-Demand ist gut skalierbar. Es gibt keine Besonderheiten zu beachten. *Load Balancing*, *Session-Replication*, *Clustering* und jede andere Maßnahme zur

Lastverteilung ist problemlos anwendbar.

Firewalls und Proxy-Server Toleranz
Offensichtlich soll das Ajax-on-Demand die Problematik mit vorgeschalteten Firewalls und Proxy-Servern lösen. Die HTTP-Response wird nicht gecacht oder abgebrochen.

Datenvolumen
Das Datenaufkommen ist hoch, weil es sich um periodische Abfragen handelt. Allerdings sind die Abfragen optimiert. Es werden nur noch HEAD-Requests (Nachrichtenkopf ohne Dokumenteninhalt) abgeschickt, um die Existenz neuer Daten zu prüfen. Das Datenvolumen ist deshalb niedriger als beim Long-Polling, bleibt jedoch hoch im Vergleich zu den anderen Technologien.

Zustellsicherheit der Daten und Datenverlust
Verfechter der Ajax-on-Demand-Technologie behaupten, dass die auf langlebigen Verbindungen basierende Server-Push-Technologien nicht zuverlässig sind. Zum einen wegen Proxy-Servern und zum anderen wegen dem Charakter vom TCP-Protokoll, das dem HTTP-Protokoll zugrunde liegt. Wenn ein Datenfragment kleiner als die zusendende Paketgröße ist, wird das Datenfragment solange zwischengespeichert, bis sich andere Fragmente ansammeln und die Paketgröße erreicht wird. Die Verfechter dieser Technologie stufen deshalb die Zustellsicherheit der Daten beim Ajax-on-Demand relativ hoch. Das trifft auch zu, wenn das Abfrageintervall klein eingestellt ist.

Datenaustauschformat
Die Nutzdaten können in jedem beliebigen Format übertragen werden. Bei diesem Kriterium gibt es keine Einschränkungen.

Umgang mit dem Verbindungslimit
Das Problem der Begrenzung der Anzahl von parallel laufenden Anfragen an eine Domain existiert bei dieser Technologie nicht.

Implementierungsaufwand
Die Implementierung ist genauso einfach wie beim Polling und es müssen keine speziellen Anforderungen an die Hardware / Software gestellt werden.

Verbreitung und Unterstützung in Frameworks
Die Verbreitung vom Ajax-on-Demand ist überraschenderweise nicht gut. Eine mögliche Umsetzung wurde einige Zeit lang in der JSF (*JavaServer Faces*) Bibliothek RichFaces angeboten. Dort gab es eine entsprechende Push-Komponente. Ansonsten gibt es keine bekannten Frameworks, die diese Technik nutzen. Es liegt vermutlich daran, dass sie nicht ausreichend bekannt ist und durch bessere Ansätze verdrängt wurde.

5.3.7 BOSH

Latenzzeit und Performance
Die Latenzzeit ist größer als beim XMPP-Protokoll, weil die BOSH-Technologie auf dem

HTTP basiert und das XMPP gewissermaßen umhüllt. Trotzdem kann man mit der Latenzzeit zufrieden sein. Tests haben z.B. bewiesen, dass man über das BOSH-Protokoll Millionen von Schachspielen in Echtzeit online spielen kann und die Performance in etwa dem populären Desktop-Spiel “Chesspark Client” angleicht [XMURL].

Serverlast
Die Serverlast ist mit den Comet-Techniken gleichgestellt. Man kann auch lesen, dass eine auf dem BOSH basierende Datenübertragung von vielen Nutzern aufgrund der hohen serverseitigen Last sowie unbefriedigender Stabilität nur als Notlösung betrachtet wird. Wegen der Einfachheit der Implementierung und der Verbreitung stellt die Serverlast allerdings kein entscheidendes Kriterium dar.

Speicherauslastung
Beim BOSH-Protokoll gibt es einen „Connection Manager“, der als „Übersetzungsproxy“ zwischen HTTP-Clients und dem XMPP-Server fungiert. Er muss zum einen die Protokolle „übersetzen“ und zum anderen ankommende HTTP-Anfragen „umorganisieren“. Die Speicherauslastung kann deshalb wegen dem „Connection Manager“ und dem damit verbundenen Verwaltungsaufwand relativ hoch ausfallen. Die Speicherauslastung ist eine zweite Schwachstelle der BOSH-Technologie. Generell wurde das HTTP-Protokoll nicht für die Vollduplex-Kommunikation entworfen. Das BOSH-Protokoll ist ein typisches Beispiel, wie die Vollduplex-Kommunikation durch zwei Verbindungen simuliert wird. Eine Verbindung wird für das Hochladen und die andere für das Herunterladen der Daten benutzt. Die Koordination der zwei Verbindungen führt zu einem höheren Ressourcenverbrauch, steigert die CPU-Last und die Komplexität im Allgemeinen.

Skalierbarkeit
BOSH-Anwendungen sind aufgrund geschickter Handhabung von zwei Verbindungen (im Kapitel 4.7.2 beschriebener Rollenwechsel) besser skalierbar als die Comet-Anwendungen. Es sind keine Probleme mit der Skalierbarkeit bekannt.

Firewalls und Proxy-Server Toleranz
Das BOSH-Protokoll gilt als firewall- und proxy-freundlich, weil es durch die geschickte Handhabung von ankommenden HTTP-Anfragen keine langlebigen Verbindungen gibt. Dadurch, dass bei jeder Datenübertragung in einer der beiden Richtungen eine HTTP-Anfrage abgeschlossen wird, tritt das Problem mit Proxy-Servern nicht auf. Es wird berichtet, dass der Port 5222 für reine XMPP-Anwendungen in Firmenumgebungen oft blockiert wird. Das BOSH-Protokoll dagegen umgeht dieses Problem ebenso gut.

Datenvolumen
CSS-, JavaScript-Dateien und HTML-Seiten werden in den heutigen Web-Applikationen häufig per GZip (Kompressionsverfahren) komprimiert. Für eine GZip-Komprimierung wird normalerweise ein spezieller *Servlet-Filter* benötigt, der diese Aufgabe übernimmt, wenn er die Daten durchschleust. Auch cleintseitig ist meist eine JavaScript-Lösung erforderlich. BOSH *Streams* haben eine eingebaute *out of the box* Kompression. D.h. die Daten werden automatisch komprimiert, wenn sie an den Client ausgeliefert werden. Die Kompression senkt das Datenvolumen, das beim BOSH ohnehin wegen geschickter

Handhabung von zwei Verbindungen gering ist.

Zustellsicherheit der Daten und Datenverlust
BOSH versteckt unzuverlässige Verbindungen und garantiert eine hohe Verfügbarkeit der Daten. Eine BOSH-Verbindung „lebt“ weiter, wenn auch die zugrundeliegende Internet-Verbindung ständig getrennt wird. Man kann eigenen Rechner mit einem BOSH-Client eingeschaltet lassen und sich an einen anderen Ort versetzen, der z.B. über einen öffentlichen drahtlosen Internetzugriffspunkt (*Hotspot*) verfügt. Die Verbindung wird nicht getrennt und deren Wiederherstellung ist nicht nötig, um weiter arbeiten zu können. Es sei denn, ein *Timeout* auf dem Server ist aufgetreten. Manche BOSH-Server speichern Responses und senden sie später erneut. Wenn eine Response aus irgendeinem Grund fehlgeschlagen ist, bekommt der Client zwischengespeicherte Daten trotzdem zugestellt, wenn er den letzten Request wiederholt. D.h. im Fall, wenn die IP-Adresse umgeschaltet wird (Beispiel mit dem Ortswechsel) oder die Netzwerk-Konnektivität schlecht / unzuverlässig ist, gehen die Daten nie verloren. Das BOSH-Protokoll zeichnet auch eine hohe Sicherheit aus. Es verfügt über sogenannte *End-to-End Signing and Object Encryption* für XMPP und unterstützt verschiedenste Verschlüsselungsverfahren, wie beispielsweise PGP und GPG.

Datenaustauschformat
Das Datenformat laut Spezifikation [BIURL] ist XML – eine plattformunabhängige Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdaten. Manche Abschnitte innerhalb XML können teilweise im JSON-Format vorliegen.

Umgang mit dem Verbindungslimit
Es existieren zwar keine langlebigen HTTP-Verbindungen, mindestens eine Verbindung pro Benutzer wird aber trotzdem zu jedem Zeitpunkt blockiert. So kann sehr schnell ein Verbindungslimit erreicht werden. Es gibt hier das gleiche Problem, wie bei Comet-Technologien.

Implementierungsaufwand
Eine BOSH-Anwendung kann dank der öffentlichen und stabilen Spezifikation des BOSH-Protokolls gut implementiert werden. Der Aufwand sollte dabei nicht hoch sein.

Verbreitung und Unterstützung in Frameworks
Ein großer Vorteil von XMPP ist, dass XMPP-Clients nahezu für jedes Betriebssystem und in jeder Programmiersprache existieren. Dieser Vorteil wird auch beim BOSH zu Nutze gemacht. Bekannte Netzwerke mit der XMPP- / BOSH-Unterstützung sind AIM, ICQ, Gadu Gadu, Facebook, IRC, MSN, MySpace, Twitter. Um das BOSH clientseitig für Echtzeit-Applikationen nutzen zu können, gibt es eine hervorragende JavaScript-Bibliothek namens Strophe.js [SRURL].

5.3.8 GAE Channel API

Latenzzeit und Performance

Channel API für die Google App Engine basiert auf beim XMPP-Protokoll. Die Latenzzeit ist deshalb vergleichbar mit der vom BOSH. Sie ist gering und hat eine interessante Besonderheit. Die Latenzzeit beim Google's Channel API bleibt konstant bei steigender Prozessoranzahl (CPU-Anteil). Die GAE-Plattform zum Entwickeln und Hosten von Web-Anwendungen garantiert eine konstante Latenzzeit und eine stabile Performance. Die Google-Infrastruktur gilt als hochperformant. Das Bild 16 zeigt die Abhängigkeit der Latenz von dem CPU-Anteil für zwei unterschiedliche Techniken - Polling und Channel API.

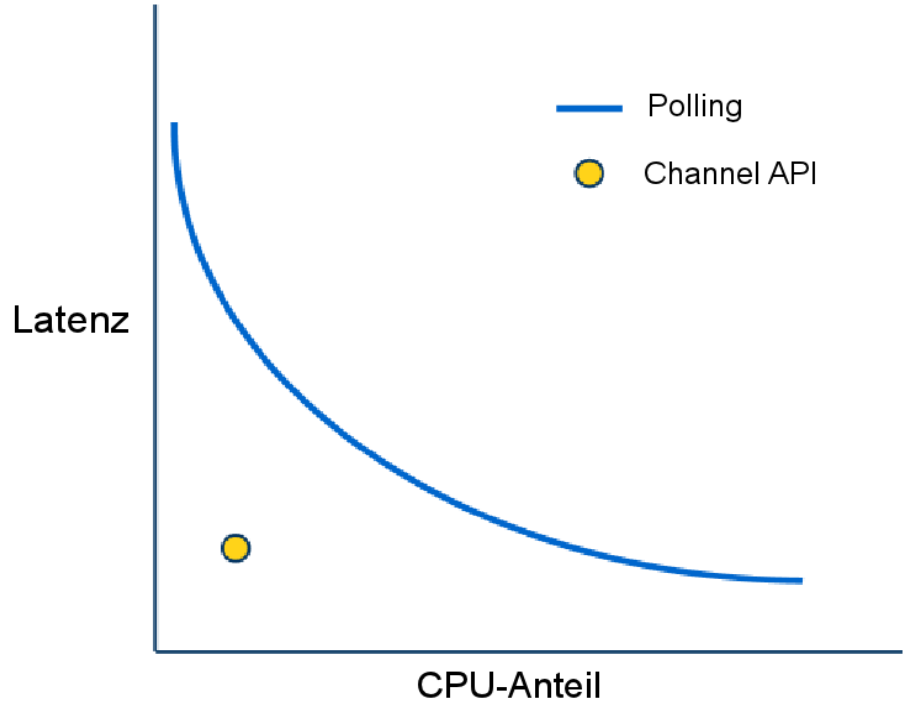

Abbildung 16: Latenz Polling vs. Channel API

In Tests konnte kein Unterschied in Antwortzeiten bei vielen Datenbankeinträgen und vielen parallel spielenden Benutzern festgestellt werden [GAE10, 26].

Serverlast

Die GAE-Infrastruktur verfügt über eine intelligente Technik für die Lastverteilung. Das Server-Verhalten kann anhand eines Kommunikationsbeispiels erklärt werden [GEURL].

1. Client sendet Request an Google App Engine (GAE).
2. GAE sucht anhand der URL nach entsprechender Applikation.
3. Der am schnellsten reagierende Server wird für die Applikation ausgewählt.
4. Die Request-Inhalte werden der Applikation übergeben und auf den Server geladen.
5. Die Applikation wird auf dem Server gestartet und die verarbeiteten Daten als Response an die GAE zurückgeschickt.
6. GAE schickt Response wieder an Client zurück.

Anhand dieses Beispiels sieht man, dass jede Applikation nicht permanent auf einem Server geladen ist. Um den Server zu entlasten, werden momentan nicht benötigte Applikationen nach einer gewissen Zeit vom Server entfernt bzw. benötigte Applikationen auf den Server geladen. Dies wirkt einer Überlastung des Servers entgegen und sichert gleichzeitig die Verfügbarkeit jeder Applikation.

Speicherauslastung
Bei der GAE kann man nicht über die Anzahl der geöffneten *Threads* reden und den verbrauchten Speicher nach dieser Anzahl messen. Die GAE unterstützt weder das Öffnen von Sockets noch *Threads*. Für jeden Request wird der am schnellsten reagierende Server ausgewählt. Die Speicherauslastung wird so quasi verteilt. Die GAE-Infrastruktur garantiert eine ausgeglichene Balance der CPU-Auslastung. Es gibt allerdings ein Problem, weil der *Google* Service nur unter gewissen Mengenbeschränkungen der genutzten Ressourcen kostenlos ist. Eine Applikation darf maximal 6.5 CPU-Stunden pro Tag verwenden. Es gibt auch andere Limits, wie der eingehende / ausgehende Datenfluß pro Tag, etc. Möchte man die Google App Engine deshalb für gewerbliche Zwecke nutzen, ist es ratsam, die Limits kostenpflichtig hochzusetzen / aufzuheben. So kann vermieden werden, dass die Anwender Fehlermeldungen aufgrund der erreichten Limits erhalten.

Skalierbarkeit
Wie auch beim BOSH sind keine Probleme mit der Skalierbarkeit bekannt. *Google* hat in den letzten Jahren ein sehr gutes Verständnis von Lastverteilung und Skalierung entwickelt, welches es GAE-Nutzern zur Verfügung stellt. Für Kundenanwendungen heißt das, dass eine hohe Anzahl von Anfragen auf der Webseite abgedeckt werden können. Die Entwickler der Google App Engine versprechen gleiche Skalierbarkeit, wie sie auch andere *Google* Applikationen besitzen. Der Leitspruch der GAE-Entwickler lautet: "all you have to do is write your application code and we'll do the rest" [GSURL]. Man braucht sich also nicht um die Anzahl der Zugriffe oder die Größe der Applikation zu sorgen, dies wird automatisch über die App Engine geregelt. Während eine Anwendung wenig Last aufweist, werden die Ressourcen automatisch für andere Anwendungen zur Verfügung gestellt.

Firewalls und Proxy-Server Toleranz
Der Umgang mit Firewalls und Proxy-Servern scheint keine Probleme zu bereiten. Offiziell wurde davon nichts berichtet.

Datenvolumen
Die transportierte Datenmenge ist nicht hoch. Der Client beginnt zwar die Kommunikation und fordert ein „Token“, das ihm der Server schickt, ein weiterer Kommunikationsverlauf ergibt aber keinen *Overhead*. Das transportierte Datenvolumen ist ungefähr mit der Comet Streaming-Technik gleichgestellt.

Zustellsicherheit der Daten und Datenverlust
Goolge App Engine ist eine Laufzeitumgebung, Entwicklungsplattform und Testumgebung in der Cloud. Beim Cloud Hosting werden die Daten oder Anwendungen nicht mehr auf Rechnern oder anderen Speichermedien gespeichert, sondern in einer "Daten-Wolke". Ein riesiger Vorteil des Cloud Hostings ist es, dass das Risiko eines Datenverlustes minimiert wird, indem ein defekter Server seine Tätigkeit auf einen anderen Server umleitet. Die Wahrscheinlichkeit des Datenverlustes bei der Nutzung des Channel APIs ist somit sehr gering. Das Channel API hat auch mit permanent offenen Kommunikationskanälen eine garantierte Nachrichtenzustellung. Die Sicherheitsaspekte sind dagegen noch nicht ausgereift. Wesentliche Probleme des Cloud Computings sind die noch ungenügende Datensicherheit und die Verschlüsselung bei der Datenübertragung [GAE10, 3].

Datenaustauschformat
Das textuelle JSON-Format ist das Datenaustauschformat jeder Client-Server-Anwendung, die auf GAE gehostet wird und das Channel API für die bidirektionale Kommunikation nutzt. Andere Formate sind bei dem proprietären Protokoll von *Google* nicht erlaubt.

Umgang mit dem Verbindungslimit
Das Channel API erzeugt eine persistente Verbindung zwischen der Web-Applikation und den Google's Servern und erlaubt damit das Senden von Nachrichten in Echtzeit an den Browser. Somit existiert hier das Problem des Verbindungslimits genauso wie z.B. beim HTTP Streaming. Dennoch spricht man beim Channel API nicht über das Verbindungslimit im Browser. Man spricht vielmehr von dem Limit der *Channels*. Um beispielsweise eine und dieselbe Konversation in mehreren Browser-Tabs oder mehrere Konversationen in einem Browser-Tab gleichzeitig führen zu können, müssen mehrere parallele *Channels* geöffnet werden. Die Anzahl der *Channels* ist aber begrenzt. Für das kostenlose Hosting liegt die momentane Anzahl der gleichzeitig verbundenen *Channels* bei 720. Für das kostenpflichtige Hosting liegt diese Anzahl bei 7200 *Channels*.

Implementierungsaufwand
Da das proprietäre Channel API bereits alles für die bidirektionale Kommunikation implementiert (sowohl server- als auch clientseitig), können viele kollaborative Anwendungen auf dessen Basis ohne viel Aufwand entwickelt werden. Das Channel API regelt dabei die Kommunikation zwischen Google's Servern via XMPP und schirmt es zugleich für den Datenverkehr nach außen zum Browser hin ab. Die proprietäre Schnittstelle kann aber auch als Nachteil angesehen werden, denn andere Technologien können nicht eingesetzt werden.

Verbreitung und Unterstützung in Frameworks
Ursprünglich wurde nur eine Laufzeitumgebung für die Programmiersprache Python angeboten. Seit April 2009 wird auch Java unterstützt. Die Java Laufzeitumgebung nutzt im Moment Java 6 und ermöglicht eine Anwendungsentwicklung mit verbreiteten Java Standards, wie Java Servlets oder Java Server Pages. Es werden aber auch andere Sprachen unterstützt, die für die Java Virtual Machine implementiert wurden, wie z.B. JavaScript, Ruby oder Scala. Für die bidirektionale Kommunikation stellt *Google* nur noch ein einziges eigenes Framework zur Verfügung – Channel API. Mit anderen Frameworks ist die bidirektionale Kommunikation auf der GAE nicht möglich.

5.3.9 Reverse HTTP

Latenzzeit und Performance
Beim Reverse HTTP kommt eine bidirektionale Kommunikation dadurch zustande, indem der Client und der Server ihre Rollen tauschen. Es ist wichtig zu verstehen, dass die Rollen erst dann getauscht werden, wenn der Client einen Request mit dem *Upgrade-Header: PTTH/0.9* an den Server stellt. Der Server kann von sich allein keine Anfragen initiieren. D.h. sendet der Client über einen langen Zeitraum keinen Request, bekommt er eventuell vorliegende neue Daten nicht geliefert. In dieser Hinsicht ähnelt das Reverse HTTP dem

Piggyback-Verfahren. Der Übertragungskanal in die entgegengesetzte Richtung ist allerdings im Unterschied zum Piggyback viel beständiger und steht für sich allein (transportiert keine “fremden” Daten). Eine zusätzliche Erhöhung der Latenzzeit ergibt sich aus der Tatsache, dass die Rollen nicht sofort getauscht werden. Der Server sendet zuerst eine Beschtätigung, dass er bereit das HTTP-Socket als Client zu nutzen. Erst dann findet das Umschalten der Übertragungsrichtung statt.

Serverlast
Die client- bzw. serverseitige Last würden sich nicht von der Last einer herkömmlichen Web-Anwendung unterscheiden, weil der Client und der Server nur ihre Rollen tauschen und ansonsten keine speziellen Mechanismen für die bidirektionale Kommunikation nutzen.

Speicherauslastung
Die Speicherauslastung sollte sich auch nicht von der Auslastung einer herkömmlichen Web-Anwendung unterscheiden. Es sind dafür keine Messwerte bekannt.

Skalierbarkeit
Web-Anwendungen auf Basis vom Reverse HTTP sind gut skalierbar. Es können alle üblichen Lösungen für die Skalierung einer Web-Anwendung eingesetzt werden.

Firewalls und Proxy-Server Toleranz
Da nur die Richtung der Datenübertragung reversiert wird und es ansonsten keine Besonderheiten für HTTP Anfragen / Antworten gibt, würde das Problem mit Caching oder dem Abbruch der Verbindungen durch andere vorgeschaltete Server nicht existieren.

Datenvolumen
Dadurch, dass der Server zuerst eine Beschtätigung über die Bereitschaft für die Protokollumschaltung sendet (*HTTP/1.1 101 Switching Protocols*), ist das Datenvolumen ein wenig größer als bei normalen Web-Anwendungen. Das Problem der Senkung der zu übertragenden Datenmengen wird verstärkt im neuen WebSocket-Standard adressiert.

Zustellsicherheit der Daten und Datenverlust
Da die Latenzzeit hoch ausfallen kann, ist ein Datenverlust auch möglich. Das Reverse HTTP beschreibt nur noch das Kommunikarionsverfahren und kümmert sich nicht um die Datenrettung während Verbindungsausfällen. Nur wenige Technologien, wie das BOSH oder die GAE, haben dafür eine Lösung parat.

Datenaustauschformat
Der Vorteil dieser Technik besteht in der Flexibilität des Nachrichtenformats. Genauso wie beim Polling können sowohl textuelle als auch binäre Nachrichten ausgetauscht werden.

Umgang mit dem Verbindungslimit
Die Umkehrung der Datenübertragung bringt keine zusätzlichen Probleme zu dem Verbindungslimit als die schon bekannten aus der HTTP-Spezifikation. Das Reverse HTTP adressiert auch nicht das Problem der gleichzeitigen Arbeit in mehreren Browser-Tabs oder mit mehreren bidirektionalen Frameworks auf einer Seite, wie dies z.B. beim OpenAjax-

Push-Protokoll der Fall ist.

Implementierungsaufwand
Es gab bis jetzt keine Implementierungsvorschläge für das Reverse HTTP. Das Protokoll ist experimentell und wird bei keinem Client / Server unterstützt. Falls es unterstützt wäre, würde sich der Implementierungsaufwand von dem Aufwand bei der Implementierung einer herkömmlichen Web-Anwendung gar nicht unterscheiden.

Verbreitung und Unterstützung in Frameworks
Der Ansatz bleibt theoretisch und ist eher nur für die akademische Lehre interessant. Es gibt leider kein Framework, das diese Technik berücksichtigen könnte.

5.3.10 WebSocket

Latenzzeit und Performance
Die Verwendung von TCP für das WebSocket-Protokoll löst das aus `XHR`, Comet und anderen bidirektionalen Technologien bekannte Request/Response-Paradigma ab. Die Restriktionen hinsichtlich der Übertragungswege und Abläufe fallen beim WebSocket weg. Der Server kann Daten an den Client schicken, ohne das der Client sie explizit anfragt. Das bewirkt kürzere Latenz und mehr verfügbare Bandbreite. Die WebSocket-Technologie verfügt über die niedrigste Latenzzeit und die beste Performance aus allen hier beschriebenen Verfahren.

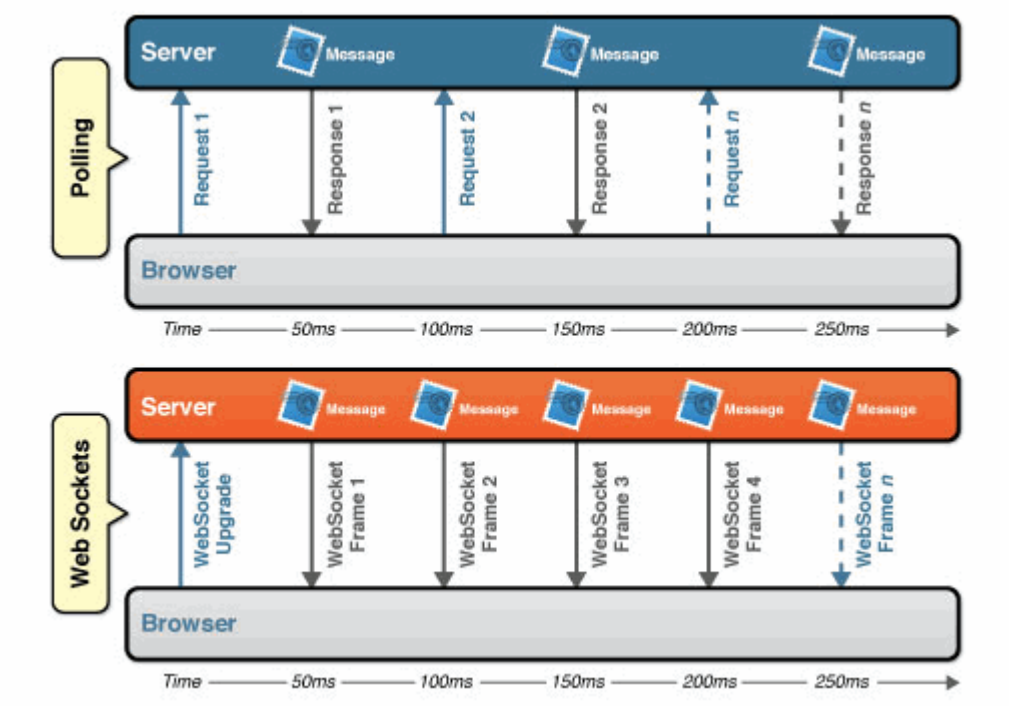

Abbildung 17: Latenz Polling vs. WebSocket

Zum Vergleich mit dem Polling nehmen wir an, dass jeder Request und jede Response jeweils 50 Millisekunden in Anspruch nehmen. Nachdem ein Request gesendet wurde, kann der Server innerhalb von 50 Millisekunden keine Nachrichten an den Browser senden. Wie

sieht es jetzt mit dem WebSocket aus? Nachdem die Verbindung zum WebSocket *upgraded* wurde, können Nachrichten vom Server zum Browser und zurück ohne neue Requests „fließen“. Dem obenstehenden Bild 17 [WSURL] kann man entnehmen, dass der Browser beim WebSocket fünf und beim Polling nur drei neue Nachrichten innerhalb von 250 Millisekunden bekommt.

Serverlast
Die WebSocket-Technologie bietet ein echtes Vollduplex auf einem einzigen TCP-Socket, während z.B. beim HTTP-Streaming zwei Kanäle belegt werden. Ein WebSocket-Server kann daher doppelt so viele gleichzeitige Verbindungen verwalten. Hinzu kommt noch, dass ein Browser beim WebSocket keine zusätzlichen AJAX-Anfragen stellen muss, um neue Daten empfangen zu können. Das reduziert die Serverlast enorm.

Speicherauslastung
Keine zusätzlichen AJAX-Anfragen und eine natürliche Unterstützung auf der Serverseite resultiert in einer geringen Speicherauslastung.

Skalierbarkeit
Manche Autoren bezeichnen das WebSocket als Quantensprung für die Skalierbarkeit im Web [OGG10, 263]. Web-Applikationen auf Basis vom WebSocket können mit halb so vielen Verbindungen als z.B. bei den Comet-Technologien auskommen. Das führt zu einer besseren Skalierbarkeit.

