DIPLOMARBEIT

LOW-LATENCY MULTI-USER WEB APPLIKATIONEN BASIEREND AUF COMET UND DEM BAYEUX-PROTOKOLL

ERWEITERUNG FÜR AUTHENTIFIZIERUNG, AUTORISIERUNG, ZUVERLÄSSIGKEIT UND NACHRICHTENINTEGRITÄT

ausgeführt zum Zweck der Erlangung des akademischen Grades eines
„Diplom-Ingenieurs für technisch-wissenschaftliche Berufe“
am Masterstudiengang Telekommunikation und Medien
der Fachhochschule St. Pölten

ausgeführt von

Manuel Irrschik, BSc.

tm0710262025

unter der Erstbetreuung von

DI(FH) Fritz Grabo

Zweitbegutachtung von

Markus Seidl, Bakk.

St. Pölten, 13. April 2009

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Diese Arbeit stimmt mit der von den Begutachtern beurteilten Arbeit überein.

Ort, Datum	Unterschrift

Kurzfassung

Diese Arbeit setzt sich mit dem Thema der Rich-Internet-Applikationen im World-Wide-Web auf Basis von purem JavaScript, (X)HTML und CSS auseinander und beschäftigt sich im Detail mit den dabei auftretenden Problemen, betreffend der Latenz-Zeit für Daten-Updates durch den Server.

Da aktuelle Implementierungen basierend auf dem HTTP Protokoll nicht annähernd eine niedrig latente Kommunikation zwischen Client und Server ermöglichen, wie diese von Desktop Anwendungen gewohnt und auch für das Web wünschenswert wäre, wurde von der *Dojo-Foundation* die Transporttechnologie *CometD* und das darauf basierende *Bayeux-Protokoll* entworfen. Obwohl bereits eine Version 1.0 der Spezifikation vorliegt, gibt es doch einige Punkte, für welche das Protokoll im Moment noch keine Lösungsvorschläge parat hat. Unter anderem betrifft dies die für ein Netzwerk Protokoll wichtigen Punkte der Authentifizierung und Autorisierung, sowie Zuverlässigkeit und Integrität.

Nach einer kurzen Erklärung der Grundlagen der Rich Internet Applikationen und einer Einführung in die Funktionsweise von CometD und dem Bayeux-Protokoll wird erklärt, warum die Erweiterungen notwendig sind und wie diese in die aktuellen Implementierungen integriert werden können.

Abstract

This Diploma Thesis deals with the topic of Rich-Internet-Applications in the World-Wide-Web based on pure JavaScript, (X)HTML and CSS and furthermore with the occurring Problems concerning latency for data updates through the web-server.

Because of the fact, that current Implementations of the HTTP-Protocol can not reach a latency as low as we are used to it from desktop applications, the *Dojo-Foundation* invented the so called *Bayeux-Protocol* which is based on a transport technology called *CometD*. Although there is already a version 1.0 of the protocol specification, there are some points which are still not being cared about, but which are important for a networking protocol to be implemented. These include authentication and authorization as well as data integrity and reliability.

After a short explanation of Rich Internet Applications a short introduction to CometD and the Bayeux-Protocol this thesis will show up, why the extensions are necessary to be made and how they can be integrated in current implementations of the protocol.

Danksagung

Ich möchte mich zu aller erst bei meinem Betreuer, DI(FH) Fritz Grabo für seine sehr engagierte und wirklich ausgezeichnete Betreuung bei der Erstellung dieser Diplomarbeit bedanken. Die fachliche Beratung und häufigen Gespräche haben mir wesentlich bei der Erstellung dieser Arbeit geholfen. Auch wäre es ohne ein paar aufbauende Worte zur richtigen Zeit wohl manchmal nicht mehr so rasch weiter gegangen!

Weiters möchte ich mich bei meiner Familie für die Unterstützung während meiner Ausbildungszeit bedanken, ohne deren Hilfe ich die Chance auf diese Ausbildung nicht wahrnehmen hätte können und heute sicher nicht auf eine so erfolgreiche Zeit an der FH St. Pölten zurückblicken könnte.

Abschließend möchte ich auch meiner Lebensgefährtin Bettina Pröll einen großen Dank aussprechen, dass sie mich so großartig während meines Studiums unterstützt hat und auch für Ihre Geduld, die Sie vor allem in den letzten Monaten während der Erstellung dieser Arbeit haben musste!

St. Pölten, Österreich — Manuel Irrschik, BSc.

13. April 2009

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problembenennung, Motivation

Diese Arbeit ist im Rahmen meines Studiums an der FH St. Pölten, Master-Studiengang Telekommunikation und Medien entstanden. Sowohl aufgrund des gewählten Schwerpunktes "Web-Technologien" als auch aufgrund meiner persönlichen Affinität zum Thema der Web Entwicklung ist dies nun bereits die 2. Arbeit dieses Themenbereichs und setzt die getätigten Forschungen und Entwicklungen meiner letzten Arbeit fort.