Firewalls und Proxy-Server Toleranz
Es gibt eine Studie, wie Firewalls und Proxy-Server auf eine WebSocket-Verbindung wirken [WPURL]. Das WebSocket-Protokoll kennt an sich keine Firewalls und Proxy-Server. Jede WebSocket-Kommunikation beginnt mit einem HTTP-kompatiblen Handshake, so dass HTTP-Server ihre HTTP und HTTPS Standard-Ports (80 und 443) mit einem WebSocket-Server teilen können. Manche Proxy-Server sind harmlos und arbeiten gut mit WebSockets. Andere verhindern eine reibungslose Kommunikation und verursachen das Fehlschlagen dauerhafter Verbindungen. In manchen Fällen ist eine Zusatzkonfiguration für Proxy-Server erforderlich oder ein Upgrade auf die Unterstützung der WebSocket-Spezifikation. Wenn ein Browser für die Benutzung eines expliziten Proxy-Servers konfiguriert ist, sendet er ein HTTP `CONNECT`-Kommando an den Proxy-Server. Darauf folgt ein Handshake-Verfahren und die bidirektionale Kommunikation kann unbehindert starten. Die Verbindung wird nicht unterbrochen. Wenn es keinen expliziten Proxy-Server gibt (d.h der Proxy ist *transparent* für den Browser), sendet der Browser kein `CONNECT`-Kommando und der Proxy ist ihm folglich nicht bewusst. Als Ergebnis wird eine laufende Verbindung mit großer Wahrscheinlichkeit fehlschlagen.

Datenvolumen
Jede Nachrichtenübertragung im WWW transportiert mit den eigentlichen Nutzdaten auch diverse Steuerinformationen. Beim HTTP werden z.B. solche Zusatzinformationen im HTTP-Header übertragen. Da der WebSocket-Standard direkt auf dem TCP/IP baut, ist der *Overhead* vom WebSocket im Vergleich zum HTTP gering. Das liegt an der Effizienz des neuen Übertragungsprotokolls. Pro Nachricht beträgt der *Overhead* genau zwei Byte, statt

mehrerer Hundert. Besonders deutlich wird der minimale *Overhead* bei Anwendungen, die eine große Anzahl aktiver Clients aufweisen. Die nachfolgenden Testszenarien und die Abbildung 18 [WSURL] zeigen, wie das Datenvolumen gering gehalten wird.

- **Szenario A**
  Polling: 1.000 Clients fragen zyklisch den Server jede Sekunde ab: Datendurchsatz ist 871 x 1.000 = 871.000 Bytes = 6.968.000 Bits pro Sekunde (6,6 Mbps).
  WebSocket: 1.000 Clients empfangen 1 Nachricht pro Sekunde: Datendurchsatz ist 2 x 1.000 = 2.000 Bytes = 16.000 Bits pro Sekunde (0,015 Mbps).
- **Szenario B**:
  Polling: 10.000 Clients fragen zyklisch den Server jede Sekunde ab: Datendurchsatz ist 871 x 10.000 = 8.710.000 Bytes = 69.680.000 Bits pro Sekunde (66 Mbps).
  WebSocket: 10.000 Clients empfangen 1 Nachricht pro Sekunde: Datendurchsatz ist 2 x 10.000 = 20.000 Bytes = 160.000 Bits pro Sekunde (0,153 Kbps).
- **Szenario C**:
  Polling: 100.000 Clients fragen zyklisch den Server jede Sekunde ab: Datendurchsatz ist 871 x 100.000 = 87.100.000 Bytes = 696.800.000 Bits pro Sekunde (665 Mbps).
  WebSocket: 100.000 Clients empfangen 1 Nachricht pro Sekunde: Datendurchsatz ist (2 x 100.000) = 200.000 Bytes = 1.600.000 Bits pro Sekunde (1.526 Kbps).

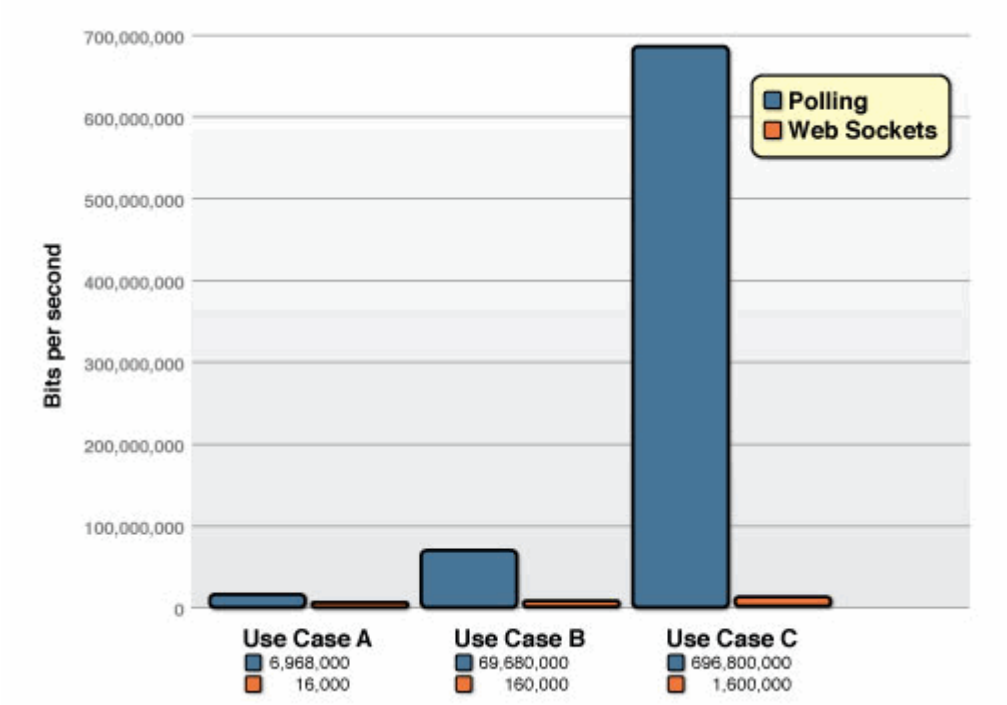

Abbildung 18: Datendurchsatz Polling vs. WebSocket

Zustellsicherheit der Daten und Datenverlust
Das WebSocket ist eine neue Technologie für die Echtzeitübertragung von Daten. Alle anderen Technologien können als nicht standardisierte Zwischenlösungen oder Hacks angesehen werden. Die verzögerungsfreie Kommunikation auf Basis vom WebSocket garantiert die höchste Zustellsicherheit der Daten ohne Datenverlust.

Datenaustauschformat
Beim WebSocket gibt es keine Bindung an bestimmte Datenformate und keine Vorgaben an Inhalte und Verarbeitung. Der Vorteil dieser Technologie ist es, dass sowohl textuelle als auch binäre Daten übertragen werden können.

Umgang mit dem Verbindungslimit
Das Problem mit dem Verbindungslimit existiert hier nicht. Mehrere WebSocket-Kommunikationen in einem Browser-Fenster werden über eine gemeinsame, schlanke und dauerhaft verfügbare TCP/IP-Verbindung abgewickelt. Das TCP/IP unterliegt nicht der gleichen Einschränkung wie das HTTP für die maximale Anzahl gleichzeitig geöffneter Verbindungen zu einem Host.

Implementierungsaufwand
Das WebSocket API bietet bereits alles, was man für die Implementierung braucht. Anhand dieser Schnittstelle lässt sich eine webbasierte Client-Server-Anwendung mit der echten bidirektionalen Kommunikation sehr schnell implementieren.

Verbreitung und Unterstützung in Frameworks
Folgende Browser bieten bereits eine native WebSocket-Unterstützung:

- Google Chrome ab der Version 4.0.249.0
- Apple Safari ab der Version 5.0.7533.16
- Mozilla Firefox ab der Version 3.7
- Opera ab der Version 10.7b
- Internet Explorer ab der Version 9

Auch mehrere Server, wie beispielsweise Jetty 6.0, GlassFish 3.1, jWebSocket, Node.js und Kaazing WebSocket Gateway unterstützen bereits den WebSocket-Standard. In älteren Browsern und Servern funktioniert das WebSocket nicht ohne zusätzliche Bibliotheken oder Plugins.

5.4 Zusammenfassung

Die wichtige Frage, welche Technologie oder welches Framework für die bidirektionale Kommunikation geeignet ist, hängt von der Client-Server-Landschaft und der Art der Software ab. Es unterliegt auch immer dem verfolgten Ziel, wann man zu dem Polling, einer der Comet-Varianten oder vielleicht einer fortgeschrittenen Technologie, wie das WebSocket, greifen sollte. Will man beispielsweise eine Web-Anwendung alle 10 Minuten auf den neuesten Stand bringen, ist es unnötig, über diesen Zeitraum eine fast ungenutzte Verbindung zum Server offen zu halten. Bei einem solchen Szenario ist das Polling auf jeden Fall die bessere Alternative. Das Polling ist auch leichter zu debuggen und zu testen, weil man sich auf festgelegte Intervalle verlassen kann. Web-Anwendungen, die auf Basis vom Polling-Verfahren implementiert werden, weisen auch eine gute Skalierbarkeit auf und bereiten keine Probleme mit Firewalls und Proxy-Servern. Das sind wenige Vorteile, die diese Technik hat.

Allerdings hängt hier die Entscheidung wieder davon ab, was für ein Server zur Verfügung

steht. Man sollte auf die richtige Serverumgebung achten. Ein schwacher Server würde sich sehr schnell über eine zu hohe Serverlast beschweren. Will man hingegen seinen Benutzern die Möglichkeit bieten, die gleichen Daten zur selben Zeit zu verändern und die Änderungen anderer Benutzer in Echtzeit mitzuverfolgen, dann ist das Comet HTTP Streaming wohl die bessere Alternative für eine performante bidirektionale Kommunikation. Doch es hat auch seine Schwäche, wie das Blockieren der Antwort durch Proxy-Server. Es ist auch nicht ganz praktikabel eine Verbindung langwierig offen zu halten. Es stellt sich zwangsläufig die Frage, wie lange die Verbindung offen bleiben kann. Die Antwort hängt von den beteiligten Ressourcen, der Serverauslastung, dem Datenvolumen usw. ab. Die Frage nach dem Auslöser, der die Verbindung beenden soll, muss auch beantwortet werden. Das kann ein *Timeout*, ein externes Ereignis oder der Client selbst sein.

OpenAjax-Push, Ajax-on-Demand und sogar Piggyback umgehen teilweise die genannten Schwachstellen von den Comet-Technologien. Sie werden nicht durch Firewalls und Proxy-Server blockiert, heben das Verbindungslimit auf und haben eine bessere Skalierbarkeit. Des Weiteren sind diese Techniken nicht so ressourcenaufwendig wie das Polling-Verfahren. Der Nachteil der drei genannten Technologien liegt vor allem in der Verbreitung - sie sind nicht verbreitet und finden kaum eine Anwendung in Frameworks. Auch die Latenzzeit ist im Allgemeinen hocher als bei den Comet-Techniken. Bei Piggyback kann zudem die Latenzzeit sehr hoch ausfallen und nicht akzeptabel sein.

Will man eine Chat-Anwendung oder eine andere *Instant Messaging* Software entwickeln, ist das BOSH-Protokoll zu empfehlen. Das BOSH wurde gerade für solche Arten der Kommunikation konzipiert. Es ist performant, zuverlässig, verursacht kein hohes Datenvolumen (Daten werden komprimiert) und garantiert die höchste Zustellsicherheit der Nachrichten bei Verbindungsabbrüchen. Bemängelt wird allerdings eine relativ hohe (serverseitige) Speicherauslastung und ein unter Umständen hoher Ressourcenverbrauch.

Für die Anwendungen, die auf der Goolge App Engine gehostet werden, gibt es nur eine einzige Alternative - Channel API. Das Channel API ist die proprietäre Technologie von *Google*, die sich über eine stabile Latenzzeit, eine gute Skalierbarkeit, minimierte Datenverluste und den geringen Implementierungsaufwand auszeichnet. Es ist durchaus möglich, kollaborative Web-Anwendungen, beispielsweise Whiteboards [CAURL], auf Basis dieser *Google's* Schnittstelle zu entwickeln. Der Nachteil dieser Technik neben der proprietären Schnittstelle besteht in der ungenügenden Datensicherheit.

All diese Nachteile scheinen durch den neuen WebSocket-Standard behoben zu sein. Das WebSocket ist eine echte bidirektionale Technologie, die Daten genau dann liefert, wenn sie verfügbar sind. Testergebnisse belegen, wie schnell Datenübertragungen unter Verwendung von WebSockets wirklich sind. Der neue Standard ist leichtgewichtiger als alle anderen Techniken, netzwerk- und ressourcenschonend. Außerdem ist das WebSocket eine zukunftsweisende, -sichere Technologie für kommende Web-Anwendungen und ist bereits in fast allen gängigen Browsern verfügbar.

Es sei noch zu erwähnen, dass es neben praktischen auch theoretische Ansätzen gibt - z.B. Reverse HTTP und JSONRequest (wird in dieser Arbeit nicht betrachtet). Sie sehen zwar

vielversprechend aus, haben sich aber nicht durchgesetzt und sind somit von weniger Bedeutung. Vermutlich verhindern technische Einschränkungen eine mögliche praktische Umsetzung. Die Idee der Umkehrung einer ausgehenden HTTP-Anfrage bei Reverse HTTP bleibt momentant utopisch.

Die Umsetzung der Beispielanwendung findet mit den drei gängigen Technologien statt – HTTP Long-Polling, HTTP Streaming, und WebSocket. Die oben gemachten Aussagen werden teilweise im praktischen Teil der Arbeit für diese Technologien nachgewiesen. Insbesondere wird die Latenzzeit gemessen und in Form von mehreren Säulen- und Liniendiagrammen visualisiert. Dieses Kapitel schließt sich mit einer tabellarischen Übersicht aller behandelten Technologien und deren Beurteilung bezüglich der erarbeiteten Vergleichskriterien. Die Übersichtstabelle sollte einen Leitfaden darstellen, wenn man die Benutzung bidirektionaler Technologien in Betracht zieht. Die Bewertung erfolgt nach dem (Schul)notenprinzip: “sehr gut – gut – mittel – schlecht – sehr schlecht” bzw. “sehr hoch – hoch – mittel – gering – sehr gering”.

	Latenzzeit	Serverlast	Speicher-auslastung	Skalierbarkeit	Proxy-Server Toleranz	Datenvolumen	Datenverlust	Datenformat	Verbindungs-limit	Implementie-rungsaufwand	Verbreitung
Polling	sehr hoch	sehr hoch	sehr hoch	sehr gut	sehr hoch	sehr hoch	hoch	beliebig	kein (nur bei mehreren offenen Tabs)	sehr gering	sehr hoch
Long-Polling	mittel	mittel	mittel	schlecht	hoch	mittel	mittel	meistens textuell	ja	mittel	sehr hoch
Streaming	gering	gering	gering	schlecht	sehr gering	gering	gering	textuell (JSON, XML)	ja	sehr hoch	sehr hoch
Piggyback	u.U. sehr hoch	sehr gering	gering	gut	sehr hoch	gering	hoch	textuell (JSON, XML)	kein	hoch	gering
OpenAjax-Push	mittel	mittel	mittel	gut	hoch	mittel	mittel	beliebig	kein	mittel	mittel
Ajax-on-Demand	hoch	hoch	mittel	sehr gut	hoch	hoch	mittel	beliebig	kein (nur bei mehreren offenen Tabs)	sehr gering	sehr gering
BOSH	gering	mittel	hoch	mittel	hoch	gering	gering	textuell (JSON, XML)	ja	mittel	hoch
GAE Channel API	gering	gering	gering	gut	hoch	gering	gering	textuell (JSON)	ja	gering	mittel
Reverse HTTP	hoch	sehr gering	gering	sehr gut	hoch	mittel	hoch	beliebig	kein (nur bei mehreren offenen Tabs)	unbekannt (vermutlich gering)	keine (theoretisch)
WebSocket	sehr gering	sehr gering	gering	gut	mittel	sehr gering	sehr gering	beliebig	kein	sehr gering	z.Z. mittel

6 Beispielanwendung „Whiteboard“

In der vorliegenden Diplomarbeit wird ein kollaboratives Online-Whiteboard mit den drei ausgewählten Server-Push-Technologien entwickelt und auf einem Web-Server deployet. Ein Whiteboard ist eine interaktive Tafel, auf der man zeichnen und schreiben kann. Es wird noch weiße Tafel genannt. Ein Whiteboard kann mehrfach verwendet werden, weil dessen Inhalte leicht gelöscht werden können. Ein Whiteboard wird kollaborativ, wenn mehrere Personen am Schreib- und Zeichenvorgang gleichzeitig teilnehmen können. Von kollaborativen Whiteboards können vor allem Lehrer und Studenten profitieren. Aufgrund seiner Interaktivität und Komplexität bietet sich das Whiteboard am besten an, um die Tauglichkeit verschiedener Kommunikationstechnologien nachzuweisen. Das Ziel dieses Kapitels ist eine konzeptionelle Überlegung und Ausarbeitung der erforderlichen Details für die praktische Umsetzung. Es wird auf Anforderungen, GUI-Gestaltung, Benutzeraktionen, Übertragungsformat und Gesamtarchitektur eingegangen.

6.1 Anforderungen an „Whiteboard“

Das Whiteboard soll folgenden Anforderungen genügen:

- Jeder kann ein Whiteboard erstellen und gemeinsam mit anderen Teilnehmern über einen herkömmlichen Web-Browser nutzen.
- Nur der Ersteller eines Whiteboards sieht einen Einladungslink, über den er andere Personen zum Whiteboard einladen kann.
- Eine eingeladene Person bekommt eine E-mail mit dem Link zum Whiteborad. Unter diesem Link verbirgt sich ein Formular, in dem der Whiteboard-Teilnehmer seinen Namen eintragen muss, um sich ans Whiteboard anzuschließen. Jedes Whiteboard ist somit über einen speziellen Link eindeutig identifizierbar.
- Das Whiteboard verfügt über eine Überschrift (Titel) und besitzt anpassbare Breite und Höhe. Die Breite und die Höhe bestimmen den Arbeitsbereich, in dem man Schreiben und Zeichnen kann.
- Es soll möglich sein, einen Text zu schreiben und die Formen (*Shapes*), wie Rechteck (*Rectangle*), Kreis (*Circle*), Ellipse (*Ellipse*), gerade Linie (*Straight Line*) und Freihandlinie (*Free Line*), zu zeichnen. Texte und Formen werden als Whiteboard-Elemente genannt.
- Es soll möglich sein, ein beliebiges Bild (*Image*) anhand der URL und ein vordefiniertes Symbol (*Icon*) hinzuzufügen. Bilder und Symbole werden ebenso als Whiteboard-Elemente bezeichnet.
- Auf jedem Whiteboard-Element können bestimmte Aktionen ausgeführt werden. Jedes Element kann markiert, dupliziert, in den Vordergrund / Hintergrund gerückt und gelöscht werden. Die entsprechenden Aktionen heißen *„Select Element“*, *„Clone Element“*, *„Bring to Front“*, *„Bring To Back“* und *„Remove Element“*.
- Das ganze Whiteboard (der Arbeitsbereich) kann gelöscht oder nachträglich in der Größe angepasst werden. Die dazugehörigen Aktionen heißen *„Clear Whiteboard“* bzw. *„Resize Whiteboard“*.
- Außer oben genannten Aktionen können alle Elemente mit der Maus per Drag-&-

Drop verschoben werden. Dazu soll eine Aktion *„Move Element“* zur Verfügung stehen.
- Whiteboard-Elemente und oben genannte Aktionen darauf werden in einem speziellen Werzeugkasten-Bereich zusammengefasst (*Toolbox*).
- Jedes Whiteboard-Element verfügt über die Eigenschaften (*Properties*). Es gibt einen Eigenschaften-Bereich zum Festlegen von Standard-Eigenschaften für neue Elemente oder zum Ändern von Eigenschaften eines bestehenden Elementes. Typische Eigenschaften eines Elementes sind beispielsweise seine Position, Größe, Farbe(n), Linienart, -stärke und Drehungswinkel. Auf die genauere Eigenschaften jedes Elementes wird weiter unten beim GUI-Design eingegangen.
- Alle Änderungen in dem Arbeitsbereich sollen in Echtzeit bei allen Teilnehmern sichtbar sein. Das Whiteboard ist kollaborativ und soll eine gemeinsame und benutzerfreundliche Nutzung ermöglichen.
- Es gibt einen Bereich für das „*Event-Monitoring*“. Das *Event-Monitoring* soll alle Aktivitäten an einem Whiteboard zur Laufzeit protokollieren. Es antwortet somit auf die Frage "Wer hat was und wann geändert?". Die Informationen werden in Echtzeit aktualisiert.
- Weitere Informationen, wie das Erstellungsdatum des Whiteboards und die Anzahl der aktiven Benutzer, die an dem Whiteboard gleichzeitig arbeiten, wären sicherlich hilfreich.
- Eine Funktionalität zum Anheften (*Pin*) bzw. Loslösen (*Unpin*) der Bereiche (*Panels*) wäre auch hilfreich. Losgelöste *Panels* lassen sich frei per Drag-&-Drop verschieben. Allerdings überlappen sich losgelöste *Panels* mit dem Arbeitsbereich selbst, falls der Browser-Fenster minimiert wird, da sie nicht in einem Raster fixiert sind.

Die folgenden Funktionalitäten werden nicht gefordert, weil sie eine geringe oder gar keine Bedeutung für diese Diplomarbeit haben: Undo- / Redo-Funktionalität, um vorgenommene Arbeitsschritte rückgängig zu machen bzw. nochmals ausführen zu lassen, Anpassungen von Whiteboard-Elementen in der Größe mit der Maus, Radiergummi, Export eines Whiteboards in Image-Formate, Gruppierung von Elementen, um Operationen auf einer Elementgruppe ausführen zu können, Zeichnen von Polygonen und Kurven, Chat-Fenster für die Kommunikation mit Whiteboard-Teilnehmern. Diese Anforderungen werden in der Beispielanwendung nicht berücksichtigt und nicht implementiert. Außerdem wird es ein Berechtigungskonzept und die Internationalisierung der Web-Anwendung fehlen. Die Sprache der Beispielanwendung ist auf Englisch begrenzt.

6.2 GUI-Design

Die Abbildung 19 zeigt den Aufbau des Whiteboards. Es ist in fünf Bereiche geteilt. Es gibt einen Bereich in der Mitte, in dem sich Whiteboard-Elemente befinden, einen Bereich für den Werkzeugkasten rechts oben, einen Bereich für die Elementeigenschaften, einen Bereich für das Event Monitoring ganz unten und einen oberen Bereich. Diese Aufteilung in fünf Bereiche ist nach ergonomischen Gesichtspunkten konzipiert und soll für eine intuitive Bedienoberfläche mit einheitlichem *Look-&-Feel* sorgen.

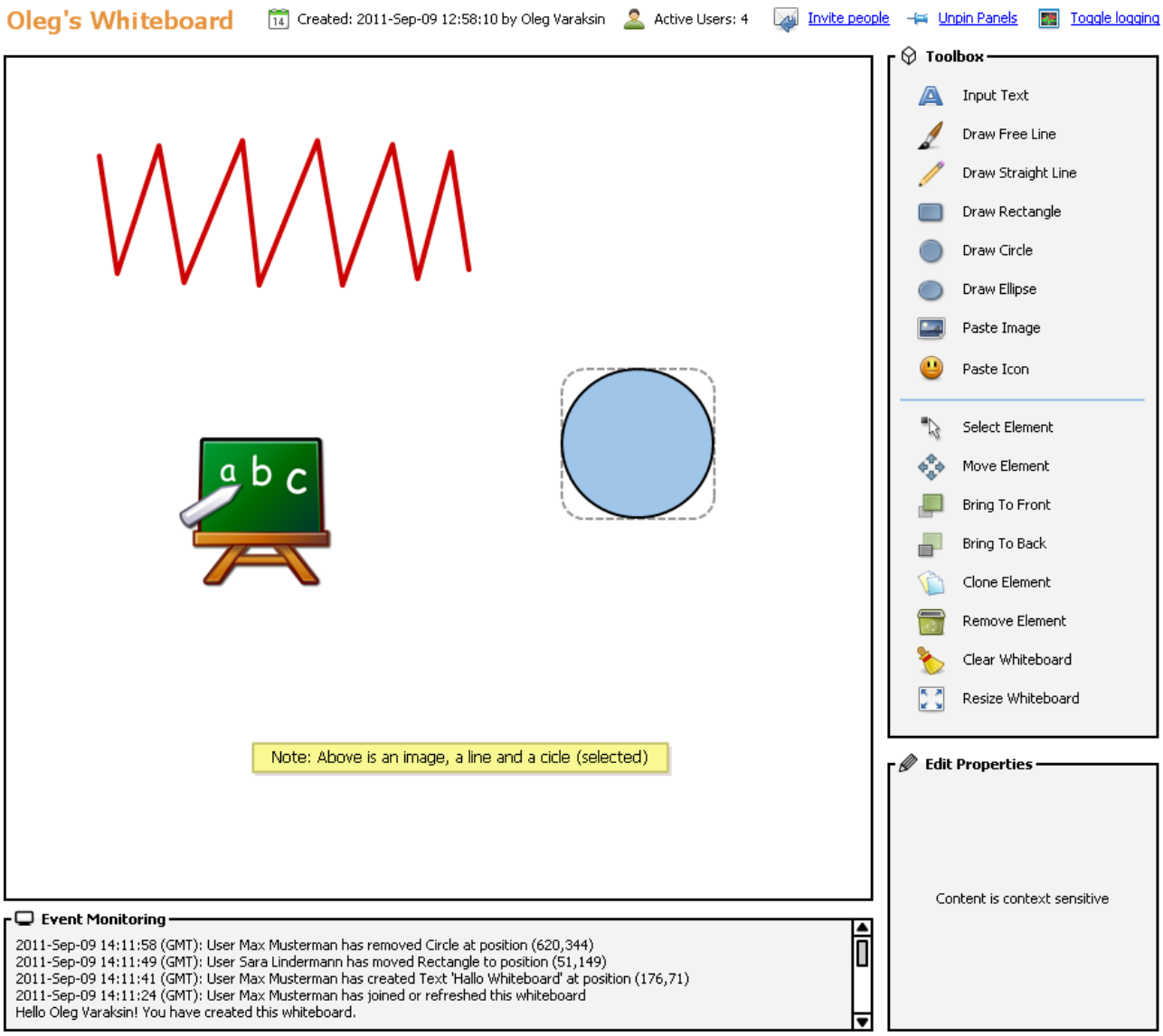

Abbildung 19: Whiteboard Arbeitsblatt

Im oberen Bereich sind den Whiteboard-Titel, den Einladungslink und die Informationen über das Whiteboard untergebracht. Außerdem gibt es dort auch einen Link „*Pin / Unpin panels*“ zum Anheften bzw. Loslösen der *Panels* und einen Link „*Toggle logging*“ für eine visuelle Ausgabe von eingehenden / ausgehenden Nachrichten und der Zeit zwischen dem Senden und dem Empfangen einer Nachricht. Der Bereich für die Elementeigenschaften ist kontextsensitiv und zeigt die Eigenschaften an, wenn ein Element mit dem Befehl „*Select Element*“ ausgewählt wird oder gerade gezeichnet wurde. In dem zweiten Fall werden sogenannte Standard-Eigenschaften für neue Elemente angezeigt. Das Festlegen der Standard-Eigenschaften wird weiter unten erläutert.

Um ein Element zeichnen zu können, wählt man einen Befehl aus dem oberen Bereich des Werkzeugkastens aus und klickt man anschließend irgendwo auf dem Arbeitsbereich. Die Befehle in dem Werkzeugkasten sind selbsterklärend. Dialoge sollten jedoch erläutert werden. Beim Klick auf „*Input Text*“ erscheint ein Dialog, in dem man einen Text eingeben

kann. Ein Klick auf „*Paste Image*“ führt zu einem Dialog, in dem man eine URL und die Bildgrößen eingeben kann und ein Klick auf „*Paste Icon*“ zu einem Dialog, in dem man ein vordefiniertes Symbol auswählen kann. Hinter dem Befehl „*Resize Whiteboard*“ verbirgt sich ein Dialog, in dem man neue Breite und Höhe des Arbeitsbereiches bestimmen kann. Sinnvoll wäre noch eine Sicherheitsabfrage für den Befehl „*Clear Whiteboard*" - „Do you really want to clear this whiteboard?".

Um ein neues Whiteboard anzulegen, sind notwendige Angaben zum Titel und zur Größe des Whiteboards, sowie zum Benutzernamen des Erstellers erforderlich. Die erste Seite, die erscheint, verlangt diese Angaben. Sie wird in der nächsten Abbildung 20 demonstriert.

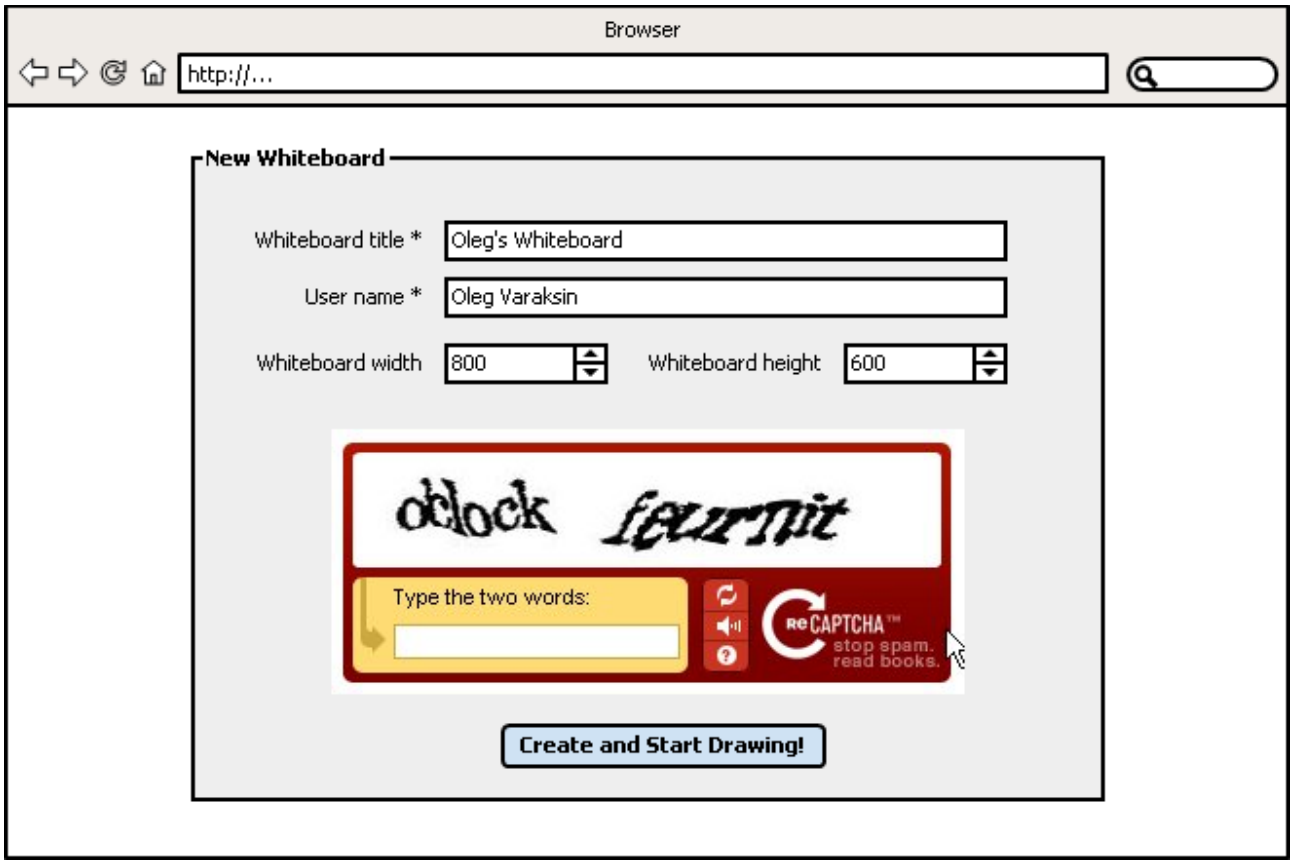

Abbildung 20: Neues Whiteboard anlegen

Eine CAPTCHA-Komponente sorgt dafür, dass die Eingaben nur durch Menschen und keine programmierten *Bots* gemacht werden können. Um zu einem bestehenden Whiteboard zu gelangen, ist nur noch den Benutzernamen erforderlich. Ein Formular dazu wird hier nicht designt, weil es trivial ist.

Elemente haben sowohl gemeinsame als auch spezifische, eigene Eigenschaften. Ein Text kann beispielsweise keine Eigenschaft „*Linienart*“ besitzen und ein Rechteck hat keine Eigenschaft „*Radius*“. Der Inhalt des Bereiches *„Edit Properties“* ändert sich deshalb in Abhängigkeit zum aktuell bearbeiteten Whiteboard-Element. Alle Eigenschaften werden in den nachfolgenden Abbildungen 21-27 gezeigt und sind größtenteils selbsterklärend. Allen Inhalten des Bereiches *„Edit Properties“* ist gemeinsam, dass sie den Button "Make as

default" enthalten. Mit Hilfe dieses Buttons können die Standard-Eigenschaften festgelegt werden. Neu erstellte Elemente bekommen die Standard-Eigenschaften automatisch zugewiesen und werden mit diesen Eigenschaften gerendert. Das ist eine nützliche Funktion, wenn man mehrere Elemente mit gleichen Eigenschaften zeichnen möchte.

Es bleibt noch zu klären, wie die Eingabewerte übernommen werden sollen. Die Eingabewerte sollten beim Verlassen von Eingabefeldern übernommen werden – wenn man irgendwo wegklickt oder die TAB-Taste druckt. HTML-mäßig werden die Eingabefelder mit dem *onchange callback* verknüpft. Gibt der Benutzer z.B. eine neue Rechteckbreite ein, werden die Änderungen erst beim Verlassen des Eingabefelds „*Width*“ übernommen. Wobei das *onchange* Ereignis nur dann ausgelöst wird, falls es wirklich eine Werteänderung gab. Daraufhin ändert sich die Breite des entsprechenden Rechtecks bei allen Whiteboard-Teilnehmern gleichzeitig. Wäre jede Änderung sofort beim Tippen übernommen, würde dies eine hohe Anzahl an ausgehenden HTTP Requests verursachen, was unter Umständen zu einem Serverausfall führen könnte. Gibt es keine Zuordnung einer Eigenschaft zu einem Eingabefeld, wird die Werteänderung einer solchen Eigenschaft sofort übernommen. Ein Beispiel dazu ist die Eigenschaft „*Color*“, die mittels eines speziellen Widgets *Color Picker* geändert wird.

Die Abbildung 21 modelliert die Eigenschaften eines Textes. Ein Text verfügt über Koordinaten, Schriftart, -größe, -stärke, -stil, Farbe und Drehungswinkel.

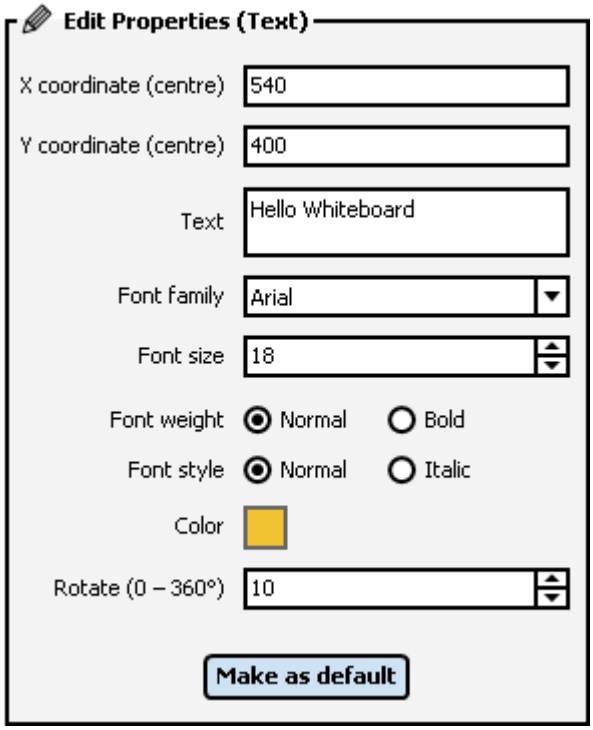

Abbildung 21: Text Eigenschaften

Die Abbildung 22 modelliert die Eigenschaften einer Linie. Die Linieneigenschaften sind Farbe, Breite, Stil, Deckkraft der Farbe und Drehungswinkel. Eine Linie hat keine Koordinaten.

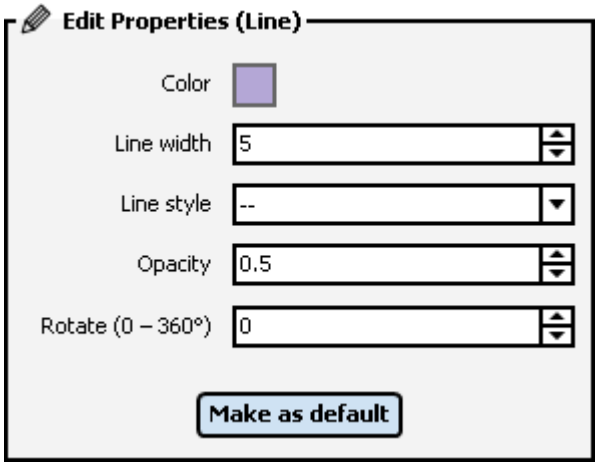

Abbildung 22: Linie Eigenschaften

Die Abbildung 23 zeigt die Rechteckeigenschaften. Ein Rechteck verfügt über Koordinaten, Breite, Höhe, Rundungsgrad für Ecken, Farben für Hintergrund und Randlinie, Randbreite, -stil, Deckkraft für Hintergrund- und Randlinienfarbe, Drehungswinkel.

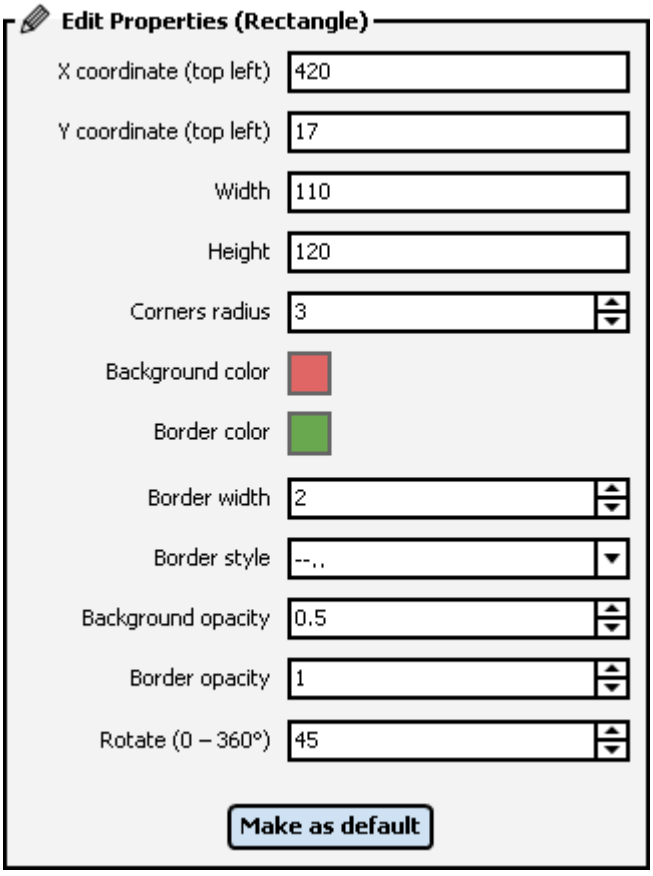

Abbildung 23: Rechteck Eigenschaften

Die Eigenschaften eines Kreises sind größtenteils den Rechteckeigenschaften ähnlich. Im Vergleich zum Rechteck gibt es keine Breite, Höhe und Rundungsgrad für Ecken. Stattdessen gibt es einen Radius.

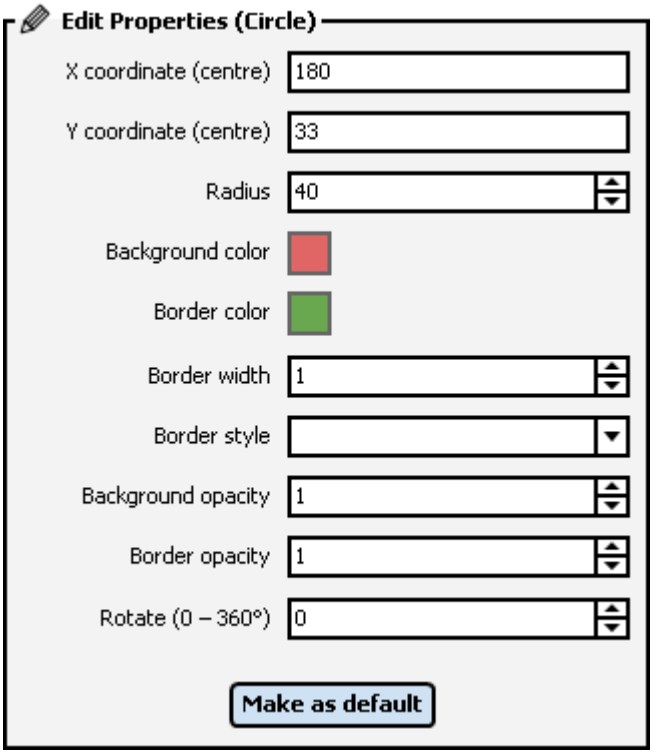

Abbildung 24: Kreis Eigenschaften

Ellipseneigenschaften sind wie beim Kreis, es gibt nur zwei Radien: horizontal und vertikal.

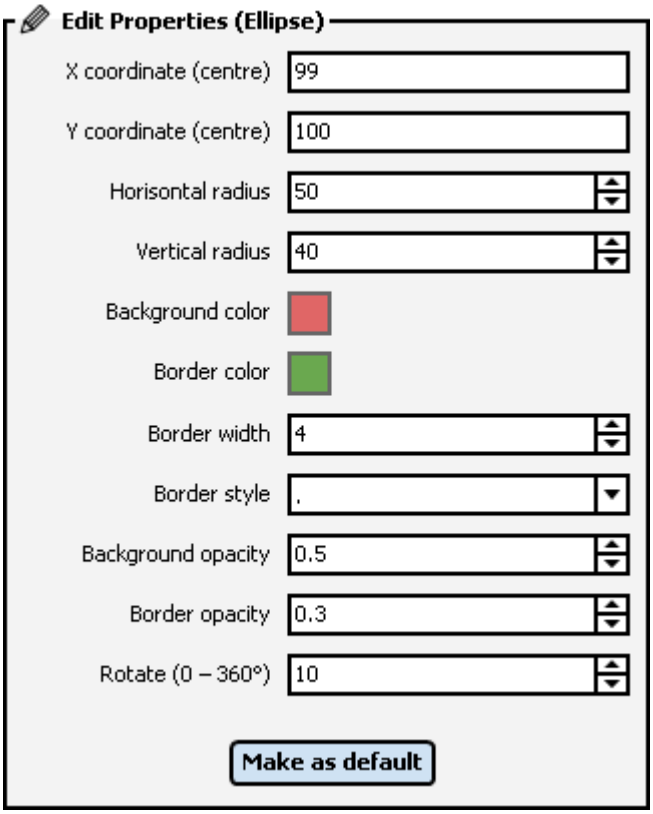

Abbildung 25: Ellipse Eigenschaften

Die Eigenschaften von Bildern sind Koordinaten, Breite, Höhe und Drehungswinkel. Die URL eines Bildes sowie dessen Größen werden in einem speziellen Dialog eingegeben. Der Dialog wird erscheinen, wenn man auf „Paste Image“ klickt. Die URL eines Bildes kann nachträglich nicht geändert werden.

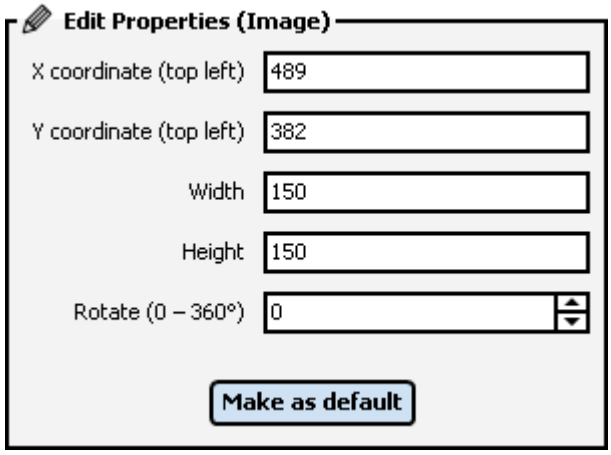

Abbildung 26: Bild Eigenschaften

Bei Symbolen sind die folgenden Eigenschaften spezifiziert: Koordinaten, Drehungswinkel und Skalierungsfaktor. Anhand des Skalierungsfaktors lassen sich Symbole vergrößern bzw. verkleinern. Alle Symbole sind vorgegeben und können aus einem speziellen Dialog ausgewählt werden.

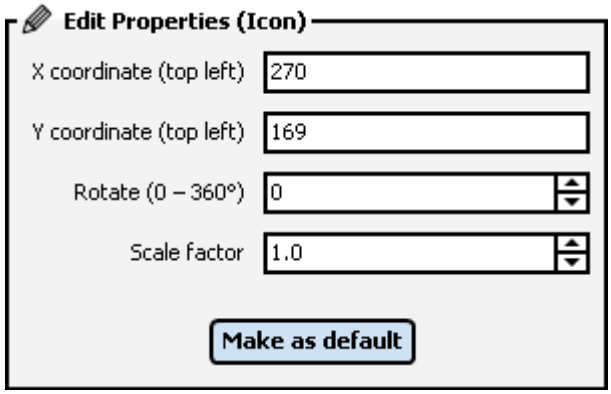

Abbildung 27: Symbol Eigenschaften

Hinter dem Einladungslink „*Invite people*“ verbirgt sich ein Dialog mit zwei Auswahlmöglichkeiten. Dies wird in der Abbildung 28 demonstriert. Ein Klick auf „Click to open your email client“ startet ein E-mail-Programm auf dem lokalen Computer und generiert eine E-mail mit dem Link zum Whiteboard. Der Link zum Whiteboard kann auch direkt per *Copy-&-Paste* verschickt werden.

Abbildung 28: Personen zum Whiteboard einladen

6.3 Aktionen und Übertragungsformat

Dieses Unterkapitel beschäftigt sich mit der Beschreibung der Whiteboard-Aktionen und dem Übertragungsformat. Es folgt eine vollständige Auflistung aller Aktionen, die sich durch Anwendungsfälle definieren lassen. Insgesamt gibt es zehn Aktionen.

1. „Join“. Diese Aktion wird ausgelöst, wenn sich ein Benutzer dem Whiteboard anschließt, zu dem er eingeladen wurde. Alle anderen Benutzer, die mit dem Whiteboard arbeiten, werden darüber informiert.
2. „Create“. Diese Aktion wird ausgelöst, wenn ein Element erzeugt wird.
3. „Update“. Diese Aktion wird bei Eigenschaftsänderungen eines Elementes ausgelöst. Gemeint wird damit eine direkte Änderung im *Panel* „*Edit Properties*“.
4. „Remove“. Diese Aktion wird beim Löschen eines Elementes ausgelöst.
5. „Clone“. Diese Aktion wird beim Duplizieren eines Elementes ausgelöst.
6. „Move“. Diese Aktion wird ausgelöst, wenn ein Element verschoben wird.
7. „ToFront“. Diese Aktion wird ausgelöst, wenn ein Element in den Vordergrund gebracht wird. Das Element wird danach über allen Elementen liegen.
8. „ToBack“. Diese Aktion wird ausgelöst, wenn ein Element in den Hintergrund gebracht wird. Das Element wird danach hinter allen Elementen liegen.
9. „Clear“. Diese Aktion wird ausgelöst, wenn alle vorhandenen Elemente auf einmal gelöscht werden.
10. „Resize“. Diese Aktion wird beim Ändern der Whiteboard-Größe ausgelöst.

Jede Aktion ist mit bestimmten Daten verknüpft, welche an den Server übertragen, dort aufbereitet und anschließend an alle beteiligten Whiteboard-Benutzer verbreitet werden (*broadcasting*). Da die Daten noch serverseitig transformiert werden müssen, unterscheidet sich die Datenstruktur der ausgehenden Nachrichten (aus Sicht des Benutzers) von der

Datenstruktur der eingehenden. Für die Datenstruktur der ausgehenden Nachrichten bieten sich folgende sinnvolle Daten an:

- Name der Aktion (*action*).
- Zeitstempel einer Änderung (*timestamp*).
- Benutzer, der diese Änderung ausgelöst hat (*user*).
- Whiteboard-Id, die das Whiteboard identifiziert (*unique Id* oder *uuid* genannt).
- Element mit seinen Eigenschaften (*properties*), welche aktualisiert werden müssen. Diese Information ist optional, da nicht jede Aktion zu einer Element-Aktualisierung führt. Jedes Element verfügt neben den Eigenschaften über ein obligatorisches Id-Attribut (*uuid*) zur eindeutigen Identifizierung.
- Mit den Elementeigenschaften sollte auch der Typ des Elementes bekannt sein. Der Typ ist eine Elementbezeichnung, wie beispielsweise „Text“ oder „Circle“. Diese Information ist technisch bedingt und wird bei der Serialisierung / Deserialisierung von JavaScript / Java Objekten benötigt, weil aus Eigenschaften allein kein Java-Klassenname hervorgeht.
- Parameter mit Zusatzinformationen (*parameters*), wie beispielsweise neue Größen des Whiteboards. Diese Daten sind ebenso optional.

Für die Datenstruktur der eingehenden Nachrichten bieten sich folgende Daten an:

- Name der Aktion (*action*).
- Zeitstempel der Änderung, die zur Aktualisierung geführt hat (*timestamp*).
- Nachricht für das *Event-Monitoring* (*message*).
- Element mit denjenigen Eigenschaften (*properties*), welche zu aktualisieren sind. Falls eine Eigenschaft für die Aktualisierung nicht relevant ist, sollte sie hier nicht aufgelistet werden. Das würde die zu übertragende Datenmenge reduzieren. Ein Beispiel dafür ist das Verschieben eines Elementes per Drag-&-Drop. Bei diesem Szenario sind nur noch neue Koordinaten zu übertragen. Die Information über die zu aktualisierenden Eigenschaften ist optional.
- Elementtyp sollte hier auch aus technischen Gründen bekannt sein.
- Parameter mit Zusatzinformationen (*parameters*). Diese Daten sind ebenso optional.

Das Datenformat der einzelnen Bestandteilen der zu übertragenden Information muss noch spezifiziert werden. Sowohl clientseitig als auch serverseitig existieren Whiteboard-Modelle. Serverseitiges Modell wird mit jeder clientseitigen Änderung aktualisiert. Schließt sich ein neuer Benutzer einem Whitboard an, wird das vorhandene serverseitige Modell in das clientseitige übertragen und transformiert. Anschließend findet das *Rendering* des Whiteboards im Browser statt. Damit aufwendige Daten-Transformation zwischen Client und Server größtenteils entfallen, kommt das schlanke JSON-Format zum Tragen. JSON (*JavaScript Object Notation*) ist ein kompaktes Datenformat für den Datenaustausch zwischen Anwendungen in einfacher lesbarer Textform. Jedes JSON-Dokument ist zugleich ein JavaScript-Objekt in dessen literaler Notation und wird deshalb sofort in JavaScript intergriert und interpretiert. Ein weiterer Vorteil von JSON ist seine Unabhängigkeit von der Programmiersprache. Das Übertragungsformat der Beispielanwendung basiert komplett auf der JSON-Notation.

An dieser Stelle muss man zuerst einen kleinen Exkurs in die JSON Welt geben. Das JSON-Format wurde durch Douglas Crockford spezifiziert (RFC 4627). Es basiert auf einer Untermenge der Programmiersprache JavaScript und unterstützt folgende Datentypen: `Number`, `String`, `Boolean`, `Array`, `Object` und `null`. Das folgende Beispiel zeigt exemplarisch die JSON-Repräsentation eines Objektes, das eine Person beschreibt. Das Objekt beinhaltet Name, Vorname, Adresse und Telefonnummern einer Person. Die Telefonnummern sind wiederum als Array aufgefasst.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": 10021
  },
  "phoneNumbers": [
    "212 732-1234",
    "646 123-4567"
  ]
}
```

Man sieht, dass der Datentyp `Object` eine kommaseparierte Liste von Schlüssel-Wert-Paaren in geschweiften Klammern darstellt.

Es existieren viele Bibliotheken, die den Umgang mit JSON erleichtern. Verfügbare Java- / JavaScript-Bibliotheken sind auf der JSON-Homepage aufgelistet [JNURL]. Auf der Client-Seite stellt der Umgang mit JSON kein großes Problem dar. Für die Serverseite existiert eine sehr gelungene Bibliothek namens *Gson* [GNURL] - eine performante und flexible Bibliothek, die eine höhere Abstraktion im Vergleich zu den anderen Bibliotheken bietet. *Gson* wird bei der Implementierung der Whiteboard-Anwendung benutzt. Ein typisches Beispiel für die Serialisierung sieht in der Programmiersprache Java folgendermaßen aus:

```
Gson gson = new Gson();
int[] intArray = {1, 2, 3};
gson.toJson(intArray);
```

Dies führt zur Ausgabe

```
[1,2,3]
```

Der `String [1,2,3]` kann wiederum deserialisiert werden:

```
int[] intArray = gson.fromJson("[1,2,3]", int[].class);
```

Die obige Ausgabe `[1, 2, 3]` ist ein gültiges JavaScript-Objekt in der literalen Notation. Mit der JavaScript-Funktion `eval()` kann daraus ein ganz normales JavaScript-Objekt erzeugt werden.

```
var myArray = eval('[1, 2, 3]');
```

Mit `myArray[0]` kann somit auf `1`, mit `myArray[1]` auf `2` usw. zugegriffen werden. Noch besser wäre die Benutzung eines JSON-Parsers oder der verfügbaren in modernen Browsern Methoden `JSON.parse` und `JSON.stringify`, weil die `eval()` Funktion keine Validierung gegen ungültige Zeichenketten durchführt.

```
var myObject = JSON.parse(myJSONText);
```

Eine umgekehrte Transformation mit der Funktion `stringify`:

```
var myJSONText = JSON.stringify(myObject);
```

`myJSONText` kann dabei an den Server übertragen und dort wiederum in ein Java-Objekt transformiert werden. Nähere Informationen zum JSON-Format können der offiziellen Referenz [JNURL] entnommen werden.

Mit den oben spezifizierten Aktionsdaten lassen sich aktionsspezifische JSON-Strukturen bilden. Das nachfolgende Beispiel dient lediglich der Veranschaulichung der ausgetauschten Nachrichten im JSON-Format. Grobe Struktur der gesendeten Daten:

```
{"action": "create",
  "element": {
    "properties": {
      "uuid": "a311517b-d8d3-43c7-be13-46a26b94dda1"
      "x": 253,
      "y": 129,
      "color": "#000000",
      "fontFamily": "Verdana",
      ... weitere Eigenschaften ...
    },
    "type": "Text"
  },
  "timestamp" : 1315773850698,
  "user": "Alex",
  "whiteboardId": "8b10da44-30bd-4872-9961-8d821dcd4daf",
  "parameters": { ... }
}
```

Grobe Struktur der empfangenen Daten:

```
{"action": "create",
  "element": {
    "properties": {
      "uuid": "a311517b-d8d3-43c7-be13-46a26b94dda1"
      "x": 253,
      "y": 129,
      "color": "#000000",
```

```
      "fontFamily": "Verdana",
      ... weitere Eigenschaften ...
    },
    "type": "Text"
  },
  "timestamp": 1315773850698,
  "message": "2011-Sep-11 22:44:10: User Alex has created
              Text '...' at position (253,129)",
  "parameters": { ... }
}
```

Konkrete JSON-Strukturen werden im Anhang A „Aktionsspezifische JSON-Strukturen“ aufgelistet. Dort wird für jede Aktion zuerst eine ausgehende und dann eine eingehende Nachricht gezeigt.

6.4 Gesamtarchitektur

Ein wichtiger Punkt für die Spezifikation sind die Vorüberlegungen zur Architektur. Da es sich um eine Client-Server-Umgebung handelt, gibt es einen Client (Browser) und einen Server (Applikationsserver). HTML, JavaScript und CSS (*Cascading Style Sheets*) sind die Standardmittel bei der Entwicklung von Anwendungen, die von einem Web-Browser ausgeführt werden. Auf der Clientseite kommen zudem zwei JavaScript-Bibliotheken JQuery [JQURL] und Raphaël [RLURL] zum Einsatz, auf die im Kapitel 7 ausführlich eingegangen wird. Als Web-Framework wurde JavaServer Faces (JSF) ausgewählt. JavaServer Faces ist ein Standard-Framework bei der Entwicklung von Web-Anwendungen in einer Java EE-Umgebung [BUR10], [JFURL]. Das JSF-Frameworks integriert sich nahtlos in jeden Java EE-konformen Applikationsserver. JSF trennt die einzelnen Schichten einer Web-Anwendung und arbeitet nach dem MVC-Prinzip (*Model-View-Controller*). View ist für die Präsentationsschicht zuständig, Model holt Anwendungsdaten aus diversen Backend-Quellen und Controller nimmt Client-Anfragen entgegen und agiert als ein Bindungsglied zwischen View und Model.

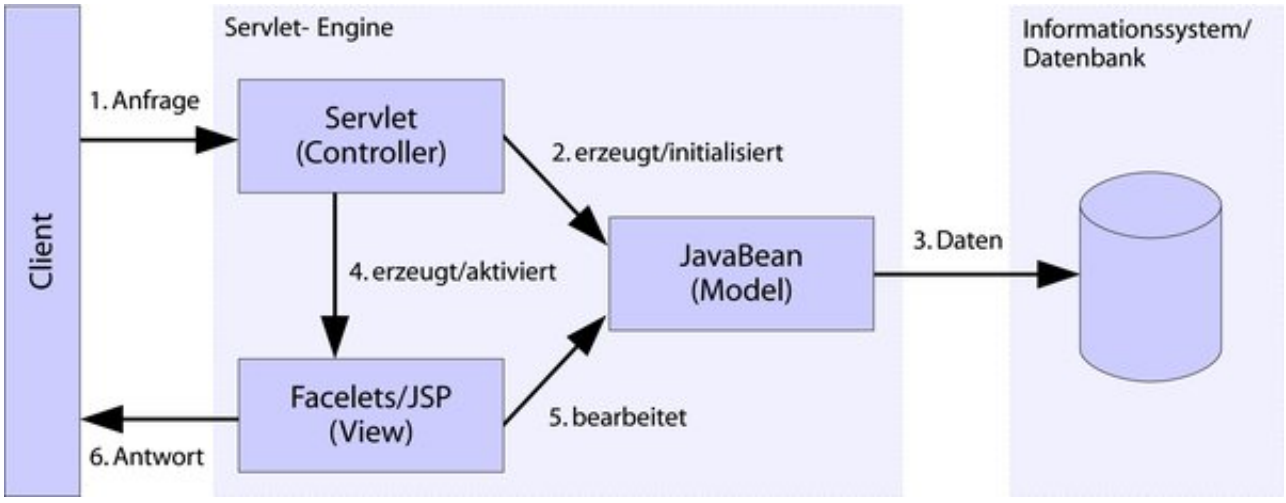

Abbildung 29: Architektur nach dem MVC-Prinzip

Durch diesen klaren Schnitt zwischen den Bereichen Modell, Ansicht und Steuerungslogik existiert keine hochkomplexe Mischung aus HTML-Tags und Java-Code, wie dies früher bei der JSP-Technologie der Fall war. Weitere Vorteile von JSF sind wiederverwendbare visuelle UI-Komponenten, ausgereifte Ereignisbehandlung, automatische Validierung und Konvertierung von Eingabewerten, gute Erweiterbarkeit und vieles mehr. Für das JSF-Framework existieren zahlreiche Komponenten-Bibliotheken. Eine davon, die in der Beispielanwendung verwendet wird, ist PrimeFaces [PFURL]. PrimeFaces ist eine sich rasch entwickelnde Bibliothek mit mehr als hundert reichhaltigen und visuell attraktiven JSF-Komponenten.

JSF-Framework an sich bietet keine bidirektionale Kommunikation. Für diesen Zweck wird in der Diplomarbeit das Atmosphere-Framework [ATURL] verwendet. Dieses Framework wird im Kapitel 7 (Unterkapitel 7.2) mit allen weiteren gängigen Frameworks und Engines behandelt. Die getroffene Wahl wird dort auch begründet. In der Beispielanwendung wird das Atmosphere-Framework sowohl clientseitig als auch serverseitig verwendet. Clientseitig stellt es ein jQuery-Plugin zur Verfügung. Serverseitig gibt es komplette Softwareschichten für fast jeden Applikationsserver. Um Java-Objekte nach bzw. von JSON zu konvertieren, wird die bereits vorgestellte *Gson*-Bibliothek [GNURL] eingesetzt. Der folgende Überblick über die Gesamtarchitektur fasst das Gesagte zusammen.

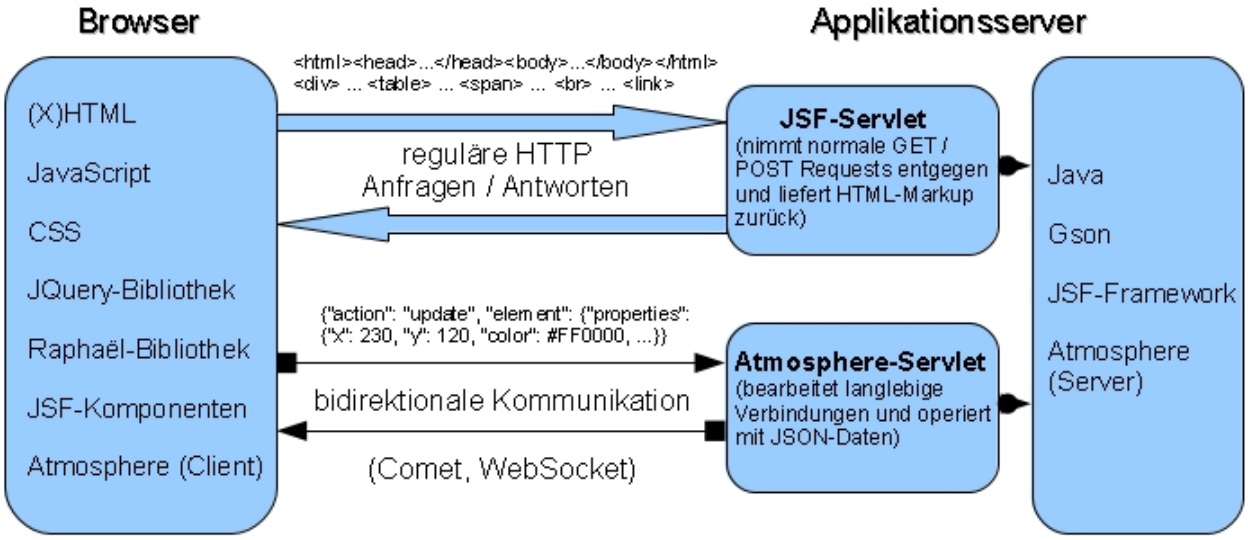

Abbildung 30: Gesamtarchitektur der Beispielanwendung

Anhand dieses Bildes kann man erkennen, dass jede HTTP-Anfrage durch einen der zwei Servlets läuft. Normale HTTP-Anfragen (Aktualisierungen der ganzen Seite oder nur Teilen davon) gehen durch den `JSF-Servlet`. HTTP-Antworten enthalten ein gewöhnliches HTML-Makup, für dessen Generierung das JSF-Framework zuständig ist. Ein Beispiel dafür wäre das Anheften bzw. das Loslösen der modellierten Bereiche. HTTP-Anfragen für die eigentlichen Whiteboard-Aktualisierungen werden durch die beschriebenen Aktionen initiiert. Sie liegen im JSON-Format vor und gehen durch den `Atmosphere-Servlet`. Der `Atmosphere-Servlet` ist für die bidirektionale Kommunikation zuständig. Der Servlet bearbeitet eingehende Requests, leitet entsprechende Aufgaben an den serverseitigen

Teil des Atmosphere-Frameworks weiter und sendet Responses an beteiligte Clients zurück.

Die Whiteboard-Applikation benötigt ein serverseitiges Modell. Es gibt acht grundlegende Elemente: `Text`, `Free Line`, `Straight Line`, `Rectangle`, `Circle`, `Ellipse`, `Image`, `Icon`. Jedes Element wird durch eine eigene Klasse repräsentiert. Außerdem scheint es sinnvoll zu sein, Basis-Klassen zu erstellen, da es auch gemeinsame Attribute gibt. Jedes Element hat z.B. eine „Id-Nummer“, die das Auffinden eines Elementes ermöglicht. Jedes Element kann auch rotiert werden und verfügt somit über ein Attribut „Drehungswinkel“. Solche Attribute können in den abstrakten Basis-Klassen `AbstractElement` und `Rotatable` gekapselt werden. Alle Elemente außer `Free Line` und `Straight Line` besitzen Koordinaten. Aus diesem Grund bietet sich an, die Koordinaten `X, Y` in eine spezielle Basis-Klasse `Positionable` auszulagern. Für `Free Line` und `Straight Line` bietet sich eine gemeinsame Basis-Klasse `Line`, weil sich deren Attribute gar nicht unterscheiden. Die Klasse `Line` erbt ihrerseits von den abstrakten Klassen `AbstractElement` und `Rotateable`. Eine genauere Klassenhierarchie wird im Kapitel 7 in Form eines Klassendiagramms aufgezeigt.

An dieser Stelle sei noch anzumerken, dass erstellte Whiteboards der Definition nach applikationsweit sind und liegen im sogenannten „*Application Scope*“ (andere *Scopes* sind „*Request Scope*“ und „*Session Scope*“), damit mehrere Benutzer die gleichen Whiteboards gleichzeitig nutzen können. Für einen besseren Speicherverbrauch, sollten daher alle Whiteboards über ein *Timeout* verfügen. Die *Timeout*-Zeit kann in der Konfiguration der Web-Anwendung eingestellt werden. Finden keine Benutzerzugriffe auf ein Whiteboard innerhalb der eingestellten Zeit statt, wird es aus dem Speicher entfernt.

7 Umsetzung der Beispielanwendung

Bevor zum praktischen Teil übergegangen wird, sind zuerst technische Anforderungen an den Webserver zu klären, welche das *Deployment* (Installation von Software auf einem Server) der Beispielanwendung ermöglichen sollen. Danach folgt eine Auflistung der meist bekannten und verbreiteten Frameworks und Engines, die zur Auswahl stehen. Sie werden stichwortartig beschrieben. Frameworks und Engines sind Softwareprodukte, die bei der Entwicklung helfen, da sie einen hohen Abstraktionsgrad von der zugrundeliegenden Technologie / Plattform bieten. Die Auswahl eines konkreten Frameworks für die praktische Umsetzung wird ausführlich begründet. In den letzten Unterkapiteln werden sowohl client- als auch serverseitige Bestandteile der Beispielanwendung entwickelt. Die eigentliche bidirektionale Kommunikation wird schrittweise mit dem zuvor ausgewählten Framework implementiert.

7.1 Technische Anforderungen an den Webserver

Webserver wurden bisher so strukturiert, dass sie einem klaren Request-Response-Ablauf folgten. Die HTTP 1.0 Spezifikation beschrieb das sogenannte „*thread pro request*“ Konzept, an dem alle Webserver eine Zeit lang hielten. Das Konzept besagt, dass mit jedem HTTP-Request eine TCP/IP-Verbindung zwischen Client und Server aufgebaut wird. An diese Verbindung bindet der Server einen *Thread*, der die HTTP-Response an den Client verschickt. Anschließend wird die TCP/IP-Verbindung beendet und der *Thread* wird freigegeben. Dabei wird normalerweise ein Pool von *Threads* erstellt, auf den dann eintreffende Anfragen verteilt werden. Wird ein *Thread* blockiert, kann eine weitere Anfrage in einem anderen *Thread* aus dem Pool von der CPU abgearbeitet werden. Dadurch, dass Anfragen in dem klassischen Request-Response-Ablauf nur sehr kurz dauern, ist es durchaus möglich, mit wesentlich weniger *Threads* als Anzahl der Benutzer auszukommen. Denn nach der Abarbeitung der laufenden Anfrage steht der *Thread* wieder als freie Ressource für eine neue Anfrage zur Verfügung.

Mit der Zeit wird die Gestaltung der Webseiten anspruchsvoller. Ressourcen müssen nun in Echtzeit nachgeladen werden. Es entstehen diverse Server-Push Ansätze, die durch die neue Kommunikation anspruchsvolle Anforderungen an den Webserver stellen, von welchem die Ereignisse übertragen werden sollen. Eine HTTP-Anfrage wird nun für das Nachladen von Ressourcen über einen sehr langen Zeitraum offen gehalten, was dem klassischen Webserverkonzept widerspricht. Die Wiederverwendung einer TCP/IP-Verbindung ist im HTTP 1.1 mit dem *keep alive* Header möglich, so dass Client und Server sich jetzt darüber einigen können, jede beliebige Verbindung bestehen zu lassen. Mit Java hat dies zur Folge, dass nun jede offen gehaltene TCP/IP-Verbindung einen *Thread* in Java Virtual Machine (JVM) belegt, was als „*thread per connection“* bezeichnet wird. Der *Thread* wird nicht direkt wieder freigegeben, sondern ist für eine längere Zeit besetzt. Die Verbindung blockiert kontinuierlich den assoziierten JVM *Thread*, obwohl dieser die meiste Zeit schläft. Dies führt dazu, dass gleichzeitig sehr viele *Threads* benötigt werden, obwohl sie die meiste Zeit nichts tun und nur auf ein Ereignis warten.

Wie verhält sich nun ein Server, wenn mehrere Hundert Benutzer mit jeweils einer, teilweise

zwei und mehr HTTP-Verbindungen arbeiten? Viele der heutzutage eingesetzten Server kommen bei solchen Anforderungen schnell ins Schwanken, weil die Speicher- und CPU-Auslastung mit kontinuierlich blockierten JVM *Threads* rapide steigen. Abhilfe schafft hier das *new non-blocking input/output* API, kurz NIO API [NIURL], das in der Java 1.4 eingeführt wurde. Das NIO API wurde dafür designet, um den Zugriff auf Schreib-Lese-Operationen des zugrundeliegenden Betriebssystems (*low-level input/output operations*) bereitzustellen. Es zielte darauf ab, dass das Warten auf Ereignisse des IO-Sockets eine Angelegenheit des Betriebssystems ist. JVM *Threads* müssen dann nicht mehr warten und somit geblockt werden - sie arbeiten ausschließlich tatsächlich eingetretene Ereignisse ab. Um das zu erreichten, werden offen gehaltene Verbindungen bis zum Publizieren serverseitiger Ereignissen in einen Ruhezustand versetzt und die *Threads* in den *Thread-Pool* des Webservers gebracht. Soll ein serverseitiges Ereignis an den Client ausgeliefert werden, wird ein *Thread* aus dem *Thread-Pool* des Webservers ausgewählt und der sich im Ruhezustand befindenden Verbindung zugeteilt. Anschließend kann das Ereignis an den Client ausgeliefert werden. Danach wird die Verbindung wieder in den Ruhezustand versetzt und der *Thread* kehrt erneut in den *Thread-Pool* des Webservers zurück. Als Ergebnis lassen sich mit der gleichen Anzahl von *Threads* mehr HTTP-Anfragen parallel abarbeiten.

Die beschriebene Vorgehensweise wurde im Rahmen der Java Servlet 3.0 Spezifikation (JSR 315) mit „*asynchronous request processing*" (ARP) standardisiert [JSURL]. Nicht alle Applikationsserver sind jedoch konform zu der neuen Servlet 3.0 Spezifikation. Daraus resultiert, dass viele Web-Anwendungen noch in relativ alten *Servlet-Containers* laufen und bis dahin auf proprietäre Lösungen angewiesen sind. All diese Lösungen ermöglichen den laufenden Request von einem Thread zu trennen, beiseitezulegen und erst bei einem anstehenden *Update* wieder aufzunehmen. Sie rüsten Applikationsserver mit einer ausgereiften ARP-Unterstützung für die asynchrone Anfrageverarbeitung aus. Der Nachteil dieser Lösungen ist es, dass sie serverspezifisch sind. Verschiedene Frameworks, wie Atmosphere [ATURL], DWR [DIURL] oder ICEPush [ICURL] können dabei helfen, serverübergreifend zu entwickeln und die Backend-Entwicklung zu vereinfachen.

ARP-Implementierungen in den modernen Applikationsservern nutzen das Java NIO API, um hochskalierbare asynchrone Client-Server-Kommunikationen zu unterstützen. All diese Implementierungen sind sehr gut geeignet für die Comet-Technologien und überall dort, wo langlebige HTTP-Verbindungen existieren. Die meist bekannten und verbreiteten ARP-Lösungen sind [AWURL]

- Tomcat und JBoss Comet Processors.
- GlassFish Grizzly Connector.
- Jetty Continuations.
- WebLogic Future Response Servlet.
- WebSphere Asynchronous Request Dispatcher.

In der vorliegenden Diplomarbeit wird nur auf die Konfiguration der ersten drei Server eingegangen.

Tomcat 6 und JBoss 4.2

Seit Tomcat 6 gibt es eine optionale ARP-Unterstützung in dem Tomcat *Servlet-Container*.

JBoss-Applikationsserver verwendet den Tomcat *Servlet-Container* und hat somit auch eine Integration der NIO-Schnittstelle. Um den *NIO-Connector* zu nutzen, ist der folgende Eintrag in der Konfigurationsdatei `server.xml` erforderlich [OYURL]:

```
<Connector port="8080"
  protocol="org.apache.coyote.http11.Http11NioProtocol"
  connectionTimeout="60000" redirectPort="8443"
/>
```

Der *Default-Connector* soll dabei auskommentiert oder komplett aus der `server.xml` entfernt werden.

GlassFish v2.1, v3

GlassFish-Server in der Version v2.1 und höher bietet die Vorteile einer asynchronen *non-blocking* Kommunikation mit dem Modul namens "Grizzly" an. Um das "Grizzly"-Modul nutzen zu können, muss man die Eigenschaft (*property*) "`cometSupport`" zu `http-listener` in der Konfigurationsdatei `domain.xml` hinzufügen [OYURL].

```
<http-listener default-virtual-server="server"
  server-name=""
  address="0.0.0.0"
  port="8080"
  id="http-listener-1">
  <property name="cometSupport" value="true" />
</http-listener>
```

Jetty 6.1

Seit Jetty 6.1 ist eine eigene Lösung namens *Jetty-Continuations* verfügbar. *Jetty-Continuations* erlaubt die Bearbeitung einer HTTP-Anfrage auszusetzen oder auf später zu verschieben und den dazugehörigen *Thread* für eine neue Anfrage zu verwenden. Es ist keine spezielle Konfiguration erforderlich. Um von der Jetty-Lösung Gebrauch zu machen, müssen Web-Entwickler explizit die Continuations-Klassen verwenden und ihre Anwendungen entsprechend umstrukturieren. Der Aufruf

```
ContinuationSupport.getContinuation()
```

liefert eine neue Instanz der Klasse `Continuation`. Die Klasse `Continuation` stellt zwei Methoden `suspend()` und `resume()` bereit, mit deren Hilfe man die aktuelle HTTP-Anfrage aussetzen bzw. bei neuen Ereignissen wieder aufnehmen kann.

7.2 Frameworks und Engines zur Auswahl

RichFaces / PrimeFaces

In den JavaServer Faces Komponenten-Bibliotheken RichFaces und PrimeFaces [PFURL] existieren spezielle Push- und Polling-Komponenten. Die Push-Komponente in RichFaces baut auf dem Atmosphere-Framework, das weiter unten noch beschrieben wird. Die Push-Komponente in PrimeFaces hat eine eigene Implementierung für WebSocket. Die einfachste

Polling-Variante erfordert kein spezielles Framework. Trotzdem gibt es fertige JSF-Komponenten (JavaScript Widgets), die den Umgang mit dem Polling vereinfachen. So existieren in RichFaces und PrimeFaces spezielle XHTML-Tags dazu. Es reicht meistens `<aj4:poll>` (RichFaces) oder `<p:poll>` (PrimeFaces) in eine JSF-Seite einzufügen, damit die auf dem Polling basierende Kommunikation beginnen kann.

CometD mit Dojo

CometD [CDURL] implementiert das Bayeux-Protokoll [BEURL] zur bidirektionalen Kommunikation über HTTP und stellt eine JavaScript-Bibliothek in zwei Varianten bereit - für Dojo und jQuery [JQURL]. Zudem gibt es eine in Java geschriebene Clientbibliothek, um das Protokoll auch in Java-Applikationen zu nutzen. Für die Serverseite stellt CometD ebenfalls in Java sowie in Perl und Python geschriebene Bibliotheken zur Verfügung, über die die eigentliche Applikationslogik umgesetzt wird.

Atmosphere-Framework

Das Atmosphere-Framework [ATURL] unterstützt folgende Protokolle: Comet, WebSocket und XMPP. In der Planung ist die Unterstützung für OpenAjax-Push. Das Framework ermöglicht das Bauen von portablen Web-Applikationen. Portabel bedeutet, dass die Web-Applikation auf Tomcat, JBoss, WebLogic, Jetty, GlassFish, Resin und jedem anderen Applikationsserver laufen kann. Der Applikationsserver muss dabei nicht unbedingt die Servlet 3.0 Spezifikation unterstützen. Das Atmosphere-Framework bietet neben einem einheitlichen Server-API auch die clientseitige Bibliothek. Zudem kümmert es sich um die Integration in diverse Cluster-Umgebungen und JMS (*Java Message Service*). Atmosphere kann in fast jedes Web-Framework integriert werden – JSF, GWT, Wicket, Spring usw.

Direct Web Remoting (DWR)

Mit DWR [DIURL] lassen sich bestehende Web-Anwendungen leicht "*ajaxifizieren*". Auch "*Reverse Ajax*" ist eine wichtige Eigenschaft von DWR. Direct Web Remoting unterstützt Piggyback, Polling and Comet als bidirektionale Technologien. Es gibt zahlreiche Quellen, die "*Reverse Ajax*" in DWR genauer beschreiben [SAL08], [ZAM10]. Genauso wie Atmosphere unterstützt DWR verschiedene Applikationsserver. Der Ansatz von DWR ist, in einer XML-Datei definierte Java-Klassen zu identifizieren, so dass deren Methoden dann per JavaScript von der Clientseite her zur Verfügung gestellt werden. Das DWR-Framework generiert JavaScript-Code, der es Browsern ermöglicht, mit dem Java-Code auf dem Server zu agieren. Man ruft damit Java-Methoden in korrespondierenden JavaScript-Methoden auf.

Ajax Push Engine (APE)

Ajax Push Engine [APURL] ist eine quelloffene Technologie für die Umsetzung von Server-Push. APE erlaubt Daten in Echtzeit über den Browser zu streamen. Das Projekt unterteilt sich in zwei Teile: den APE Server und das APE JavaScript Framework. Der APE Server wurde in der Programmiersprache C geschrieben und ist daher sehr perfomant. Der Kern des APE JavaScript Frameworks wurde mit Hilfe der JavaScript-Bibliothek MooTools programmiert. Das eigene APE-Protokol ist das Bindeglied zur Kommunikation zwischen dem APE Server und dem APE JavaScript Framework. Das APE-Protokol ist ein JSON-basiertes Protokoll und nutzt wahlweise eine der drei bidirektionalen Technologien: HTTP Long-Polling, HTTP Streaming oder WebSocket. Weitere Vorteile sind eine ausführliche

Dokumentation und die aktive Weiterentwicklung.

ICEPush
ICEPush-Framework [ICURL] baut auf OpenAjax-Push. Es wirkt damit vor allem dem Verbindungslimit entgegen. ICEPush-Integration mit den Web-Frameworks, -Bibliotheken, wie beispielsweise GWT, Wicket, Grails, JQuery, JSF und JSP, ist hervorragend. Das Framework besteht aus einer einzigen `jar` Datei (Archivdatei), die einfach im Klassenpfad (*Classpath*) der Anwendung liegen muss, um die bidirektionale Kommunikation nutzen zu können. Es gibt auch eine kommerzielle Variante von ICEPush, welche einen sehr leistungsfähigen Enterprise Push-Server beinhaltet. Der Enterprise Push-Server profitiert von der oben erwähnten Java NIO-Schnittstelle und unterstützt somit *Asynchronous Request Processing*. Der Push-Server ermöglicht hochskalierbare, JMS-basierte Lösungen für eine große Anzahl der heutigen Applikationsserver zu bauen.

xLightweb
xLightweb [XLURL] ist eine HTTP-Bibliothek, die die Entwicklung von perfomanten und hochskalierbaren Netzwerkanwendungen ermöglicht. Die Bibliothek bietet einfache und intuitive Schnittstellen, sowohl clientseitig als auch serverseitig. Der Client kann in allen gängigen Programmiersprachen implementiert werden. Serverseitig bietet xLightweb eine gute ARP-Unterstützung und eine gelungene Implementierung von WebSocket-API sowie Server-Sent Events (diskutiert im Kapitel 4.8.1). Server-Sent Events ist die wichtigste Voraussetzung für das Echtzeit-Web überhaupt. Leider wird die xLightweb-Bibliothek nicht mehr weiterentwickelt. Die Fehlerbehebung wird dennoch weiter gemacht.

Lightstreamer
Lightstreamer [LIURL] ist eine Push-Engine, die sich seit Jahren in geschäftskritischen Bereichen bewährt hat. Lightstreamer kann diverse Echtzeit-Daten von bzw. zu einer großen Anzahl von Geräten / Browsern / Applikationen streamen. Unterstützt werden unter anderem folgende Technologien: HTML, HTML5, Ajax, Flex, Silverlight, iOS, Android, BlackBerry, Windows Phone, Java, .NET, und Excel. Man sieht, dass Lightstreamer nicht nur auf webbasierte Technologien beschränkt ist. In webbasierten Applikationen macht Lightstreamer verschiedene Polling-Varianten, Comet und XMPP zunutze.

jWebSocket
Mit jWebSocket [JWURL] steht eine OpenSource Java Implementierung des neuen HTML5 WebSocket-Protokolls sowie eine kleine JavaScript-Bibliothek zur Verfügung. jWebSocket ist ein Java-Kommunikationsserver und ein JavaScript-Client ohne spezielle *Third-Party-Plugins*. Der Server ist eine einfache `jar` Datei und kann *standalone* oder als Dienst betrieben sowie in Web-Applikationen unter Tomcat, JBoss, GlassFish oder Jetty integriert werden. Für die Clientseite ist lediglich eine einzelne JavaScript-Datei erforderlich. Laut Herstellerangaben werden folgende Browser unterstützt: Chrome 4.0.429 und höher, Safari 5 und höher, Firefox 3.7a6 und höher, Opera 10.7 und höher. Internet Explorer 6-9 wird nicht oder nur teilweise unterstützt.

Kaazing Plattform
Die Kaazing Plattform (noch das Kaazing Gateway genannt) [KAURL] steht in zwei

Varianten zur Verfügung. Es gibt eine OpenSource-Lösung, die einen Server mit der WebSocket-Implementierung bereitstellt. Mit dem Kaazing-Server können Entwickler eine Vollduplex-Kommunikation in Echtzeit in ihren Web-Anwendungen realisieren. Darüber hinaus wird auch eine kostenpflichtige Enterprise-Variante angeboten, die neben Support auch eine kommerzielle Lizenz, verschiedene Management-Tools, Security-Features und spezielle Bibliotheken für die Unterstützung von Adobe Flex mit enthält. Die Kaazing Plattform bietet die Unterstützung für Server-Sent Events, Gruppenchat-Unterstützung über XMPP, eine HTML5 Speicherlösung auf DOM-Basis und vieles mehr.

Adobe Messaging-Technik BlazeDS
BlazeDS [BLURL] ist eine OpenSource-Version der serverbasierten *Remoting*- und *Messaging*-Funktionen von *Adobe*. Mit BlazeDS lassen sich Daten in Echtzeit an Adobe-Flex- und AIR-Programme übermitteln und damit *Rich Internet Applications* beschleunigen. BlazeDS arbeitet nach dem Publish/Subscribe-Prinzip. Eine wesentliche Rolle spielt dabei das dazugehörige binäre *Action Message Format* (AMF), das es laut Adobe Anwendungen erlaubt, Daten zehnmal schneller zu laden als es mit textbasierten Formaten, wie XML, der Fall wäre. BlazeDS kann mit den folgenden bidirektionalen Technologien benutzt werden: Polling, Piggyback, HTTP Long-Polling und HTTP Streaming. Ein Nachteil von BlazeDS ist die erforderliche Installation des Flash-Plugins im Web-Browser. Mit Flex entwickelte Anwendungen werden in einer Flash-Player-Umgebung ausgeführt.

In der vorliegenden Diplomarbeit wurde das Atmosphere-Framework ausgewählt. Es bietet ein einheitliches Comet und WebSocket API für verschiedene Java-EE-Server. Ein Entwickler schreibt sein Programm gegen das Atmosphere API und das Framework stellt zur Laufzeit sicher, dass die korrekte Comet- oder WebSocket-Implementierung genutzt wird. Dabei ist es auch unwichtig, ob der Server Servlet 3.0-konform ist oder nicht. Atmosphere zeichnet sich über eine serverübergreifende ARP-Unterstützung und eine hervorragende Skalierbarkeit aus. Dank der JMS-Integration gehen zudem neue Daten (*Updates*) nie verloren. Das ist besonders für Long-Polling und überall dort interessant, wo der Datenverlust hoch ausfallen kann. Auf der Clientseite nimmt das jQuery-Plugin die ganze Arbeit bei der Einbindung bidirektionaler Kommunikation ebenso ab. Mit dem Atmosphere-Framework lassen sich somit webbasierte Applikationen auf Basis von drei wichtigsten Transport-Protokollen – HTTP Long-Polling, HTTP Streaming und WebSocket – mühelos entwickeln. Falls der Browser den WebSocket-Standard noch nicht unterstützt, kann das Framework automatisch eine der eingestellten *Fallback*-Varianten nutzen – Long-Polling oder Streaming. Diese Vorteile macht die Beispielanwendung zunutze und integriert alle drei genannten Transport-Protokolle für einen praktischen Vergleich bei der Auswertung der Ergebnisse [Kapitel 8].

7.3 Entwicklung einer JavaScript-Bibliothek für die Weboberfläche

Für die kollaborative Beispielanwendung „Whiteboard“ wurde eine eigene JavaScript-Bibliothek entwickelt. Als Grundlagen für die Entwicklung dienten zwei populäre JavaScript-Bibliotheken - jQuery [JQURL] und Raphaël [RLURL]. JQuery ist eine umfangreiche Bibliothek, die komfortable Funktionen zu DOM-Manipulation, -Navigation, Effekten und Animationen, Ajax-Funktionalitäten, Ereignisbehandlung usw. zur Verfügung

stellt. Es gibt zahlreiche jQuery-Plugins, die von jQuery-Community bereitgestellt werden. In der Beispielanwendung werden z.B. Plugins für die Validierung von Eingabewerten, für die Einschränkung von Eingaben auf Minimal- / Maximal- und numerische Werte benutzt. Auch der JavaScript-Teil des Atmosphere-Frameworks wurde in Form eines jQuery-Plugins realisiert. Die Raphaël-Bibliothek erleichtert den Umgang mit Vektorgrafiken im Web und setzt dabei auf SVG (*Scalable Vector Graphics)* und VML (*Vector Markup Language* von *Microsoft*). Vektorgrafiken lassen sich mit einem Zitat aus der Wikipedia beschreiben: *„Vektorgrafiken basieren anders als Rastergrafiken nicht auf einem Pixelraster, in dem jedem Bildpunkt ein Farbwert zugeordnet ist, sondern auf einer Bildbeschreibung. So kann beispielsweise ein Kreis in einer Vektorgrafik über Lage des Mittelpunktes, Radius, Linienstärke und Farbe vollständig beschrieben werden; nur diese Parameter werden gespeichert. Im Vergleich zu Rastergrafiken lassen sich Vektorgrafiken daher oft mit deutlich geringerem Platzbedarf speichern. Eines der wesentlichen Merkmale und Vorteile gegenüber der Rastergrafik ist die stufenlose und verlustfreie Skalierbarkeit“*. Mit Raphaël lassen sich Vektorgrafiken browserunabhängig erstellen. Raphaël unterstützt die Browser Firefox 3.0+, Safari 3.0+, Chrome 5.0+, Opera 9.5+ und Internet Explorer 6.0+. In der Anwendung „Whiteboard“ wird diese Bibliothek für das Zeichnen von Figuren, Texten, Bildern und für alle im Kapitel 6 spezifizierten Manipulationen auf diesen Objekten verwendet.

Für das Zeichnen im Web gibt es auch andere Alternativen. Zu nennen sind hier vor allem PaintWeb [PBURL] und SVG-edit [STURL]. PaintWeb hat den Nachteil, dass es auf dem Canvas-Element basiert. Canvas ist ein Bestandteil von HTML5 und ermöglicht ein dynamisches Rendern von Bitmap-Grafiken. Leider nicht alle Browser unterstützen das neue Canvas-Element. SVG-edit basiert auf SVG zur Beschreibung zweidimensionaler Vektorgrafiken. Der Nachteil von SVG ist eine mangelhafte Unterstützung im Internet Explorer. Internet Explorer bis Version 9 implementiert seine eigene proprietäre Sprache VML zur Beschreibung von Vektorgrafiken. Diverse andere Lösungen, die auf HTML basieren, sind nicht effektiv. Es liegt daran, dass HTML nicht für das Zeichnen von Grafiken gedacht wurde. In der Regel machen all diese Lösungen Gebrauch von gruppierten HTML `div` Elementen, die gewünschte Formen bilden.

7.3.1 Initialisierungsskript

Die JavaScript-Bibliothek für die Weboberfläche besteht im Wesentlichen aus drei Teilen. Der erste Teil ist die JavaScript-Datei `whiteboard.js`. Diese Datei enthält notwendige Initialisierungsschritte für die gesamte Anwendung sowie einige Hilfsfunktionen. Das Script bindet alle Teile der Web-Applikation zusammen und stellt eine einheitliche Schnittstelle zur Kommunikation mit dem Benutzer nach außen dar. Dort werden unter anderem alle Dialoge aufbereitet und die Funktionalität für *Pin / Unpin* der *Panels* implementiert. Man kann nicht in einer Arbeit alle Funktionen abdecken. Exemplarisch wird gezeigt, wie man einen Dialog für das Einfügen eines Bildes aufbereitet. PrimeFaces hat eine Dialog-Komponente, die auf XHTML-Seiten (*Facelets*) mit dem Tag `p:dialog` erzeugt wird. Der Dialog lässt sich in JSF mit dem folgenden Code schreiben:

```
<p:dialog id="dialogInputImage" header="Input image URL"
```

```
  resizable="false" closable="true" modal="true"
  styleClass="validatable">
  <ul id="errorImageUl" class="errormsg"/>
  <h:panelGrid id="dlgImageGrid" columns="2">
    <h:outputLabel value="Image URL" for="inputUrl"/>
    <p:inputText id="inputUrl"/>
    <h:outputLabel value="Image width (px)" for="imgWidth"/>
    <h:panelGroup>
      <p:inputText id="imgWidth" maxlength="4"/>
      <h:outputLabel value="Image height (px)"
            for="imgHeight" style="margin: 0 10px 0 10px;"/>
      <p:inputText id="imgHeight" maxlength="4"/>
    </h:panelGroup>
  </h:panelGrid>
</p:dialog>
```

Das JSF-Framework erzeugt aus `p:dialog` einen modalen Dialog. Die JSF Dialog-Komponente basiert auf dem jQuery UI Dialog. JQuery UI [JUURL] ist eine Sammlung von GUI-Widgets für das Web. Der Dialog erfordert eine Konfiguration. Es müssen ggf. *Event-Handlers* für das Öffnen / Schließen des Dialogs registriert und zwei Buttons “Accept” / “Close” hinzugefügt werden. Der Dialog sollte nur die gültigen Eingaben einer Bild-URL, einer Bildbreite und einer Bildhöhe akzeptieren. Die gültigen Eingaben können mit Hilfe von JQuery Validation Plugin [VPURL] validiert werden. Die Bild-URL muss einer gültigen URL-Schreibweise entsprechen und für die Bildbreite, -höhe sind nur positive Zahlen erlaubt. Das jQuery Validation Plugin hat eine Methode `validate()`, die auf einer HTML-Form aufgerufen wird und ein Validator-Objekt erzeugt. Sobald der Benutzer den Button “Accept” betätigt, werden alle Benutzereingaben anhand des Validator-Objektes überprüft. Die `validate()` Methode erwartet mehrere Optionen als Parameter, um z.B. durch das Plugin vorgegebene oder benutzerdefinierte Validierungsregeln, Fehlertexte und den Platz für die Anzeige der Fehlertexte festzulegen. Der nächste Codeausschnitt zeigt den Aufruf dieser Methode für den oben genannten Dialog.

```
// create validator
dialogValidator = jQuery("#mainForm").validate({
  onfocusout: false,
  onkeyup: false,
  errorPlacement: function(label, elem) {
  elem.closest(".validatable").
              find(".errormsg").append(label);
  },
  wrapper: "li",
  rules: {
    inputUrl: {
      url: true
    },
    imgWidth: {
```

```
      imageSize: "#inputUrl:filled"
    },
    imgHeight: {
      imageSize: "#inputUrl:filled"
    }
  },
  messages: {
  inputUrl: "Please enter a valid image URL.",
  imgWidth: "Please enter a valid image width
            (only positive digits are allowed).",
  imgHeight: "Please enter a valid image height
            (only positive digits are allowed)."
  }
});
```

`inputUrl`, `imgWidth` und `imgHeight` sind die Id-Attribute der entsprechenden Eingabefelder (`p:inputText`) im oben erzeugten Dialog (`p:dialog`) und verknüpfen damit die Validierungsregeln sowie Fehlertexte mit den Eingabefeldern.

```
jQuery("#dialogInputImage").dialog("option", "buttons", {
  "Accept": function() {
    // validate fields if user click on the "Accept" button
    var isValid1 = dialogValidator.element("#inputUrl");
    var isValid2 = dialogValidator.element("#imgWidth");
    var isValid3 = dialogValidator.element("#imgHeight");

    if ((typeof isValid1 !== 'undefined' && !isValid1) ||
      (typeof isValid2 !== 'undefined' && !isValid2) ||
      (typeof isValid3 !== 'undefined' && !isValid3)) {
      // validation failed
      return false;
    }

    // validation is ok ==> create image
    ...
  },
  "Close": function() {
    jQuery(this).dialog("close");
  }
}).bind("dialogclose", function(event, ui) {
  // reset input
  jQuery(this).find("#inputUrl").val('');
  // clean up validation messages
  jQuery("#errorImageUl").html('');
});
```

Das endgültige Ergebnis, wenn alle festgelegten Plausibilitätsprüfungen fehlschlagen, wird

in der Abbildung 31 präsentiert.

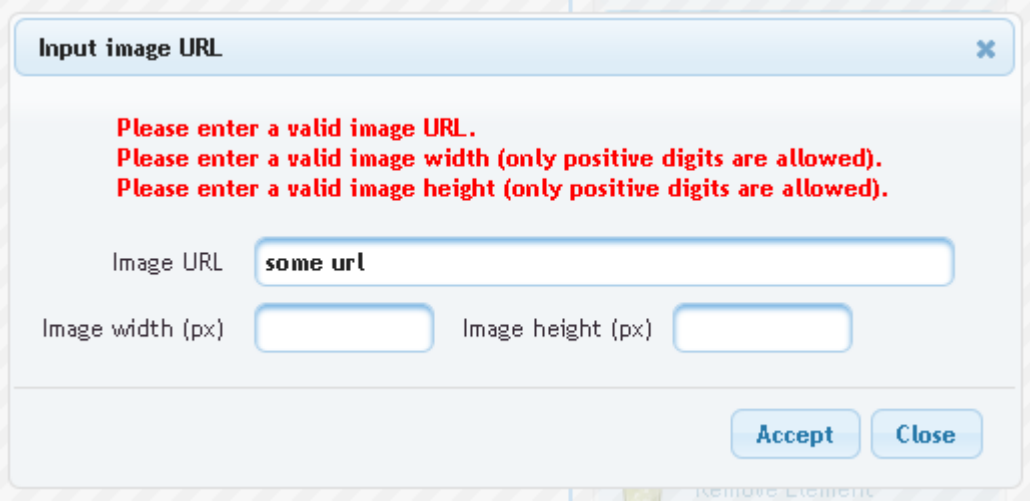

Abbildung 31: Dialog "Input image URL"

7.3.2 Klasse WhiteboardConfig

Der zweite Teil der JavaScript-Bibliothek für die Weboberfläche stellt die Datei `wboard-config.js` dar, die für die Whiteboard-Konfiguration vorgesehen ist. Die Konfiguration besteht aus Konstanten, die auf der Clientseite benötigt werden. Die Konstanten sind in objektorientiertem Stil definiert und befinden sich in einem JavaScript-Objekt namens `WhiteboardConfig`. Ein Auszug aus dem Quellcode soll die Struktur dieses Objektes verdeutlichen:

```
/**
 * Whiteboard configuration.
 */
WhiteboardConfig = function() {
    // ids to external objects
    this.ids = {
        whiteboard: "whiteboard",
        dialogInputText: "dialogInputText",
        dialogInputImage: "dialogInputImage",
        dialogIcons: "dialogIcons",
        dialogResize: "dialogResize"
    };

    // default properties for all elements
    this.properties = {
        text: {
            "text": "",
            "font-family": "Verdana",
            "font-size": 18,
            ...
        },
        freeLine: {
            "stroke": "#000000",
```

```
            "stroke-width": 3,
            "stroke-dasharray": "No",
            ...
        },
        straightLine: {
            "stroke": "#000000",
            "stroke-width": 3,
            "stroke-dasharray": "No",
            ...
        },
        rectangle: {
            "width": 160,
            "height": 100,
            "r": 0,
            ...
        },
        circle: {
            "r": 70,
            "fill": "#008080",
            "stroke-opacity": 1.0,
            ...
        },
        ellipse: {
            "rx": 80,
            "ry": 50,
            "fill": "#BA55D3",
            ...
        },
        image: {
            "width": 150,
            "height": 150,
            "rotation": 0
        },
        icon: {
            "scale": 1.0,
            "rotation": 0
        }
    };

    // dasharray mapping
    this.dasharrayMapping = {
        "No": "",
        "Dash": "-",
        "Dot": ".",
        "DashDot": "-.",
        "DashDotDot": "-..",
        "DotBlank": ". ",
        "DashBlank": "- ",
        "DashDash": "--",
        "DashBlankDot": "- .",
        "DashDashDot": "--.",
        "DashDashDotDot": "--.."
    }
```

```
    // attributes for various helper objects
    this.attributes = {
        helperRect: {"stroke-width": 2, "stroke-opacity": 0,
             "fill": "#0D0BF5", "fill-opacity": 0},
        circleSet: {"stroke": "#0D0BF5", "stroke-width": 1,
             "stroke-opacity": 0, "fill": "#0D0BF5", "fill-opacity": 0},
        opacityHidden: {"stroke-opacity": 0, "fill-opacity": 0},
        opacityVisible: {"stroke-opacity": 0.8, "fill-opacity": 0.8}
    };

    ...

    // icons definitions (paths)
    this.svgIconSet = {
        snow: "M25.372,6.912c-0.093-3.925-3.302-7.078-7.248-7...,
        rain: "M25.371,7.306c-0.092-3.924-3.301-7.077-7.248-7...,
        smile: "M16,1.466C7.973,1.466,1.466,7.973,1.466,16c0,8...,
        ...
    };
}
```

Wesentliche Bestandteile dieser Konfiguration sind Referenzen auf externe Objekte, wie Dialoge und die Arbeitsfläche des Whiteboards, Standard-Eigenschaften für Whiteboard-Elemente, Mapping zwischen Werten und Texten des Linienstils, Attribute für diverse Hilfsobjekte, die beim Markieren (*Select Element*), Löschen (*Remove Element*), Verschieben (*Move Element*) und weiteren Aktionen auf Elementen erscheinen, Symbol-Definitionen als SVG-Pfade und noch weitere Konstanten, deren Aufzählung den Rahmen dieser Arbeit sprengen würde. `WhiteboardConfig` kann mit dem `new` Operator instanziiert werden

```
new WhiteboardConfig()
```

und wird an `WhiteboardDesigner` übergeben – das Hauptobjekt für das Zeichnen mit der Raphaël-Bibliothek. Auf `WhiteboardDesigner` wird noch weiter eingegangen.

7.3.3 Klasse WhiteboardDesigner

Der letzte und wichtigste Teil ist die Datei `wboard-drawing.js`. Diese Datei bildet den Kern der gesamten Anwendung, weil dort alle Zeichen-Routinen entwickelt wurden. Die Zeichen-Routinen befinden sich in dem JavaScript-Objekt `WhiteboardDesigner`. Das Objekt wurde genauso wie die Konfiguration in objektorientiertem Stil entwickelt und kapselt die Methoden aus der Raphaël-, und Atmosphere-Bibliothek. Die Anbindung an den clientseitigen Teil des Atmosphere-Frameworks wird im Unterkapitel 7.5 demonstriert. Die nachfolgende Aufstellungen zeigt exemplarisch die wichtigsten Methoden aus der Raphaël-Bibliothek. Insgesamt wurden mehr als 18 Methoden benutzt. Die Arbeitsfläche (in Raphaël als *Canvas* genannt) wird wie folgt angelegt:

```
var paper = Raphael(container, width, height);
```

Übergeben wird ein HTML-Element (z.B. `<div>`), das die Arbeitsfläche umschließen soll. Anstatt des HTML-Elements kann auch eine Id stehen, die auf das HTML-Element

verweist. Weitere Übergabeparameter sind die Whiteboard-Größen. Zurückgegeben wird eine Referenz auf die Arbeitsfläche, mit der Whiteboard-Elemente gezeichnet werden können. Um einen Text zu zeichnen, ruft man die Methode `text()` auf dieser Referenz auf und übergibt man die Text-Koordinaten (linke obere Ecke) sowie den Textinhalt selbst.

```
var t = paper.text(x, y, text);
```

Linien lassen sie mit der Methode `path()` zeichnen.

```
var c = paper.path(pathString);
```

`pathString` ist ein sogenannter SVG-Pfad und kann beispielsweise wie folgt aussehen: `"M10 10L90 90".` Diese Schreibweise bedeutet: “gehe zu dem Punkt mit Koordinaten (10, 10) und ziehe eine Linie bis zum Punkt (90, 90)”. Um einen Kreis zu zeichnen, ruft man die Methode `circle()` auf.

```
var c = paper.circle(x, y, radius);
```

Diese Methode erwartet die Koordinaten des Kreismittelpunktes und den Radius. Für das Zeichnen von Rechteck und Ellipse gibt es ähnliche Methoden. Man kann auch ein Set von Elementen erzeugen. Das Set wird als ein einziges Element betrachtet. Um z.B. zwei Kreise zu einem Set zu gruppieren, ruft man auf

```
var st = paper.set();
st.push(
    paper.circle(10, 10, 5),
    paper.circle(30, 10, 5)
);
```

Weitere nützliche Methode ist `rotate()`. Mit `rotate()` lässt sich ein Element rotieren. Die Übergabeparameter sind der Drehungswinkel und die Angabe, ob das Rotieren absolut oder relativ zur vorherigen Position erfolgen muss. Um z.B. einen Kreis absolut um 45 Grad zu rotieren, ruft man auf

```
c.rotate(45, true);
```

Eine wichtige Methode für die Drag-&-Drop-Funktionalität heißt `translate()`. Diese Methode ermöglicht das Verschieben von Elementen. Die Übergabeparameter sind die Pixel, um welche das Element verschoben werden muss. Der folgende Aufruf verschiebt beispielsweise den Kreis um 10 Pixel nach rechts und unten.

```
c.translate(10, 10);
```

Mit Hilfe der Methoden `remove()` und `clone()` kann man ein beliebiges Element löschen bzw. duplizieren. Mit Hilfe der Methode `clear()` kann man auch das ganze Whiteboard löschen.

```
paper.clear();
```

Als letzte wird die Methode `attr()` genannt. Diese Methode ermöglicht das Setzen von verschiedenen Eigenschaften, wie Koordinaten, Breite, Höhe, Radius, Farbe, Linienart,

-stärke, -stil usw. Sie wird immer dann aufgerufen, wenn die Eigenschaften eines Elementes geändert werden müssen. Um z.B. die Hintergrundfarbe eines Kreises auf Schwarz zu setzen, ruft man auf

```
c.attr("fill", "#000000");
```

`WhiteboardDesigner` stellt 41 *public* Methoden zur Vefügung. Die *public* Methoden bilden eine umfangreiche Schnittstelle (API) für die Arbeit mit dem Whiteboard. Zum Demonstrationszweck wird die Methode `restoreWhiteboard()` aufgeführt, die immer dann aufgerufen wird, wenn sich ein neuer Benutzer dem Whiteboard anschließt oder wenn das Whiteboard aktualisiert werden muss (z.B. durch Drücken der Taste F5 im Browser). Die Aufgabe dieser Methode ist das Wiederherstellen aller vorhandenen Elemente. Elemente werden im JSON-Format an diese Methode übergeben. Das Java-Backend bereit dafür die entsprechende Datenstruktur vor, die folgendermaßen aussieht:

```
{
  "elements": [
    {
      "type": "Circle",
      "properties": {
        "uuid": "2aa262bb-8fcc-4af0-932f-bcf32554537a",
        "radius": 250,
        ...
      }
    },
    {
      "type": "Text",
      "properties": {
        "uuid": "0b21ef9a-28be-4931-93d9-b81f97584374",
        "text": "Hello Whiteboard",
        ...
      }
    },
    ...
  ],
  "message": "5 whiteboard elements have been restored"
}
```

Man sieht, dass `"elements"` ein JavaScript-Array ist, über die iteriert werden kann. Die Methode `restoreWhiteboard()` iteriert über alle Elemente und ruft ihrerseits die *public* Methode `createElement()` auf, um ein Element zu erzeugen und in die Liste aller Whiteboard-Elemente aufzunehmen. Die Liste wird intern verwaltet. Anschließend findet ein Aufruf der *private* Methode `prependMessage()` statt, um eine Nachricht im *Event-Monitoring* Bereich auszugeben (in diesem Fall lautet die Nachricht - "5 whiteboard elements have been restored").

```
this.restoreWhiteboard = function(jsWhiteboard) {
```

```
  var arrElements = jsWhiteboard["elements"];
  for (var i = 0; i < arrElements.length; i++) {
    var objElement = arrElements[i];
    this.createElement(objElement.properties,
                       objElement.type);
  }

  prependMessage(jsWhiteboard["message"]);
}
```

An dieser Stelle sei noch die clientseitige Aktualisierung von Elementen bzw. des Whiteboards selbst zu klären. Whiteboard-Teilnehmer arbeiten parallel und lösen mehrere Aktionen aus, die zu den gleichzeitigen Änderungen führen können. Wie werden die Änderungen synchronisiert? Müssen eventuelle Konflikte bei der Zusammenführung der Änderungen gelöst werden? Die Antworten auf diese Fragen lassen sich aus der JavaScript-Theorie ableiten. JavaScript ist keine *Multi-Threading* Sprache. Die Script-Ausführungen laufen in einem einzigen *Thread* und wenn die Antwort des Servers auf eine AJAX-Anfrage kommt, wird solange mit der Auswertung gewartet, bis die aktuelle Script-Ausführung zu Ende ist. Das heißt, dass ein 2 Sekunden dauernder AJAX-Aufruf innerhalb eines 4 Sekunden laufenden `<script>...</script>` Blocks noch 2 Sekunden warten wird. Erst dann wird mit der Auswertung der Response-Ergebnisse fortgesetzt. Ob das Element gerade in der Bearbeitung ist oder nicht, spielt deshalb für die Element-Aktualisierung keine Rolle. Es gilt somit: Synchronisierung von Aktualisierungen, wenn ein Element oder das ganze Whiteboard gerade in der Bearbeitung ist, erübrigt sich komplett. Eine Warteschlange für angekommene Änderungen ist aus diesem Grund auch nicht nötig. Mehrere *Threads* sind erst ab HTML5 mit dem Web Worker Konzept möglich.

Das Projekt beinhaltet auch andere notwendige Hilfsbibliotheken von Drittanbietern. Es gibt beispielsweise Bibliotheken für die Generierung einer eindeutigen Id (*UUID*), für die Konvertierung von JSON-Daten, für die Einschränkung der Eingabe auf numerische Werte, für die Browser-Erkennung, für die *Logging*-Komponente und für das *Color Picker* Widget. Eine vollständige Auflistung und der Quellcode aller JavaScript-Dateien kann auf der Homepage des Projektes “Collaborative Online-Whiteboard” [OWURL] angesehen bzw. heruntergeladen werden. Im Kapitel 11 wird es dazu noch eine ausführliche Anleitung geben.

7.4 Entwicklung eines Java Modells für das Backend

Das Backend (Teil von Programmen, die auf dem Server laufen) besteht im Wesentlichen aus den Java-Modellklassen für Whiteboard-Elemente und dem Whiteboard selbst, den JSF *Managed Beans* und vielen Hilfsklassen (*Utilities*) sowie JSON-Konvertierungsmethoden. Des Weiteren ist ein Atmosphere-Handler vorhanden, der im nächsten Unterkapitel 7.5 behandelt wird. Die Beispielanwendung benötigt keine Anbindung an eine Datenbank oder andere Persistenzschichten.

Die wichtigsten Modellklassen wurden bereits bei der Gesamtarchitektur im Unterkapitel 6.4 spezifiziert. Ausgehend vom Design gibt es acht Java-Klassen: `Text`, `Free Line`, `Straight Line`, `Rectangle`, `Circle`, `Ellipse`, `Image`, `Icon`. Jede Klasse erbt von der abstrakten Klasse `Rotatable`, welche wiederum von der abstrakten Basis-Klasse `AbstractElement` erbt. Außerdem gibt es abstrakte Klassen `Positionable` und `Line`. Die Klassenhierarchie wird im Klassendiagramm 32 abgebildet. Gezeigt werden nur noch die Eigenschaften. Die Getter- / Setter-Methoden werden nicht gezeigt, weil sie gleichnamig und trivial sind. Andere Methoden existieren nicht. Bei Modellklassen handelt es sich um die gewöhnlichen *JavaBeans*. *JavaBeans* besitzen keine Geschäftslogik und bieten nur die Zugriffsmethoden auf die Eigenschaften nach außen hin an.

Es gibt noch drei Modellklassen, die zu erwähnen sind. Das sind `TruncatedElement`, `TrancatedLine` und `TrancatedPositionable`. Sie wurden zwecks Optimierung der Datenmenge bei der Datenübertragung eingeführt. Oft ist es nicht sinnvoll, die ganzen Elementeigenschaften an den Client zu übertragen. Es gibt einige Anwendungsfälle, bei denen nur noch die Untermenge der Menge aller Eigenschaften interessant ist. Ein typisches Beispiel ist das Verschieben von Elementen. Wird ein beliebiges Element per Drag-&-Drop verschoben, ändern sich nur seine Koordinaten bzw. sein Linienpfad (SVG-Pfad im Falle einer Linie). Alle anderen Eigenschaften bleiben unverändert. JSON-Daten, die an den Client zu übertragen sind, sollten daher nur wenige veränderte Eigenschaften enthalten. Das wird dadurch erreicht, indem die Instanzen der Klassen `TrancatedLine` bzw. `TrancatedPositionable` erzeugt werden. `TrancatedLine` hat nur noch eine Eigenschaft `path` und `TrancatedPositionable` die Eigenschaften `x` und `y` für Koordinaten. Wurde der Linienpfad geändert, bekommt `TrancatedLine` einen neuen SVG-Pfad gesetzt und wird in das JSON-Format transformiert. Wurden die Koordinaten `x` und `y` eines positionierbaren Elementes geändert, werden sie in das neue Objekt `TrancatedPositionable` gesetzt, welches genauso in das JSON-Format transformiert wird. Mit dieser Strategie wird die übertragene Datenmenge reduziert.

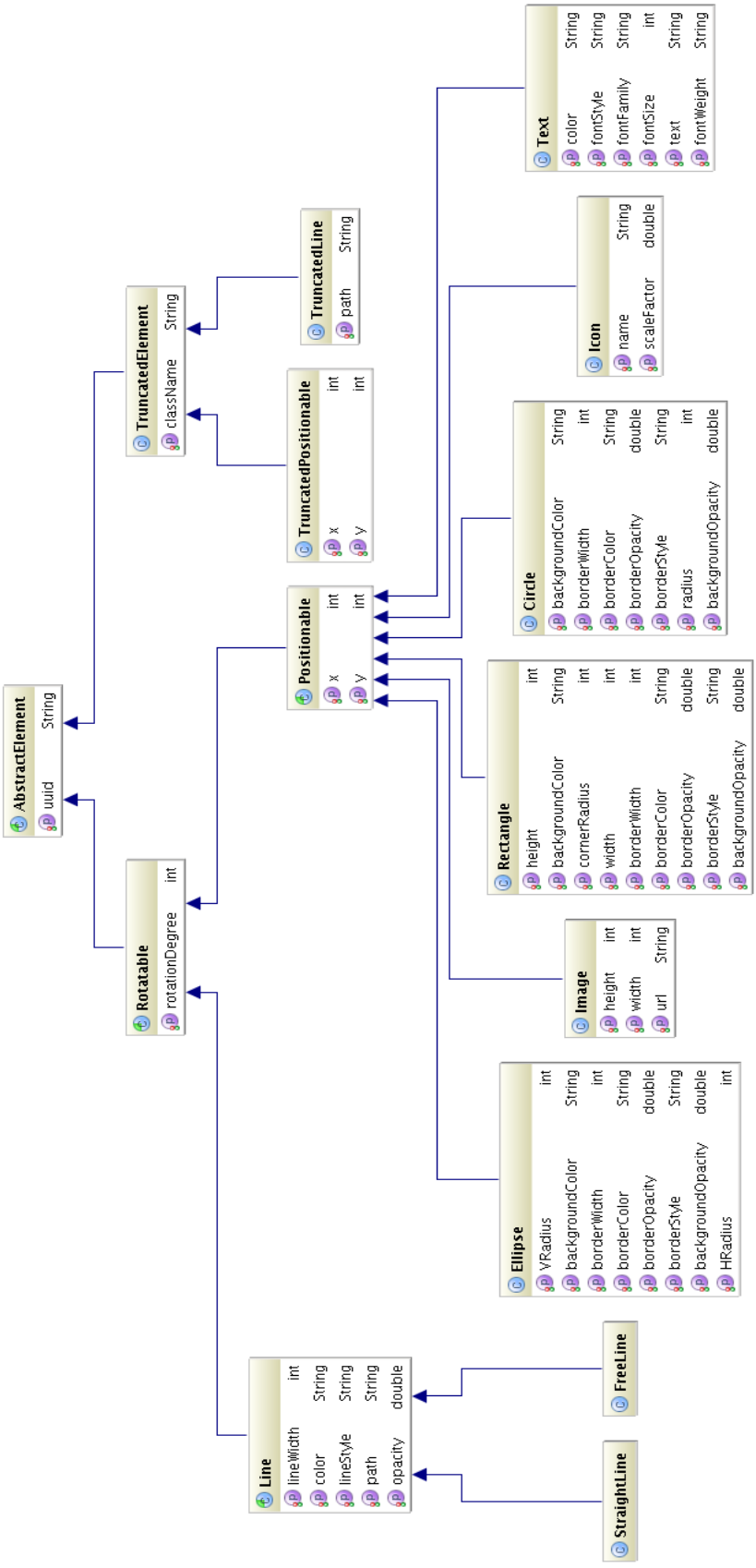

Abbildung 32: Klassendiagramm "Whiteboard-Elemente"

Zu den Modellklassen gehört auch die Klasse `Whiteboard`. `Whiteboard` repräsentiert ein Whiteboard und besitzt die Angaben zur Größe, zum Ersteller, zum Erstellungsdatum, zur Benutzerkennung, zur Anzahl aller angemeldeten Benutzer, zum Transport-Protokoll (Long-Polling, Streaming oder WebSocket) und zu allen Elementen, die auf diesem Whiteboard gezeichnet wurden. Die Abbildung 33 zeigt die genauen Eigenschaften eines Whiteboards.

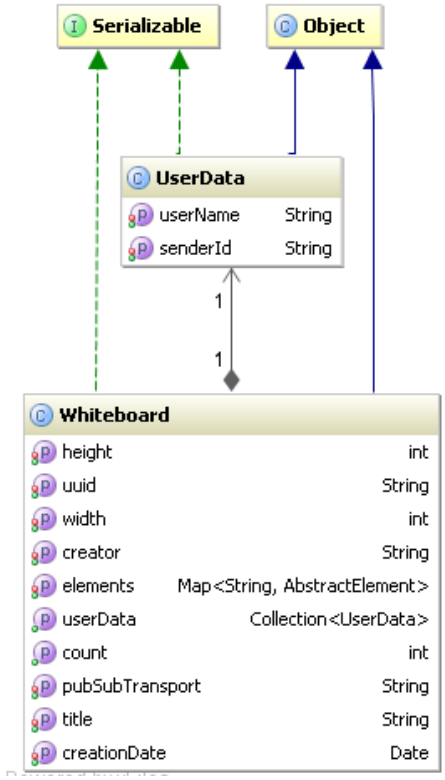

Abbildung 33: Klassendiagramm "Whiteboard"

Die *Managed Beans* sind ein zentraler Bestandteil von JavaServer Faces. In einer JSF-Anwendung bilden sie das Modell und stellen die Verbindung zur Geschäftslogik her. Im Hinblick auf eine strikte Trennung von Präsentation und Logik kommt diesen Klassen damit eine sehr wichtige Rolle zu. In der Praxis sind die Aufrufe der Geschäftslogik komplett in den *Managed Beans* gekapselt. Die Beispielanwendung hat drei Views: Login-Maske für ein neues Whiteboard, Login-Maske, um sich einem bestehenden Whiteboard anzuschließen und die Hauptseite, wie sie in der Abbildung 19 modelliert wurde. Diesen drei Views ist jeweils eine *Managed Bean* zugeordnet: `CreateWhiteboard`, `JoinWhiteboard` und `DisplayWhiteboard`. Alle drei genannten Bean-Klassen halten eine Referenz auf `WhiteboardsManager`. `WhiteboardsManager` ist ebenfalls eine *Managed Bean*, die im „*Application Scope*“ liegt. Die Aufgabe von `WhiteboardsManager` ist die Verwaltung aller Whiteboards. Dafür stellt sie die Methoden `addWhiteboard()`, `updateWhiteboard()` und `removeWhiteboard()` zur Verfügung. Es gibt auch andere *Managed Beans*, die als Container für bestimmte Werte / Zustände benutzt werden oder der Aufbereitung einer Fehlermeldung dienen. Das nachfolgende Klassendiagramm 34

zeigt alle *Managed Beans* mit deren Methoden und Eigenschaften zusammen.

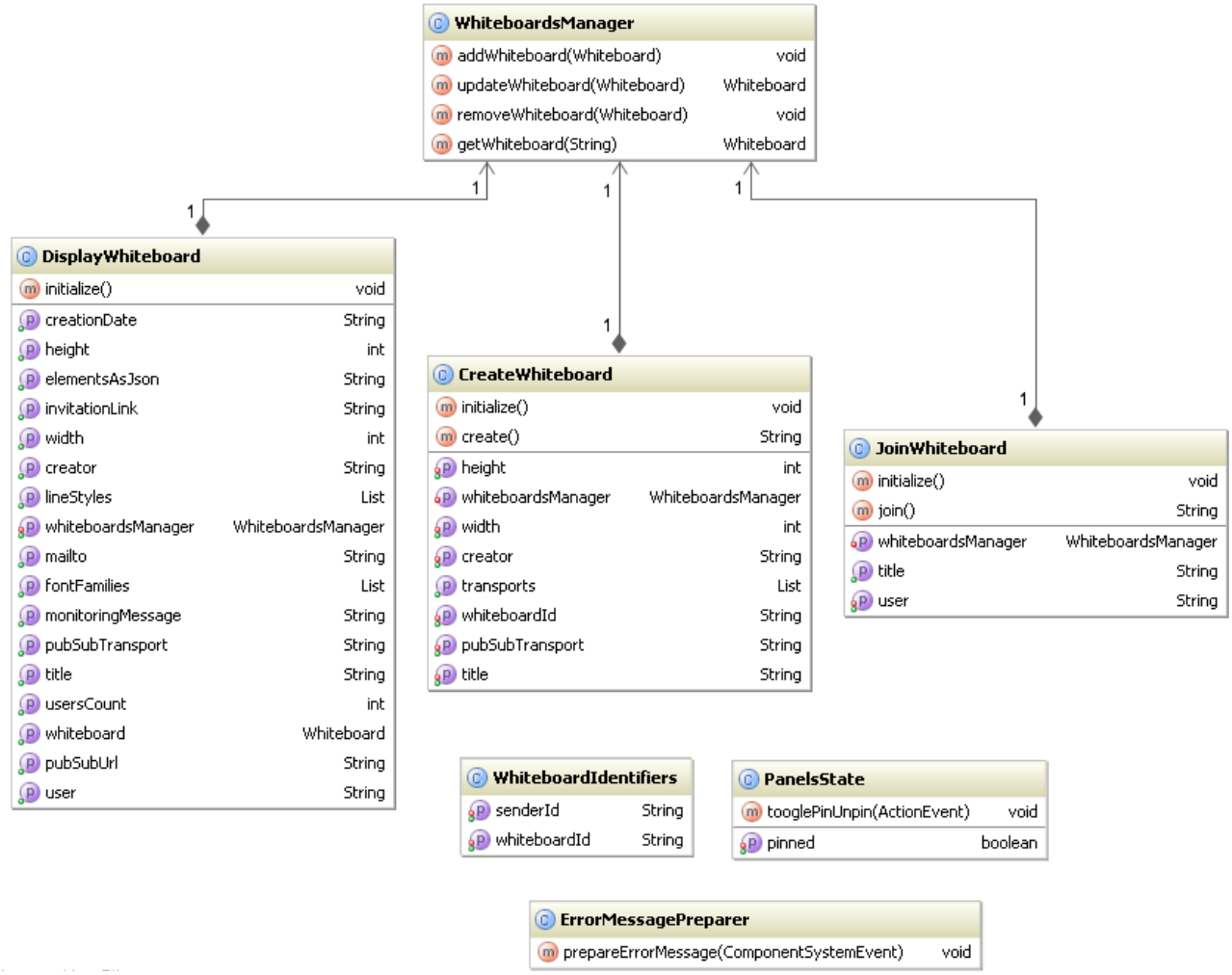

Abbildung 34: Klassendiagramm "Managed Beans"

7.5 Kommunikation mit Atmosphere-Framework

Das Ziel dieses Unterkapitels ist eine systematische Beschreibung aller erforderlichen Schritte zur Implementierung einer bidirektionalen Kommunikation mit dem Atmosphere-Framework. Damit die bidirektionale Kommunikation zustande kommt, muss zuerst ein *Topic* (Thema, Inhalt) definiert werden. Benutzer fungieren als *Subscribers* und abonnieren das *Topic,* um neue Nachrichten zu erhalten. Ein Benutzer, der ein *Topic* abonniert hat, fungiert auch gleichzeitig als *Publisher* und kann die Nachrichten für alle anderen Abonnenten bekanntgeben. Die Abbildung 35 zeigt schematisch das Publisher-Subscriber-Prinzip.

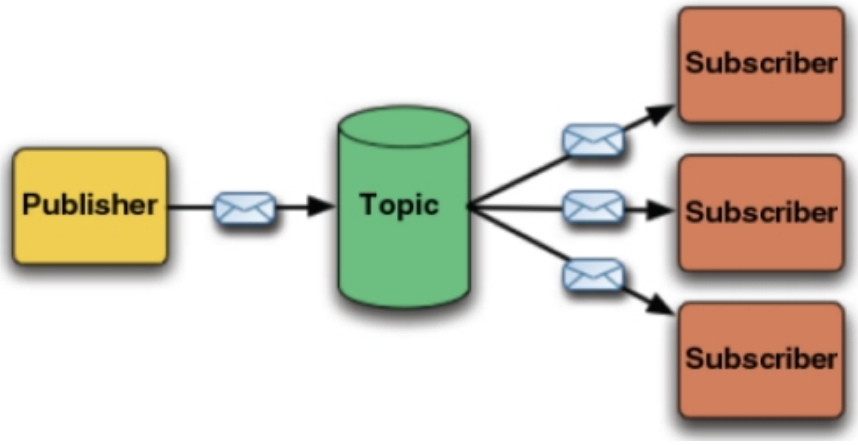

Abbildung 35: Publisher-Subscriber-Prinzip

Technisch gesehen kann das *Topic* über eine eindeutige URL ausgedrückt werden, über welche die bidirektionale Kommunikation stattfindet. Diese URL muss derart definiert werden, dass die bidirektionale Kommunikation über den `Atmosphere-Servlet` läuft (Abbildung 30). HTTP-Anfragen müssen sich in diesem Fall von den herkömmlichen JSF-Anfragen unterscheiden, da sie ansonsten über den `JSF-Servlet` laufen würden. Das kann durch das `servlet-mapping` in `web.xml` erreicht werden. Das Mapping wird weiter besprochen. In der Beispielanwendung setzt sich die Topic-URL aus Whiteboard-Id und Benutzer-Id zusammen. Es bietet sich an, das eigentliche *Topic* mit der Whiteboard-Id gleichzustellen. Die Whiteboard-Id und somit das *Topic* sind für alle eingeladenen Benutzer bekannt, so dass Whiteboard-Teilnehmer dasselbe Topic abonnieren können. Die Benutzer-Id, hier noch *Sender* genannt, dient dazu, den *Publisher* von den Selbstbenachrichtigungen auszuschließen. Wenn eine Nachricht an alle *Subscribers* verschickt wird, hat der Benutzer selbst, von dem die Nachricht ausgegangen ist, kein Interesse daran. Eine technische Umsetzung wird noch weiter in diesem Unterkapitel gezeigt. Eine typische Topic-URL sieht in der Beispielanwendung folgendermaßen aus:

```
http://localhost:8080/pubsub/ea288b4c-f2d5-467f-8d6c-
8d23d6ab5b11/e792f55e-2309-4770-9c48-de75354f395d.topic
```

Das Muster, nach dem die URL gebaut wird, lässt sich wie folgt definieren:

```
http://host:port/pubsub/{topic}/{sender}.topic
```

Das Atmosphere-jQuery Plugin [ATURL] ist in der Datei `jquery.atmosphere.js` enthalten, die in die Webseite eingebunden werden muss. Der zweite Schritt besteht in dem Aufruf der `subscribe()` Methode aus dem Atmosphere-jQuery Plugin. Die wichtigsten Übergabeparameter sind die Topic-URL, die Callback-Funktion, das Transport-Protokoll und die maximal erlaubte Anzahl an gesendeten Requests. Die Callback-Funktion wird immer dann aufgerufen, wenn der Server neue Daten an *Subscribers* sendet, die das *Topic* abonniert haben. `WhiteboardDesigner` kapselt den Aufruf der `subscribe()` Methode.

```
this.subscribePubSub = function() {
  jQuery.atmosphere.subscribe(
    this.pubSubUrl,
    this.pubSubCallback,
    jQuery.atmosphere.request = {
      transport: this.pubSubTransport,
      maxRequest: 100000000
    });

    this.connectedEndpoint = jQuery.atmosphere.response;
}
```

Diese Methode wird nur einmal beim Aufbau der Hauptseite aufgerufen. Der erste Parameter `this.pubSubUrl` ist mit der Topic-URL vorbelegt. Der zweite Parameter `this.pubSubCallback` ist die Callback-Funktion. Der dritte Parameter ist die Request-Konfiguration und hat unter anderem die Definition des Transport-Protokolls (Konstanten "`long-polling`", "`streaming`", "`websocket`" für `this.pubSubTransport`). Die letzte Zeile weist das Objekt `jQuery.atmosphere.response` einer Variablen zu. Das Objekt `jQuery.atmosphere.response` ermöglicht dem *Publisher* das Versenden von Nachrichten an alle *Subscribers*.

Werden nun neue Daten empfangen, wird die Callback-Funktion `this.pubSubUrl` aufgerufen. Dort werden die empfangenen Daten aus dem `response` Parameter extrahiert.

```
this.pubSubCallback = function(response) {
  if (response.transport != 'polling' &&
      response.state != 'connected' &&
      response.state != 'closed' && response.status == 200) {
      var data = response.responseBody;
      if (data.length > 0) {
        // convert to JavaScript object
        var jsData = JSON.parse(data);

        var action = jsData.action;
        var sentProps = (jsData.element != null ?
                         jsData.element.properties : null);
        switch (action) {
          case "join" :
            ...
            break;
          case "create" :
            ...
            break;
          case "update" :
            ...
            break;
```

```
          case "remove" :
            ...
            break;
          ...
          default:
        }

        // show new message in the event monitoring pane
        prependMessage(jsData.message);
      }
  }
}
```

Insgesamt gibt es zehn `case` Anweisungen, die sich nach der Anzahl der Aktionen richten (Unterkapitel 6.3). In `case` Blöcken werden die empfangenen Daten je nach Aktion entsprechend behandelt. Auf eine detaillierte Datenbehandlung wird an dieser Stelle verzichtet. Das Versenden der Daten findet mit Hilfe der oben deklarierten Variablen `this.connectedEndpoint` statt. `WhiteboardDesigner` stellt dafür die Methode `sendChanges()` bereit.

```
this.sendChanges = function(jsObject) {
  // set timestamp
  var curDate = new Date();
  jsObject.timestamp = curDate.getTime() +
                       curDate.getTimezoneOffset() * 60000;

  // set user
  jsObject.user = this.user;

  // set whiteboard Id
  jsObject.whiteboardId = this.whiteboardId;

  var outgoingMessage = JSON.stringify(jsObject);

  // send changes to all subscribed clients
  this.connectedEndpoint.push(this.pubSubUrl, null,
     jQuery.atmosphere.request =
       {data: 'message=' + outgoingMessage});
}
```

Nachdem alle Daten gesetzt wurden, wird das entsprechende JavaScript-Objekt mit dem Aufruf `JSON.stringify(jsObject)` nach JSON konvertiert. Die JSON-Daten werden anschließend mit dem Aufruf `this.connectedEndpoint.push(...)` an die Topic-URL versendet. Der eigentliche Aufruf der Methode `sendChanges()` sieht je nach Anwendungsfall unterschiedlich aus. Als Beispiel wird das Versenden der Daten nach dem Einfügen eines Bildes demonstriert.

```
// send changes to server
this.sendChanges({
  "action": "create",
  "element": {
    "type": this.config.classTypes.image,
    "properties": {
      "uuid": ...,
      "x": ...,
      "y": ...,
      "rotationDegree": ...,
      "url": ...,
      "width": ...,
      "height": ...
    }
  }
});
```

Auf der Clientseite sind damit alle Schritte erledigt.

Die oben eingeführten AJAX-Aufrufe `jQuery.atmosphere.subscribe()` und `jQuery.atmosphere.response.push()` müssen auf der Serverseite abgefangen werden. Das Atmosphere-Framework kann von der Jersey-Bibliothek [JEURL] Gebrauch machen. Jersey ist die OpenSource-Implementierung von Java API für RESTful Web Services. Jersey bietet eigene Java *Annotations* an, die das Abfangen von GET- / POST-Requests für jede beliebige URL deklarativ ermöglichen. Dafür stehen `@GET`, `@POST` und `@PATH` *Annotations*. In der Diplomarbeit wurde die Klasse `WhiteboardPubSub` entwickelt, welche die Topic-URL dank der *Annotation* `@Path("/pubsub/{topic}/{sender}")` abfangen kann. Die mit `@GET` annotierte Methode `subscribe()` fängt den clientseitigen Aufruf `jQuery.atmosphere.subscribe()` ab. Diese Methode teilt dem Atmosphere-Framework mit, den aktuell laufenden Request anzuhalten und so lange offen zu halten, bis ein Ereignis eintritt. In dieser Methode wird auch *Topic* erstellt (`Broadcaster`). Die *Topic*-Zeichenkette wird aus der URL mittels der *Annotation* `@PathParam` extrahiert. Man kann sich `Broadcaster` als eine Warteschlange vorstellen. Sobald eine Nachricht in die Warteschlange landet, wird sie an alle *Subscribers* (alle Browser, die mit dem *Topic* verbunden sind) versendet. Die mit `@POST` annotierte Methode `publish()` fängt den Aufruf `jQuery.atmosphere.response.push()` ab. Die übergebene Nachricht wird serverseitig bearbeitet und an alle *Subscribers* versendet (*broadcasted to all subscribed connections*). Das geschieht unabhängig vom verwendeten Transport-Protokoll.

```
@Path("/pubsub/{topic}/{sender}")
@Produces("text/html;charset=ISO-8859-1")
public class WhiteboardPubSub
{
  private @PathParam("topic") Broadcaster topic;
```

```
@GET public SuspendResponse<String> subscribe() {
  return new SuspendResponse.
       SuspendResponseBuilder<String>().
       broadcaster(topic).
       outputComments(true).build();
}

@POST
@Broadcast
public String publish(@FormParam("message") String message,
                      @PathParam("sender") String sender,
                      @Context AtmosphereResource resource) {
  // find current sender in all suspended resources and
  // remove it from the notification
  Collection<AtmosphereResource<?, ?>> ars =
    topic.getAtmosphereResources();
  if (ars == null) {
    return "";
  }

  Set<AtmosphereResource<?, ?>> arsSubset =
    new HashSet<AtmosphereResource<?, ?>>();
  HttpServletRequest curReq = null;
  for (AtmosphereResource ar : ars) {
    Object req = ar.getRequest();
    if (req instanceof HttpServletRequest) {
      String pathInfo =
        ((HttpServletRequest)req).getPathInfo();
      String resSender =
        pathInfo.substring(pathInfo.lastIndexOf('/') + 1);
      if (!sender.equals(resSender)) {
        arsSubset.add(ar);
      } else {
        curReq = (HttpServletRequest) req;
      }
    }
  }

  if (curReq == null) {
    curReq = (HttpServletRequest) resource.getRequest();
  }

  // process current message (JSON) and create
  // a new one (JSON) for subscribed client
  String newMessage = WhiteboardUtils.
```

```
      updateWhiteboardFromJson(curReq, message);

    // broadcast subscribed clients except sender
    topic.broadcast(newMessage, arsSubset);

    return "";
  }
}
```

Die Methode `publish()` sucht den *Sender* (Nachricht-*Publisher*) unter allen *Subscribers* und schließt ihn von der Benachrichtigung aus. Es bleibt jetzt zu kären, wie die Beziehung zwischen den client- und serverseitigen Teilen eingerichtet wird. Dies erfolgt mittels einer Servlet-Konfiguration in `web.xml`.

```
<!-- Faces Servlet -->
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.jsf</url-pattern>
</servlet-mapping>

<!-- Atmosphere Servlet -->
<servlet>
  <description>AtmosphereServlet</description>
  <servlet-name>AtmosphereServlet</servlet-name>
  <servlet-class>org.atmosphere.cpr.AtmosphereServlet</servlet-class>
  <init-param>
    <param-name>
      com.sun.jersey.config.property.resourceConfigClass
    </param-name>
    <param-value>
      com.sun.jersey.api.core.PackagesResourceConfig
    </param-value>
  </init-param>
  <init-param>
    <param-name>com.sun.jersey.config.property.packages</param-name>
    <param-value>com.googlecode.whiteboard.pubsub</param-value>
  </init-param>
  <init-param>
    <param-name>org.atmosphere.useWebSocket</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>org.atmosphere.useNative</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>org.atmosphere.cpr.WebSocketProcessor</param-name>
```

```
    <param-value>
      org.atmosphere.cpr.HttpServletRequestWebSocketProcessor
    </param-value>
  </init-param>
  <init-param>
    <param-name>org.atmosphere.cpr.broadcastFilterClasses</param-name>
    <param-value>
      org.atmosphere.client.JavascriptClientFilter
    </param-value>
  </init-param>
  <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>AtmosphereServlet</servlet-name>
  <url-pattern>*.topic</url-pattern>
</servlet-mapping>
```

Man sieht, dass die `*.jsf` Requests auf den `FacesServlet` und die `*.topic` Requests auf den `AtmosphereServlet` „gemappt“ sind, was der Anwendungsarchitektur entspricht. `AtmosphereServlet` kann umfassend konfiguriert werden. Mit dem Servlet-Konfigurationsparameter `com.sun.jersey.config.property.packages` kann die Klasse `WhiteboardPubSub` gefunden werden. Mit der eingestellten Konfiguration scannt Jersey das Verzeichnis `com.googlecode.whiteboard.pubsub`, in dem diese Klasse liegt.

Zum Schluss sollen noch die JSON-Konvertierungen demonstriert werden. Die Klasse `WhiteboardPubSub` ruft `WhiteboardUtils.updateWhiteboardFromJson()` auf. Die Methode `updateWhiteboardFromJson()` ist `synchronized`, weil darauf mehrere HTTP-Requests gleichzeitig zugreifen können.

```
public static synchronized String updateWhiteboardFromJson(
    HttpServletRequest request, String transferedJsonData)
```

Im ersten Schritt müssen die JSON-Daten nach Java konvertiert werden. Erzeugt wird eine Instanz der Klasse `ClientChangedData`.

```
// create Java object from JSON
ClientChangedData ccd = JsonConverter.getGson().
    fromJson(transferedJsonData, ClientChangedData.class);
```

Die statische Methode `JsonConverter.getGson()` erzeugt eine `Gson`-Instanz, mit der JSON-Konvertierungen gemacht werden können.

```
GsonBuilder gsonBilder = new GsonBuilder();
gsonBilder.serializeNulls();
...
Gson gson = gsonBilder.create();
return gson;
```

Im zweiten Schritt müssen die eingegangenen Daten je nach Aktion behandelt werden. Die

neuen Daten, die für *Subscribers* bestimmt sind, werden in eine Instanz der Klasse `ServerChangedData` gepackt.

```
ServletContext servletContext = (HttpServletRequest)request).
  getSession().getServletContext();
WhiteboardsManager manager = (WhiteboardsManager)
  servletContext.getAttribute("whiteboardsManager");

ServerChangedData scd = null;

switch (ccd.getAction()) {
  case Create:
    scd = WhiteboardUtils.createElement(whiteboard, ccd);
    break;
  case Update:
    scd = WhiteboardUtils.updateElement(whiteboard, ccd);
    break;
  case Remove:
    scd = WhiteboardUtils.removeElement(whiteboard, ccd);
    break;
    ...
  default:
    LOG.warning("Unknown client action!");
    break;
}
```

Anschließend wird das betroffene Whiteboard aktualisiert, den aktuellen Zeitstempel gesetzt und die neue JSON-Ausgabe generiert.

```
// update changed whiteboard
manager.updateWhiteboard(whiteboard);

scd.setTimestamp(ccd.getTimestamp());

// generate output JSON for subscribed clients
return JsonConverter.getGson().toJson(scd);
```

8 Auswertung der Ergebnisse

Die Web-Applikation „Whiteboard“ verfügt über eine nützliche *Logging* Komponente, die eine Auswertung der eingehenden bzw. ausgehenden Nachrichten sowie die verstrichene Zeit zwischen dem Senden und dem Empfangen einer Nachricht ermöglicht. Anhand dieser Komponente kann man somit die Latenzzeit messen. Man öffnet dazu zwei Browserfenster und schaltet das *Logging* per Klick auf den Link „*Toggle logging*“ ein. Die Abbildungen 36 und 37 zeigen dasselbe Whiteboard in zwei unterschiedlichen Browserfenstern.

Bei der Datenerfassung wurden 2 Testszenarien mit jeweils 13 Aktionen herausgearbeitet. Die Testszenarien und Aktionen sind in den Tabellen 1 und 2 aufgeführt. Jede Aktion wurde 5 Mal wiederholt und die Latenzzeit für jede Wiederholung in Millisekunden gemessen. Aus den 5 Messwerten wurde ein Durchschnittswert gebildet, der bei der Diagrammdarstellung verwendet wird. Die Messungen wurden mit zwei modernen Browsern durchgeführt, die das WebSocket-Protokoll bereits unterstützen. Ob ein Browser es unterstützt oder nicht, kann in einem Online-Test geprüft werden [WTURL]. Das erste Testszenario lief in Firefox 6.0.2. Das zweite Testszenario lief in Google Chrome 6.0.472.63. Die Testumgebung war in beiden Fällen ein Linux-System mit OpenSuse 11.4.

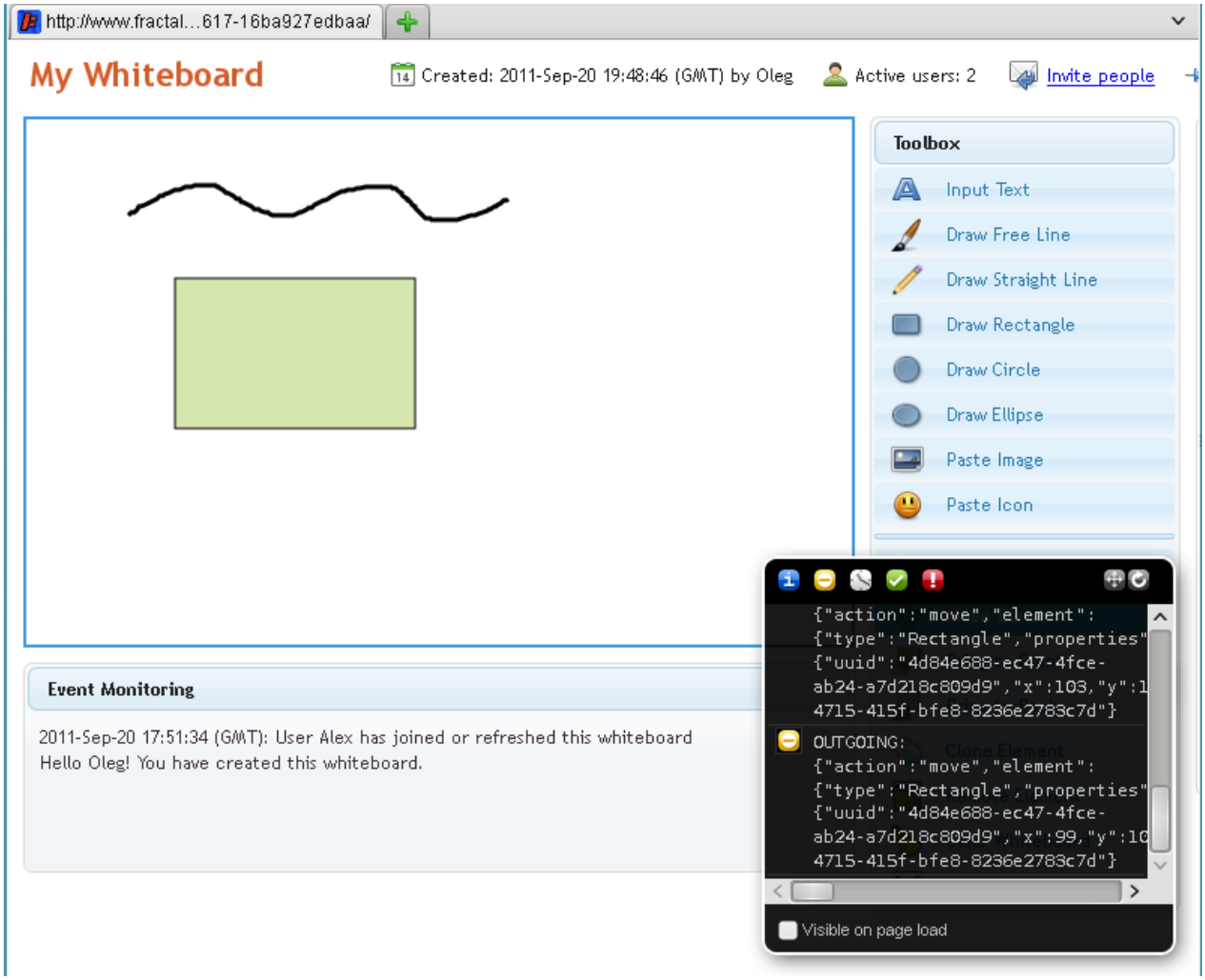

Abbildung 36: Erstes Browserfenster für Logging öffnen

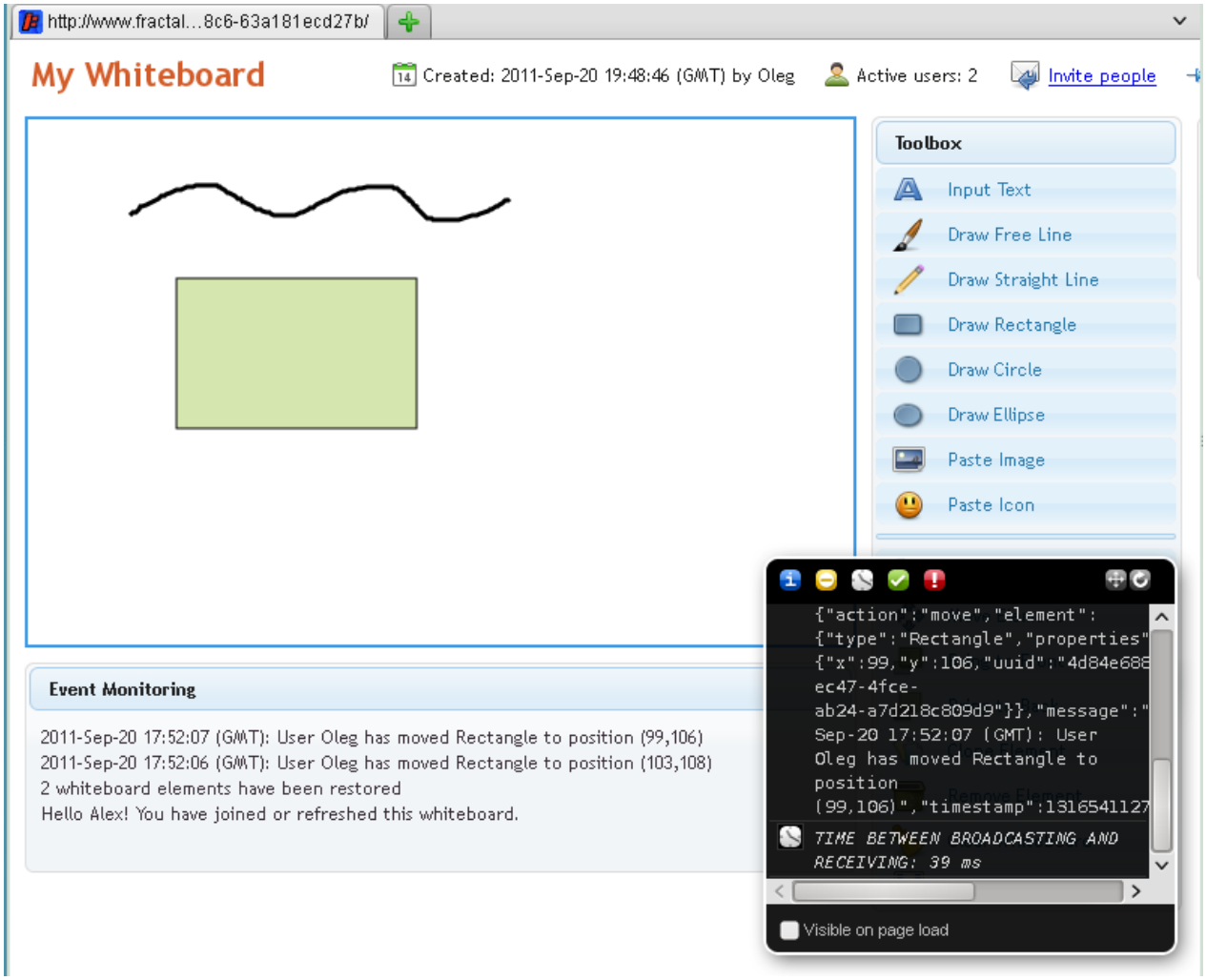

Abbildung 37: Zweites Browserfenster für Logging öffnen

Jeder Testversuch wurde jeweils für ein Transport-Protokoll wiederholt: Long-Polling, Streaming und WebSocket. Von der Projekt-Homepage [OWURL] kann man dazu ausführbare WAR-Dateien (*executable war files*) mit der Whiteboard-Anwendung für jedes genannte Protokoll herunterladen und ausführen lassen. Eine Anleitung dazu wird es im Kapitel 11 geben. Bei der Auswertung der Ergebnisse war es wichtig, eine Tendenz zu erkennen, die zur veränderten Latenzzeit geführt hat.

Die untenstehende Tabelle 1 wurde für die in der ersten Spalte aufgeführten Testszenarien unter Firefox 6.0.2 erfasst. Die letzte Spalte zeigt die Durchschnittszeit in Millisekunden für die fünf Testversuche und jeweils eine Technologie.

		1	2	3	4	5	Ø
1. Join Whiteboard	Long-Polling Streaming WebSocket	15 13 11	12 10 11	12 8 8	10 9 9	10 10 8	11,8 10 9,4
2. Create Text "Hallo Whiteboard"	Long-Polling Streaming	110 108	117 100	114 102	112 105	114 106	113,4 104,2

	WebSocket	101	96	99	105	102	100,6
3. Create Ellipse	Long-Polling Streaming WebSocket	21 20 19	19 17 18	19 17 17	18 18 16	18 17 16	19 17,8 17,2
4. Create Image	Long-Polling Streaming WebSocket	107 104 102	98 96 92	106 97 92	99 99 97	111 98 92	104,2 98,8 95
5. Move Text	Long-Polling Streaming WebSocket	11 9 9	9 10 9	11 9 8	13 10 8	9 9 9	10,6 9,4 8,6
6. Move Ellipse	Long-Polling Streaming WebSocket	11 8 6	10 9 7	13 11 7	10 7 8	10 9 9	10,8 8,8 7,4
7. Change Text (font family)	Long-Polling Streaming WebSocket	118 116 82	110 110 90	113 111 88	119 110 92	113 110 90	114,6 111,4 88,4
8. Change Ellipse (background color)	Long-Polling Streaming WebSocket	16 15 15	15 13 10	15 13 11	20 18 10	14 13 10	16 14,4 11,2
9. Change Image (width)	Long-Polling Streaming WebSocket	15 14 12	15 11 11	18 14 11	15 13 10	13 12 11	15,2 12,8 11
10. Bring Text to Front	Long-Polling Streaming WebSocket	13 9 9	12 9 8	13 11 8	13 12 10	18 11 9	13,8 10,4 8,8
11. Clone Ellipse	Long-Polling Streaming WebSocket	15 15 12	15 13 9	14 11 10	14 13 10	13 13 11	14,2 13 10,4
12. Remove Image	Long-Polling Streaming WebSocket	18 11 9	12 10 10	12 13 10	15 15 11	12 13 9	13,8 12,4 9,8
13. Clear Whiteboard	Long-Polling Streaming WebSocket	24 15 12	18 18 13	23 17 13	21 16 15	18 19 14	20,8 17 13,4

Tabelle 1: Erstes Testszenario für Firefox

In der nächsten Tabelle 2 sind die Testszenarien für Google Chrome 6.0.472.63 aufgeführt. Google Chrome baut auf der Rendering-Engine WebKit und verwendet die JavaScript-Engine V8. V8 ist eine hochperformante Engine und erreicht im Vergleich zu anderen gebräuchlichen Engines eine deutlich überlegene Ausführungsgeschwindigkeit. Aus diesem Grund liegen die Tabellenwerte für Google Chrome deutlich niedriger als beim Firefox.

		1	2	3	4	5	Ø
1. Join Whiteboard	Long-Polling Streaming WebSocket	9 8 6	9 8 7	8 7 6	7 7 7	8 8 7	8,2 7,6 6,6
2. Create Straight Line (800 px width)	Long-Polling Streaming WebSocket	9 7 6	7 8 7	6 7 6	8 6 5	8 6 6	7,6 6,8 6
3. Create Circle	Long-Polling Streaming WebSocket	9 7 5	8 8 8	10 7 7	7 6 6	8 7 5	8,4 7 6,2
4. Create Icon	Long-Polling Streaming WebSocket	8 6 5	6 5 5	5 5 6	6 5 5	6 6 5	6,2 5,4 5,2
5. Move Straight Line	Long-Polling Streaming WebSocket	8 7 7	8 8 6	8 7 6	8 6 6	9 7 7	8,2 7 6,4
6. Move Icon	Long-Polling Streaming WebSocket	9 7 7	10 9 8	11 9 7	9 8 6	9 9 8	9,6 8,4 7,2
7. Change Straight Line (line style)	Long-Polling Streaming WebSocket	11 10 9	8 8 6	9 8 7	8 7 7	7 7 8	8,6 8 7,4
8. Change Circe (position)	Long-Polling Streaming WebSocket	7 5 4	8 6 6	6 5 5	7 4 4	6 5 4	6,8 5 4,6
9. Rotate Icon (15 degree)	Long-Polling Streaming WebSocket	7 5 4	5 4 4	8 6 6	7 7 6	8 7 5	7 5,8 5
10. Bring Straight Line to Back	Long-Polling Streaming WebSocket	9 8 6	8 8 5	8 7 5	7 6 6	6 6 5	7,6 7 5,4
11. Clone Circle	Long-Polling Streaming WebSocket	10 9 7	8 7 6	9 8 5	10 8 6	8 7 6	9 7,8 6
12. Remove Icon	Long-Polling Streaming WebSocket	7 6 5	10 8 5	9 9 7	8 8 7	8 7 6	8,4 7,6 6
13. Resize Whiteboard	Long-Polling Streaming WebSocket	7 7 6	9 8 5	8 6 7	8 7 5	9 8 5	8,2 7,2 5,6

Tabelle 2: Zweites Testszenario für Google Chrome

Auf Basis von diesen Tabellenwerten werden Säulen- und Liniendiagrammen mit den

Durchschnittswerten gezeichnet. Das Säulendiagramm 38 zeigt die gemessene Zeit für alle drei bidirektionalen Technologien im Firefox-Browser.

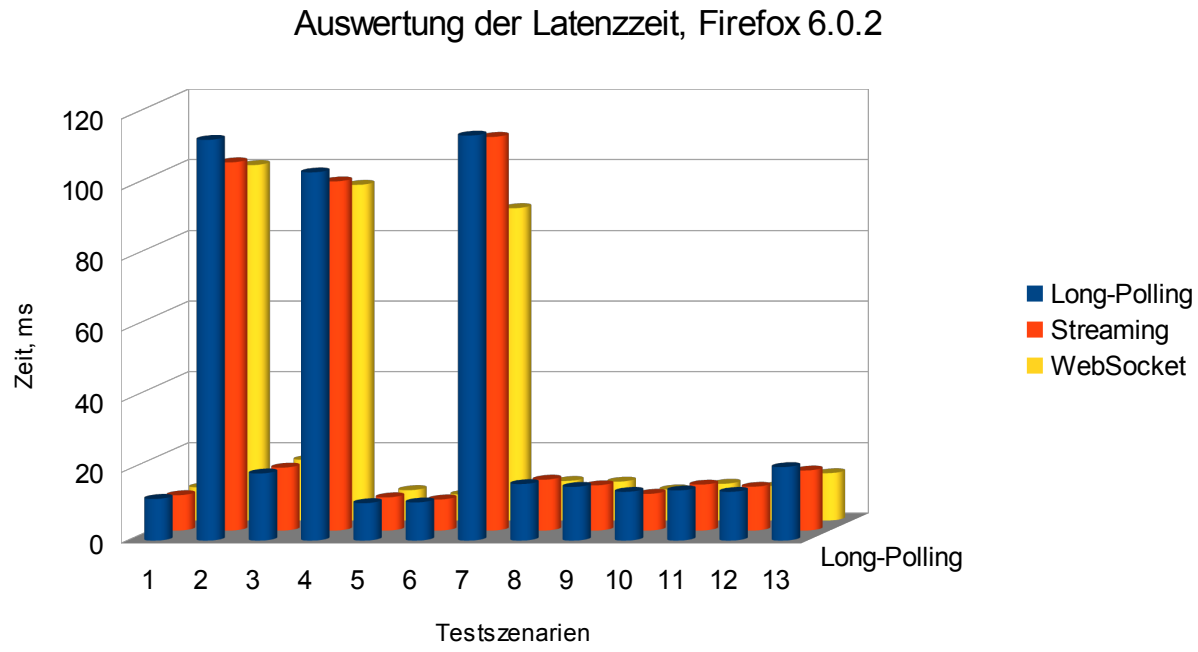

Abbildung 38: Säulendiagramm, Firefox 6.0.2

Die nächste Abbildung 39 zeigt die gleiche Auswertung als Liniendiagramm. Das Liniendiagramm verbindet die Zeitmessungen der einzelnen Testversuche.

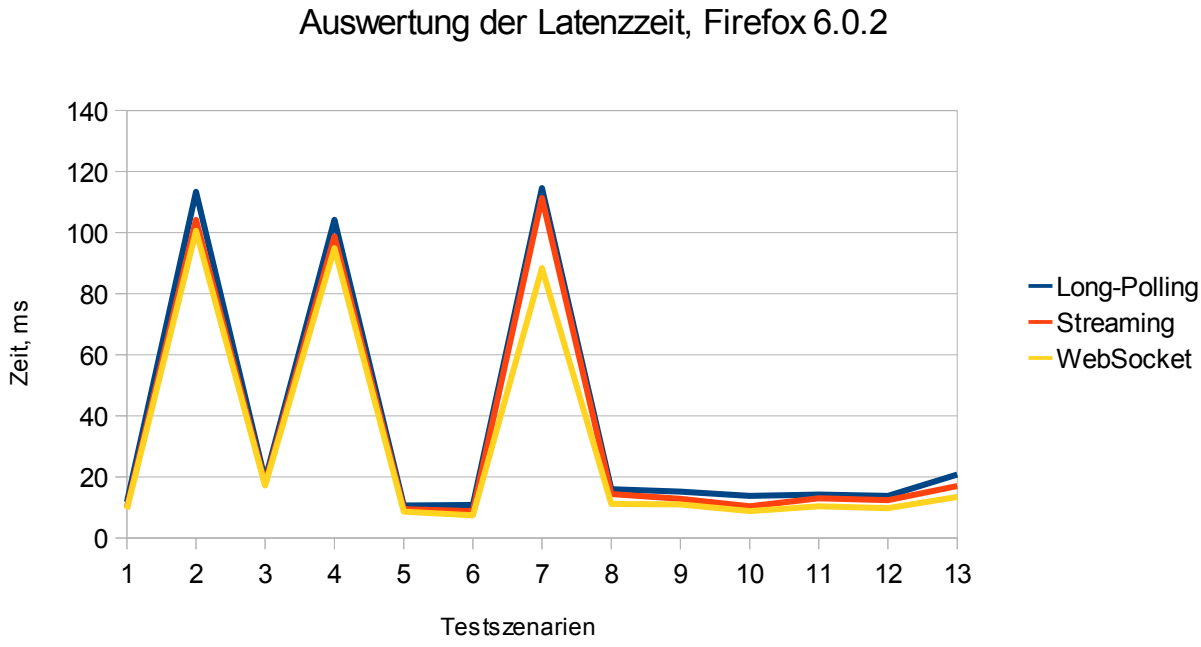

Abbildung 39: Liniendiagramm, Firefox 6.0.2

Das Säulendiagramm 40 zeigt die gemessene Zeit im Google Chrome-Browser.

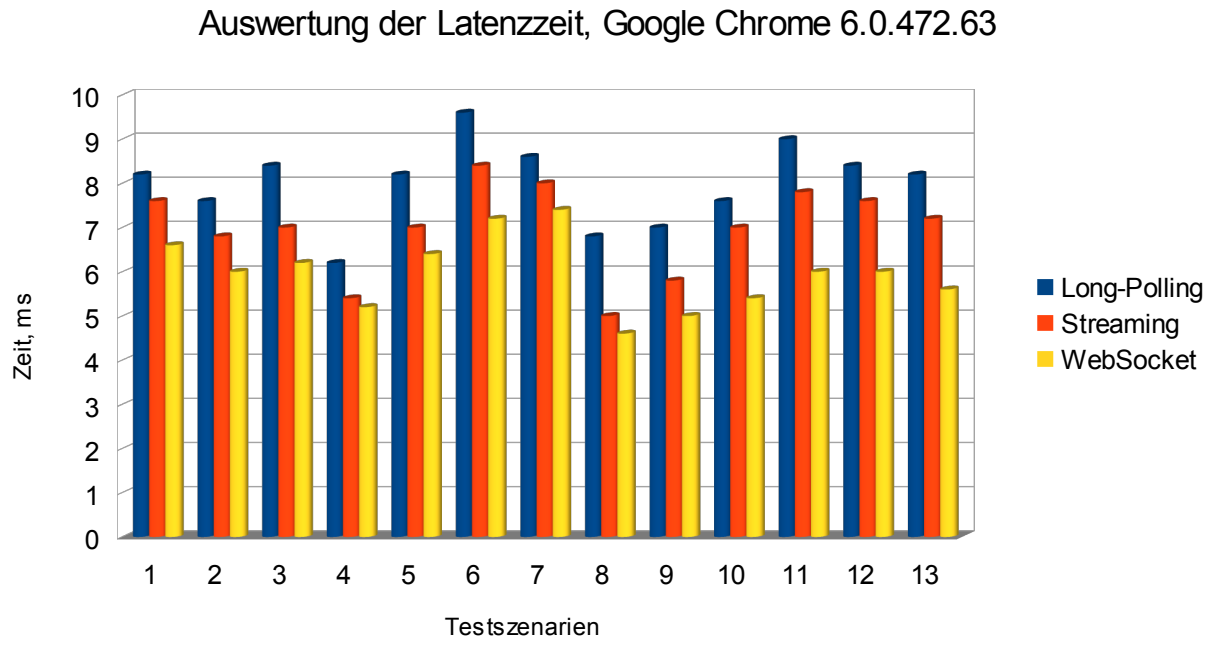

Abbildung 40: Säulendiagramm, Google Chrome 6.0.472.63

Die nächste Abbildung 41 zeigt die gleiche Auswertung als Liniendiagramm.

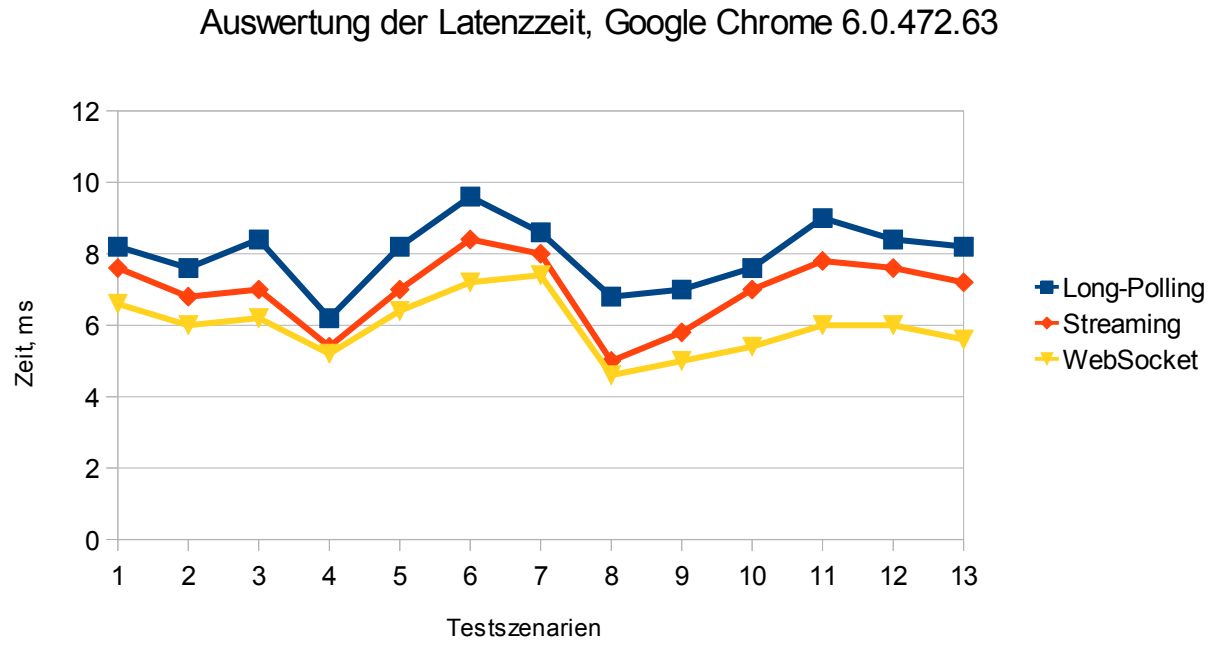

Abbildung 41: Liniendiagramm, Google Chrome 6.0.472.63

Anhand dieser Diagramme kann man eine deutliche "Dominanz" des WebSocket-Protokolls erkennen. Seine Latenzzeit ist die niedrigste. Das ist auf die Effizienz dieses Protokolls zurückzuführen. Das Streaming schneidet besser ab als das Long-Polling. Dieses Ergebnis

war zu erwarten, da beim Streaming keine neuen HTTP Anfragen aufgebaut werden müssen, falls aktualisierte Datenbestände vorliegen. Google Chrome zeigt auch eine deutlichere Tendenz zur besseren Performance beim WebSocket. Es liegt vermutlich daran, dass der Chrome-Browser von Anfang an das WebSocket-Protokoll unterstützt und es konform zur Spezifikation implementiert hat.

Bei der Erfassung der Testergebnisse wurden zudem Datenverluste bei der Long-Polling-Variante festgestellt. Im Unterkapitel 5.3.2 wurde bereits diese Schwäche des Long-Pollings erwähnt. Liegen neue oder aktualisierte Daten zwischen einer Response und einem darauf folgenden GET-Request (zum Abhören von Änderungen) vor, können sie verloren gehen. Inkrementiert der Benutzer z.B. sehr schnell den Drehungswinkel eines Elementes (mit der Maus) oder ändert sehr schnell die Schriftgröße eines Textes, kann es unter Umständen zu Datenverlusten kommen. Das Atmosphere-Framework hat dafür eine Lösung parat. Man kann dort einen serverseitigen Cache für alle vorliegenden Daten einrichten. Dafür ist der folgende Konfigurationsparameter für `AtmosphereServlet` in `web.xml` einzutragen.

```
<init-param>
  <param-name>
    org.atmosphere.cpr.broadcasterCacheClass
  </param-name>
  <param-value>
    org.atmosphere.cache.HeaderBroadcasterCache
  </param-value>
</init-param>
```

Mit diesen Einstellungen werden die Nachrichten an *Topic*-Abonnenten garantiert zugestellt.

9 Zusammenfassung und Ausblick

In der vorliegenden Diplomarbeit wurden die webbasierten Kommunikationstechnologien im Client-Server-Umfeld untersucht. Es wurden zuerst das HTTP-Protokoll, synchroner / asynchroner Datenaustausch und AJAX vorgestellt. Der Schwerpunkt der Arbeit lag in der Untersuchung und Bewertung der wichtigsten Technologien für die bidirektionale Kommunikation. Insgesamt wurden zehn bidirektionale Kommunikationstechnologien ausführlich behandelt. Anhand der herausgearbeiteten Vergleichskriterien fand ein Vergleich der vorgestellten Technologien statt und es wurden deren Vor- und Nachteile aufgezeigt. Die behandelten Technologien mit allen elf Vergleichskriterien und deren Beurteilungen wurden in einer Übersichtstabelle zusammengefasst (Unterkapitel 5.4). Die Übersichtstabelle soll die Auswahl einer bestimmten Technologie erleichtern. In der Zusammenfassung zum Kapitel 5 wurde die Antwort auf die Leitfrage, welche der Technologien für welchen Einsatzzweck geeignet sind und worin sich diese Technologien unterscheiden, gegeben.

Im praktischen Teil der Diplomarbeit wurde ein asynchrones, kollaboratives Whiteboard modelliert und entwickelt. Es fand eine umfassende Darstellung aller Funktionalitäten und der Gesamtarchitektur statt. Es wurden sowohl client- als auch serverseitige Teile der Web-Anwendung implementiert. Zahlreiche Codeauszüge demonstrieren wichtige Schritte bei der Umsetzung der Whiteboard-Anwendung. Auf das Atmosphere-Framework, mit dem das Whiteboard implementiert wurde, wurde verstärkt eingegangen. Außer Atmosphere fand auch die Vorstellung von anderen Frameworks und Engines für die praktische Umsetzung auf Basis der behandelten Technologien statt. Die fertiggestellte Web-Anwendung wurde auf einem Jetty 8 Server mit der WebSocket-Unterstützung deployet. Außerdem wurden drei ausführbare WAR-Dateien mit der Web-Anwendung und jeweils einer Technologie - Long-Polling, Streaming, WebSocket - zur Verfügung gestellt. Alle Links sind auf der Projekt-Homepage zu finden: http://code.google.com/p/online-whiteboard/ Die Auswertung der Messdaten für die Beispielanwendung schließt diese Arbeit ab. Man erkennt eine deutliche Überlegenheit des WebSocket-Protokolls gegenüber Comet-Techniken.

Es stellt sich die Frage über die Zukunft der bidirektionalen (Server-Push) Technologien. Ohne Zweifel werden sie weiterentwickelt; denn für deren Einsatz im Internet / Intranet gibt es zahlreiche sinnvolle und interessante Möglichkeiten. Ein Gebiet, für das sich der Einsatz anbietet, sind kollaborative Multi-Benutzer-Anwendungen, bei denen mehrere Nutzer in Echtzeit zusammenarbeiten können. Beispiele dazu sind Google Talk / Docs / Wave, Zoho Office oder Meebo. Verschiedene Nutzer können zeitgleich an einem Dokument schreiben, in Echtzeit Bilder zeichnen, Diagramme erstellen, Zeit- und Ressourcenpläne verwalten, chatten oder auch spielen. Lernen und Lehren mit webbasierten Anwendungen durch den Einsatz kollaborativer Werkzeuge ist noch ein Beispiel dazu. Ein weiteres Gebiet sind Anwendungen mit sehr aktuellen und veränderlichen Daten. Dazu zählen beispielsweise Darstellung von Börsenkursen, Überwachung von diversen Systemwerten, Newstickers und Liveübertragungen von Fußballergebnissen. Es lassen sich viele weitere Anwendungsfälle finden [COURL], [BIS06].

Durch die Vielfalt der Einsatzgebiete ist es offensichtlich, dass der Einsatz dieser oder jener Technologie damit durch den Zweck der Software und die Rahmenbedingungen bedingt ist.

Muss eine *Instant Messaging* Software entwickelt werden, ist es anzuraten, sich an das BOSH-Protokoll zu wenden. Muss die Anwendung auf der GAE-Plattform laufen, können nur noch Google Channel API benutzt werden. Der Einsatz (und zugleich auch Erfolg) jeder Technologie hängt auch mit technischen Fortschritten zusammen. Einige Technologien, wie beispielsweise das Reverse HTTP-Protokoll, können aus technischen Gründen gar nicht eingesetzt werden. Die anderen, wie das WebSocket-Protokoll, werden noch nicht bei allen Servern und Browsern unterstützt. Es wurde auch berichtet, dass in den ersten Versionen des WebSocket-Designs eine Sicherheitslücke entdeckt wurde. Aus diesem Grund mussten die Browser Firefox und Opera die WebSocket-Unterstützung sogar abstellen [WDURL]. Nachdem eine neue, verbesserte Version des Protokolls verabschiedet wurde, ist die WebSocket-Unterstützung in diesen Browsern wieder aktiviert. Der WebSocket-Standard findet derzeit dank seiner Effizienz eine zunehmende Anwendung. Es ist zu erwarten, dass die WebSocket-Technologie den Markt bald dominieren wird. Bis dahin empfiehlt sich auf die bewährten Comet-Techniken auszuweichen, falls keine speziellen Anforderungen und Rahmenbedingungen an die Software gestellt werden.

10 Literaturverzeichnis

[AAD08] Ali Mesbah, Arie van Deursen. A component- and push-based architectural style for ajax applications.
Journal of Systems and Software, Volume 81 Issue 12, December, 2008.

[AFURL] AJAX Frameworks.
http://ajaxpatterns.org/Ajax_Frameworks

[APURL] Ajax Push Engine (APE).
http://www.ape-project.org/home.html

[ATURL] Atmosphere - a portable AjaxPush/Comet and WebSocket Framework.
http://atmosphere.java.net

[AWURL] Asynchrones Web.
http://www.theserverside.com/news/1363576/What-is-the-Asynchronous-Web-and-How-is-it-Revolutionary

[BER05] Olaf Bergmann, Carsten Bormann. AJAX - Frische Ansätze für das Web-Design.
Teia AG; Auflage: 1 (10. Januar 2005), ISBN: 978-3935539265.

[BEURL] The Bayeux Specification, Bayeux Protocol 1.0.0.
http://svn.cometd.com/trunk/bayeux/bayeux.html

[BIS06] Andreas Bischoff. Virtual Reality und Streaming-Technologien in der Web-basierten multimedialen Lehre und für Ubiquitous Computing.
Books on Demand Gmbh; Auflage: 1 (Oktober 2006),
ISBN: 978-3833460661.

[BIURL] Bidirectional-streams Over Synchronous HTTP (BOSH).
http://xmpp.org/extensions/xep-0124.html

[BLURL] BlazeDS - the server-based Java remoting and web messaging technology.
http://opensource.adobe.com/wiki/display/blazeds/BlazeDS

[BRU10] Building the Realtime User Experience: Creating Immersive and Interactive Websites.
O'Reilly Media; 1 edition (July 7, 2010), ISBN: 978-0596806156.

[BUR10] Ed Burns, Neil Griffin. JavaServer Faces 2.0. The Complete Reference.
Mcgraw-Hill Professional; Auflage: 1 (1. Februar 2010),
ISBN: 978-0071625098.

[CAURL] Collaborative Whiteboard - Channel API and Comet.
http://blog.azprogrammer.com/2010/12/collaborative-whiteboard-channel-api.html

[CDURL] CometD – reference implementation of the Bayeux protocol.
http://cometd.org

[COURL] Comet and the Rise of Highly Interactive Websites.
http://www.slideshare.net/joewalker/comet-and-the-rise-of-highly-interactive-websites-presentation

[CRA07] Dave Crane, Bear Bibeault, Jord Sonneveld. Ajax in practice:
Das Praxisbuch für die Web 2.0-Entwicklung.
Addison-Wesley, München; Auflage: 1 (28. November 2007),
ISBN: 978-3827325969.

[CRA08] Dave Crane, Phil McCarthy. Comet and Reverse Ajax:

The Next-Generation Ajax 2.0 (Firstpress).
Apress; Auflage: New. (7. Oktober 2008), ISBN: 978-1590599983.

[DEURL] Ajax-on-Demand.
https://docs.google.com/View?docid=ddmtrs6k_10hnrmff

[DIURL] Direct Web Remoting (DWR).
http://directwebremoting.org/dwr/index.html

[EBA07] E. Bozdag. Integration of HTTP push with a JSF Ajax framework.
Master's thesis, Delft University of Technology, December 2007.

[EBG07] E. Bozdag, A. Mesbah, and A. van Deursen. A comparison of push and pull techniques for Ajax.
In Proceedings of the 9th IEEE International Symposium on Web Site Evolution (WSE'07), pages 15–22. IEEE Computer Society, 2007.

[EPURL] Extensible Messaging and Presence Protocol (XMPP) .
http://xmpp.org/about/

[ESC09] Rainer Eschen. Icefaces 1.8: Next Generation Enterprise Web Development.
Packt Publishing (2. November 2009), ISBN: 978-1847197245.

[EXURL] Exploring Reverse Ajax.
http://gmapsdotnetcontrol.blogspot.com/2006/08/exploring-reverse-ajax-ajax.html

[FRE10] Frederic P. Miller, Agnes F. Vandome, John McBrewster. Client-server: Server (computing), Computer Network, Inter-server, Mobile Software,
Push Technology, Pull Technology, Servent, Thin Client.
Alphascript Publishing (Februar 2010), ISBN: 978-6130631482.

[GAURL] Google App Engine, Channel API Overview.
http://code.google.com/intl/de/appengine/docs/python/channel/overview.html

[GAE10] Goolge App Engine. Eine genaue Betrachtung der GAE.
Florian Hartl, Galina Koleva, Maren Steinkamp. Masterpraktikum "Cloud Computing" SS2010, TU München, Fakultät für Informatik.

[GEURL] Google App Engine.
http://de.wikipedia.org/wiki/Google_App_Engine

[GMURL] Google Mail (gmail)
http://gmail.com

[GNURL] Google-gson - Java library to convert JSON to Java objects and vice-versa.
http://code.google.com/p/google-gson/

[GOY02] David Gourley, Brian Totty. HTTP: The Definitive Guide.
O'Reilly Media; Auflage: 1 (4. Oktober 2002), ISBN: 978-1565925090.

[GRO07] Christian Gross. Ajax Design Patterns und Best Practices.
Mitp-Verlag; Auflage: 1 (2007), ISBN: 978-3826616921.

[GRU10] Bcher Gruppe, Bucher Gruppe. Xmpp: Extensible Messaging and Presence Protocol, Liste Von Xmpp-Clients, Multi-User Chat, Xmpp-Transport, Jabber Identifier.
Books Llc (22. Juli 2010), ISBN: 978-1159356491.

[GSURL] Skalierbarkeit der Google App Engine.
http://wiki.fh-stralsund.de/index.php/Skalierbarkeit_der_Google_App_Engine

[ICURL] ICEPush Framework.
http://www.icepush.org

[JEURL] Jersey - Java API for RESTful Web Services.
http://jersey.java.net/
[JFURL] JavaServer Faces 2, Mojarra Projekt.
https://javaserverfaces.dev.java.net
[JQURL] JavaScript-Framework jQuery.
http://jquery.com/
[JNURL] JavaScript Object Notation (JSON).
http://www.json.org
[JSURL] JSR 315: Java Servlet 3.0 Specification
http://jcp.org/en/jsr/detail?id=315
[JUURL] jQuery UI Library
http://jqueryui.com/
[JWURL] jWebSocket.
http://jwebsocket.org
[KAURL] Kaazing Platform.
http://www.kaazing.com/
[LIURL] Lightstreamer.
http://www.lightstreamer.com
[LLC10] LLC Books. World Wide Web Introduction: Link Topology, Off Topic, C-HTML, Link Awareness, Ecologyfund.Com, Meta-Moderation System, Web Sockets.
Books Llc (31. Mai 2010), ISBN: 978-1157521358.
[LUB10] Peter Lubbers, Brian Albers, Frank Salem. Pro Html5 Programming: Powerful APIs for Richer Internet Application Development.
Apress; Auflage: New. (27. August 2010), ISBN: 978-1430227908.
[MAR09] Martin Marinschek, Michael Kurz, Gerald Müllan. JavaServer Faces 2.0: Grundlagen und erweiterte Konzepte.
dpunkt Verlag; Auflage: 2., (1. November 2009), ISBN: 978-3898646062.
[MAURL] Maven - Apache build manager for Java projects.
http://maven.apache.org/
[MIP09] Mikko Pohja. Server push with instant messaging.
SAC '09 Proceedings of the 2009 ACM symposium on Applied Computing.
[MOS09] Christiane Moser. Interaktionsparadigma - Web 2.0: Blogs, Tags, RSS, Social Bookmarks, Podcasts, Mash-ups, AJAX.
VDM Verlag Dr. Müller (8. Dezember 2009), ISBN: 978-3639212525.
[MUN10] Stefan Münz, Clemens Gull. HTML 5 Handbuch.
Franzis Verlag GmbH (15. November 2010), ISBN: 978-3645600798.
[NAURL] Ajax: A New Approach to Web Applications.
http://www.adaptivepath.com/ideas/essays/archives/000385.php
[NEURL] New Adventures in Comet: polling, long polling or Http streaming with AJAX. Which one to choose?
http://jfarcand.wordpress.com/2007/05/15/new-adventures-in-comet-polling-long-polling-or-http-streaming-with-ajax-which-one-to-choose
[NIURL] New I/O Java API.
http://download.oracle.com/javase/1.4.2/docs/guide/nio/
[OGG10] Bernd Öggl, Klaus Förster. HTML5: Leitfaden für Webentwickler.

Addison-Wesley, München; Auflage: 1 (9. November 2010),
ISBN: 978-3827328915.

[OAURL] OpenAjax Alliance.
http://www.openajax.org

[OPURL] OpenAjax – Simple Protokoll für Ajax Push.
http://www.openajax.org/member/wiki/Simple_protocol_for_Ajax_Push

[OYURL] Optimizing Asynchronous Communications for Scalability.
http://icefaces.org/docs/latest/htmlguide/devguide/AdvancedTopics7.html

[OWURL] Homepage des Projektes “Collaborative Online-Whiteboard“.
http://code.google.com/p/online-whiteboard/

[PBURL] PaintWeb – Drawing inside web browsers by using of HTML5 Canvas API.
http://code.google.com/p/paintweb/

[PFURL] PrimeFaces, JSF Komponentenbibliothek.
http://www.primefaces.org

[PPURL] Peer-to-Peer, Wikipedia.
http://de.wikipedia.org/wiki/Peer-to-Peer

[ROC09] Christof Rodejohann. Diplomarbeit “Dynamische Auswahl von HTTP-Interaktionsmechanismen”.
Technische Universität Dresden, Fakultät Informatik, 2009.

[QRY09] Quan Zhou, Ruixiang Bian, Yuchun Pan. Design of Electric Power Web System Based on Comet.
ICICTA, vol. 3, pp.42-45, 2009. Second International Conference on Intelligent Computation Technology and Automation, 2009.

[REURL] Reverse HTTP.
http://wiki.secondlife.com/wiki/Reverse_HTTP

[RLURL] Raphaël - JavaScript library for vector graphics on the web.
http://raphaeljs.com/

[RTURL] ICEpush for Real-time Web Collaboration.
http://java.dzone.com/articles/icepush-real-time-web

[SAL08] Sami Salkosuo. DWR Java Ajax Applications.
Packt Publishing (29. Oktober 2008), ISBN: 978-1847192936.

[SCURL] HTTP Status Code Definitions.
http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

[SEURL] Server Push Comet.
http://www.slideshare.net/mayflowergmbh/server-push-comet

[SIURL] Comet Server Implementationen.
http://cometdaily.com/maturity.html

[SRURL] Strophe library for XMPP / BOSH
http://strophe.im

[SSURL] Server-Sent Events
http://dev.w3.org/html5/eventsource/

[STURL] SVG-edit - Web-based, Javascript-driven SVG editor.
http://code.google.com/p/svg-edit/

[SUURL] Subversion Clients.
https://svn.tugraz.at/clients.php

[SVURL] Servlet, Wikipedia.

http://de.wikipedia.org/wiki/Servlet
[VPURL] jQuery Validation Plugin.
http://docs.jquery.com/Plugins/Validation
[WAURL] WebSocket API.
http://dev.w3.org/html5/websockets
[WDURL] WebSocket disabled in Firefox 4.
http://hacks.mozilla.org/2010/12/websockets-disabled-in-firefox-4/
[WEI08] Stefanie Weiser. Web 2.0: Strategien, Anwendungen und Technologien.
VDM Verlag Dr. Müller (Juni 2008) , ISBN: 978-3639029048.
[WEURL] Wikipedia. Extensible messaging and presence protocol.
http://de.wikipedia.org/wiki/Extensible_Messaging_and_Presence_Protocol.
[WON00] Clinton Wong. HTTP, kurz & gut.
O'Reilly (2000), ISBN: 978-3897212305.
[WPURL] How HTML5 Web Sockets Interact With Proxy Servers.
http://www.infoq.com/articles/Web-Sockets-Proxy-Servers
[WSURL] What is WebSocket?
http://websocket.org
[WTURL] Real-Time Web Test – Does HTML5 WebSockets work for your?
http://websocketstest.com/
[XLURL] xLightweb.
http://xlightweb.sourceforge.net
[XMURL] XMPP Is Better With BOSH.
http://metajack.wordpress.com/2008/07/02/xmpp-is-better-with-bosh
[XRURL] XMLHttpRequest.
http://www.w3.org/TR/XMLHttpRequest
[ZAM10] Frank Zammetti. Practical DWR 2 Projects (Expert's Voice in Java).
Apress; Auflage: 1 (2. Juni 2010), ISBN: 978-1590599419.

11 Projekt-Quellcode

Dieses Kapitel beschreibt die Projektstruktur der Beispielanwendung „Whiteboard“ und gibt eine Anleitung zur Ausführung der Web-Applikation. Das Projekt wird als OpenSource Software (Apache License V2) auf Google Code mit dem Versionsverwaltungssystem Subversion (SVN) gehostet: http://code.google.com/p/online-whiteboard/ [OWURL]. Der Quellcode des Projektes kann von der Homepage ausgecheckt werden. Zum Auschecken kann jeder SVN-Client benutzt werden [SUURL]. Mit dem SVN Command-Line-Client lässt sich der Projekt-Quellcode wie folgt auschecken:

```
svn checkout http://online-
whiteboard.googlecode.com/svn/trunk/ online-whiteboard
```

Danach liegt das Project unter dem Verzeichnis `online-whiteboard`. Das Projekt ist mit einem Jetty-Maven Plugin vorkonfiguriert, so dass sich die Web-Applikation direkt nach dem Auschecken des Quellcodes starten lässt. Man geht dazu in das Wurzel-Verzeichnis des ausgecheckten Projektes (es heißt in unserem Fall `online-whiteboard`), öffnet die DOS-Console oder das Linux-Terminal (je nach Betriebssystem) und ruft man auf

```
mvn jetty:run
```

Mit diesem Befehl wird der *embedded* Jetty-Server gestartet. Vorausgesetzt ist, dass das Maven-Tool auf dem Rechner installiert ist. Maven ist ein Build-Management-Tool der *Apache Software Foundation* [MAURL]. Wenn der Jetty-Server läuft, kann die Web-Applikation unter der Adresse

```
http://localhost:8080/whiteboard-showcase
```

erreicht werden. Vorkonfiguriert ist das WebSocket-Protokoll. Die Beispielanwendung mit dem WebSocket-Protokoll wurde auch online gestellt. Sie läuft auf einem Jetty 8 Server. Die Web-Anwendung kann unter dem folgenden direkten Link erreicht werden:

http://www.fractalsoft.net/whiteboard-websocket/

Des Weiteren wurden ausführbare WAR-Dateien für alle drei Protokolle, Long-Polling, Streaming und WebSocket, zur Verfügung gestellt. Sie können genauso von der Projekt-Homepage heruntergeladen werden. Nach dem Herunterladen kann jede WAR-Datei unter Java gestartet werden. Man geht dazu in das Verzeichnis, in dem die Datei liegt, öffnet die DOS-Console oder das Linux-Terminal und tippt man ein (hier ein Beispiel für die Long-Polling-Variante der Beispielanwendung):

```
java -jar whiteboard-long-polling.war
```

Daraufhin startet eine kleine Java-GUI mit den Buttons „*Start*“ und „*Exit*“. Wird der Button „*Start*“ betätigt, startet automatisch der als Standard eingestellte Browser, in dem dann die Web-Applikation läuft.

Das Web-Projekt ist ein typisches Maven-basiertes JSF-Projekt, dessen Struktur in der Abbildung 42 demonstriert wird.

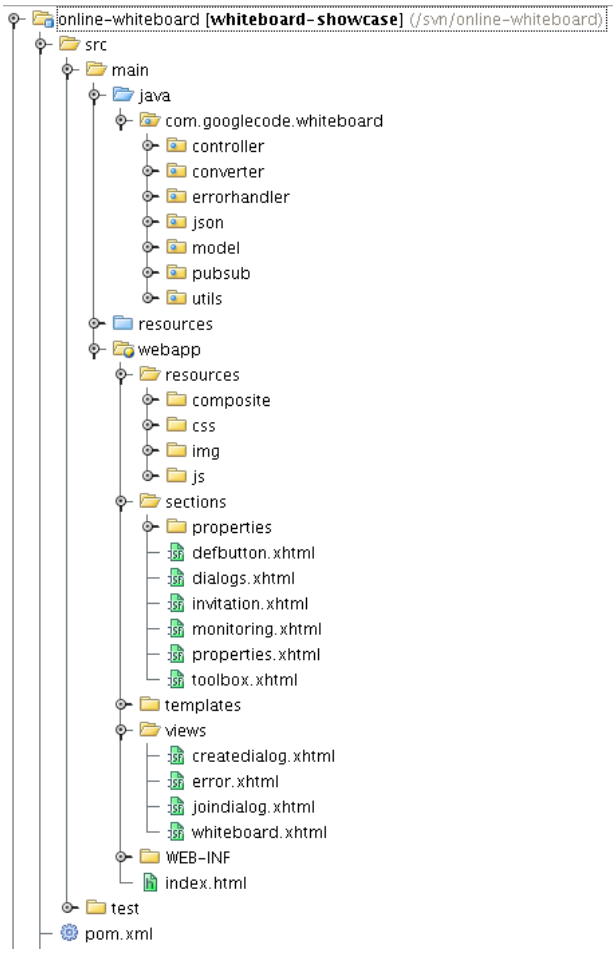

Abbildung 42: Projektstruktur

In dem Verzeichnis `src/main/java` (Package `com.googlecode.whiteboard`) befinden sich Java-Klassen. Im Package `controller` befinden sich JSF *Managed Beans*. Im Package `errorhandler` sind Klassen untergebracht, die für die Fehlerbehandlung zuständig sind. Im Package `json` liegen die Gson-Klassen für die JSON-Konvertierungen. Das Package `model` beinhaltet die im Unterkapitel 7.4 vorgestellten Modellklassen. Im Package `pubsub` befinden sich Klassen für die bidirektionale Kommunikation mit Atmosphere (unter anderem `WhiteboardPubSub` aus dem Kapitel 7.5). Das Package `utils` enthält die Hilfsklassen. Im Verzeichnis `src/main/webapp` liegen diverse Web-Ressourcen, wie CSS-, JavaScript-Dateien, Bilder, und die eigentlichen Web-Seiten in Form von JSF *Facelets* (XHTML-Seiten). Der Ablageort einzelner Web-Seiten ist gut strukturiert. Komplette JSF-Views liegen z.B. unter `src/main/webapp/views` und Seitenfragmente unter `src/main/webapp/sections`. XML-Konfigurationsdateien liegen, wie üblich für Web-Anwendungen, unter `src/main/webapp/WEB-INF`. Das sind z.B. `faces-config.xml` und `web.xml`. Direkt im Projekt-Wurzel-Verzeichnis liegt die Datei `pom.xml`. Das ist eine Maven-Konfigurationsdatei, mit der das gesamte Projekt gebaut wird. Auf der Projekt-Homepage befindet sich auch die Online-Dokumentation des Quellcodes (JavaDoc und JSDoc).

12 Anhang A. Aktionsspezifische JSON-Strukturen

- Aktion „Join“.

```
{ "action" : "join",
  "parameters" : { "usersCount" : 2 },
  "timestamp" : 1315772921120,
  "user" : "Alex",
  "whiteboardId" : "8b10da44-30bd-4872-9961-8d821dcd4daf"
}

{ "action" : "join",
  "element" : null,
  "message" : "2011-Sep-11 22:28:41 (GMT): User Alex has joined or refreshed
              this whiteboard",
  "parameters" : { "usersCount" : "2" },
  "timestamp" : 1315772921120
}
```

- Aktion „Create“.

```
{ "action" : "create",
  "element" : { "properties" : {
          "color" : "#000000",
          "fontFamily" : "Verdana",
          "fontSize" : 18,
          "fontStyle" : "normal",
          "fontWeight" : "normal",
          "rotationDegree" : 0,
          "text" : "Hello Whiteboard!",
          "uuid" : "a311517b-d8d3-43c7-be13-46a26b94dda1",
          "x" : 253,
          "y" : 129
        },
      "type" : "Text"
    },
  "timestamp" : 1315773850698,
  "user" : "Alex",
  "whiteboardId" : "8b10da44-30bd-4872-9961-8d821dcd4daf"
}

{ "action" : "create",
  "element" : { "properties" : {
          "color" : "#000000",
          "fontFamily" : "Verdana",
          "fontSize" : 18,
          "fontStyle" : "normal",
          "fontWeight" : "normal",
          "rotationDegree" : 0,
          "text" : "Hello Whiteboard!",
          "uuid" : "a311517b-d8d3-43c7-be13-46a26b94dda1",
          "x" : 253,
          "y" : 129
        },
      "type" : "Text"
    },
  "message" : "2011-Sep-11 22:44:10 (GMT): User Alex has created Text 'Hello
              Whiteboard!' at position (253,129)",
  "parameters" : null,
```

```
  "timestamp" : 1315773850698
}
```

- Aktion „Update“.

```
{ "action" : "update",
  "element" : { "properties" : {
          "backgroundColor" : "rgb(128, 0, 70)",
          "backgroundOpacity" : 1,
          "borderColor" : "rgb(0, 0, 0)",
          "borderOpacity" : 1,
          "borderStyle" : "No",
          "borderWidth" : 1,
          "radius" : 70,
          "rotationDegree" : 0,
          "uuid" : "81a08476-ee0e-4096-ad1d-6de85af19e7c",
          "x" : 490,
          "y" : 198
        },
      "type" : "Circle"
    },
  "timestamp" : 1315774186925,
  "user" : "Oleg",
  "whiteboardId" : "8b10da44-30bd-4872-9961-8d821dcd4daf"
}

{ "action" : "update",
  "element" : { "properties" : {
          "backgroundColor" : "rgb(128, 0, 70)",
          "backgroundOpacity" : 1.0,
          "borderColor" : "rgb(0, 0, 0)",
          "borderOpacity" : 1.0,
          "borderStyle" : "No",
          "borderWidth" : 1,
          "radius" : 70,
          "rotationDegree" : 0,
          "uuid" : "81a08476-ee0e-4096-ad1d-6de85af19e7c",
          "x" : 490,
          "y" : 198
        },
      "type" : "Circle"
    },
  "message" : "2011-Sep-11 22:49:46 (GMT): User Oleg has updated properties of
              Circle at current position (490,198)",
  "parameters" : null,
  "timestamp" : 1315774186925
}
```

- Aktion „Remove“.

```
{ "action" : "remove",
  "element" : { "properties" : {
      "uuid" : "81a08476-ee0e-4096-ad1d-6de85af19e7c" },
      "type" : "Circle"
    },
  "timestamp" : 1315774389052,
  "user" : "Alex",
  "whiteboardId" : "8b10da44-30bd-4872-9961-8d821dcd4daf"
}
```

```
{ "action" : "remove",
  "element" : { "properties" : {
      "uuid" : "81a08476-ee0e-4096-ad1d-6de85af19e7c" },
      "type" : "Circle"
    },
  "message" : "2011-Sep-11 22:53:09 (GMT): User Alex has removed Circle at
              position (490,198)",
  "parameters" : null,
  "timestamp" : 1315774389052
}
```

- Aktion „Clone“.

```
{ "action" : "clone",
  "element" : { "properties" : {
          "name" : "gear",
          "rotationDegree" : 0,
          "scaleFactor" : 1,
          "uuid" : "28f1ac1e-a3a8-4dcb-a83b-cc8df9cd1195",
          "x" : 356,
          "y" : 238
        },
      "type" : "Icon"
    },
  "timestamp" : 1315774599369,
  "user" : "Oleg",
  "whiteboardId" : "8b10da44-30bd-4872-9961-8d821dcd4daf"
}

{ "action" : "clone",
  "element" : { "properties" : {
          "name" : "gear",
          "rotationDegree" : 0,
          "scaleFactor" : 1.0,
          "uuid" : "28f1ac1e-a3a8-4dcb-a83b-cc8df9cd1195",
          "x" : 356,
          "y" : 238
        },
      "type" : "Icon"
    },
  "message" : "2011-Sep-11 22:56:39 (GMT): User Oleg has cloned Icon at
              position (356,238)",
  "parameters" : null,
  "timestamp" : 1315774599369
}
```

- Aktion „Move“.

```
{ "action" : "move",
  "element" : { "properties" : {
          "uuid" : "ecaf4113-3d3c-43dd-86bd-19d41fb26084",
          "x" : 595,
          "y" : 167
        },
      "type" : "Ellipse"
    },
  "timestamp" : 1315774753540,
  "user" : "Oleg",
```

```
  "whiteboardId" : "8b10da44-30bd-4872-9961-8d821dcd4daf"
}

{ "action" : "move",
  "element" : { "properties" : {
          "uuid" : "ecaf4113-3d3c-43dd-86bd-19d41fb26084",
          "x" : 595,
          "y" : 167
        },
      "type" : "Ellipse"
    },
  "message" : "2011-Sep-11 22:59:13 (GMT): User Oleg has moved Ellipse to
              position (595,167)",
  "parameters" : null,
  "timestamp" : 1315774753540
}
```

- Aktion „ToFront“.

```
{ "action" : "toFront",
  "element" : { "properties" : {
      "uuid" : "d4a91ff2-8c65-4f84-90e0-a9c2fd36e459" },
      "type" : "FreeLine"
    },
  "timestamp" : 1315774935536,
  "user" : "Alex",
  "whiteboardId" : "8b10da44-30bd-4872-9961-8d821dcd4daf"
}

{ "action" : "toFront",
  "element" : { "properties" : {
      "uuid" : "d4a91ff2-8c65-4f84-90e0-a9c2fd36e459" },
      "type" : "FreeLine"
    },
  "message" : "2011-Sep-11 23:02:15 (GMT): User Alex has brought Free Line to
              front (in front of other elements)",
  "parameters" : null,
  "timestamp" : 1315774935536
}
```

- Aktion „ToBack“.

```
{ "action" : "toBack",
  "element" : { "properties" : {
      "uuid" : "05d9ce3d-d8e0-41b7-9925-e8ebe814a1a1" },
      "type" : "Rectangle"
    },
  "timestamp" : 1315775113728,
  "user" : "Alex",
  "whiteboardId" : "8b10da44-30bd-4872-9961-8d821dcd4daf"
}

{ "action" : "toBack",
  "element" : { "properties" : {
      "uuid" : "05d9ce3d-d8e0-41b7-9925-e8ebe814a1a1" },
      "type" : "Rectangle"
    },
  "message" : "2011-Sep-11 23:05:13 (GMT): User Alex has brought Rectangle to
              back (behind other elements) at position (115,285)",
```

```
  "parameters" : null,
  "timestamp" : 1315775113728
}
```

- Aktion „Clear“.

```
{ "action" : "clear",
  "timestamp" : 1315775228358,
  "user" : "Alex",
  "whiteboardId" : "8b10da44-30bd-4872-9961-8d821dcd4daf"
}

{ "action" : "clear",
  "element" : null,
  "message" : "2011-Sep-11 23:07:08 (GMT): User Alex has cleared this
               Whiteboard",
  "parameters" : null,
  "timestamp" : 1315775228358
}
```

- Aktion „Resize“.

```
{ "action" : "resize",
  "parameters" : { "height" : "500", "width" : "400"},
  "timestamp" : 1315775323420,
  "user" : "Oleg",
  "whiteboardId" : "8b10da44-30bd-4872-9961-8d821dcd4daf"
}

{ "action" : "resize",
  "element" : null,
  "message" : "2011-Sep-11 23:08:43 (GMT): User Oleg has resized this
               Whiteboard to (400,500) px",
  "parameters" : { "height" : "500", "width" : "400" },
  "timestamp" : 1315775323420}
```