Das Internet erfreut sich weltweit immer größerer Beliebtheit. In Europa ist bereits jeder Zweite online, in den USA besitzen sogar rund 75% der Bevölkerung einen Online-Zugang (siehe Stats [2008]). Mit der auch noch weiter steigenden Nutzung des Internets eröffnen sich für alle Firmen weltweit neue Märkte. Immer leistungsfähiger werdende Internetleitungen, Computer und Web-Browser etablierten schließlich das Web auch als Plattform für Applikationen. Mit dem Begriff Web-2.0 waren auch die so genannten "Rich Internet Applications" (RIA) geboren. Dies sind Anwendungen, welche über das World Wide Web und somit über den Web-Browser zur Verfügung gestellt und ohne Installation direkt ausgeführt werden können. Dabei lassen diese Applikationen die Grenze zwischen Web-Site und Desktop-Applikation nahezu verschwinden, da mit neuen Techniken und Frameworks auch die von Desktop-Anwendungen bekannte Usability erreicht werden kann.

Die Kerntechnologie der RIA benannte Jesse James Garret mit dem Akronym AJAX (**A**synchronous **J**avaScript **A**nd **X**ML; mehr dazu siehe Kapitel 2.2). Jedoch schafft man es auch mit AJAX nur schwer, eine für mehrere Benutzer synchrone Web-Anwendung zu entwickeln. Es scheitert im Web daran, dass ein Web-Server keine Verbindung zu einem Client aufbauen und diesem (aktualisierte) Daten zukommen lassen kann. Das Web ist ein reines *Anfrage - Antwort* Szenario, immer vom Client (Web-Browser) initiiert.

Um diese Problem zu umgehen, existieren nun bereits mehrere Ansätze, welche jedoch alle in Ihrer Entwicklung noch in sehr frühen Phasen stecken. Basierend auf einer dieser Entwicklungen, dem Bayuex-Protokoll der DOJO-Foundation, wird sich diese Arbeit dem Thema der *Low-Latency Multi-User Web Applikationen* widmen. In weiterer Folge werden einige Schwächen des aktuellen Entwicklungsstandes des Bayeux-Protkolles aufgezeigt und Lösungsansätze generiert, welche diese Schwächen kompensieren können.

1.2 Ziele, Vorgehensweise

Ziel der Arbeit ist es, das *Bayeux-Protokoll* der DOJO-Foundation zu analysieren, Schwächen aufzuzeigen und diese bestmöglich zu beseitigen.

Kapitel 2 stellt den aktuellen Ist-Zustand im World-Wide-Web und die Kerntechnologie AJAX mit ihren Vorteilen und Innovationen dar, zeigt aber auch deren Schwächen auf und geht auf vorhandene Lösungsansätze näher ein.

Kapitel 3 beleuchtet das Bayeux-Protokoll näher, geht auf den Aufbau und die enthaltenen Schwächen ein. Weiters werden aktuelle Implementierungen kurz vorgestellt, sowie die dahinterliegende Architektur und technischen Vorraussetzungen für die Forschung beleuchtet.

Kapitel 4 befasst sich mit dem Themen Authentifizierung und Autorisierung, welche im Bayuex-Protokoll nur am Rande erwähnt, aber nicht näher spezifiziert werden. Da für Business-Anwendungen Sicherheit und somit Authentifizierung unumgänglich ist, ist eine Erweiterung in diese Richtung ein Muss.

Kapitel 5 greift schließlich auch noch das Thema Zuverlässigkeit auf. Vor allem in kritischen Mehrbenutzer-Applikationen müssen Daten immer konsistent sein. Deshalb muss das Protokoll auch gewährleisten, dass Nachrichten, welche ausgesendet werden, auch tatsächlich zugestellt werden. Weiters sollte das Protokoll in der Lage sein, Übertragungsfehler oder Manipulationen der Nachrichten zu erkennen und zu beheben.

Kapitel 6 wird die Funktionalität der in den Kapiteln 4 und 5 erforschten Erweiterung anhand eines praktischen Beispiels unter Beweis stellen.

Kapitel 7 bietet eine Conclusio und Ausblicke für zukünftige Arbeiten.

1.3 Hinweis zur Berücksichtigung von Gender-Mainstreaming Prinzipien

In dieser Arbeit wird zugunsten einer besseren Lesbarkeit auf die weibliche Bezeichnung verzichtet bzw. eine geschlechtsneutrale Bezeichnung gewählt. In Zitaten wurde der ursprüngliche Wortlaut beibehalten.

Kapitel 2

Grundlagen für moderne Web-Applikationen

2.1 Anwendungen im World Wide Web

Während anfangs das rein auf Seiten basierende Web nur der Informationsbeschaffung diente (vgl. Ballard [2008], S. 9ff), sind im Laufe der Zeit immer mehr Aspekte (Kommunikation in Foren, Online-Shopping, uvm.) hinzugekommen. Der Hype um das World Wide Web wird immer größer, viele Unternehmen Investieren in Internet Technologien, bis es schließlich im Herbst 2001 zum Platzen der so genannten "dot-com-Blase" kommt. Dieses Ereignis versetzte das ganze Web in Aufruhr, bereitete aber auch den Weg für das "Web 2.0", welches Tim O'Reilly[1] seinem Online Blog (O'Reilly [2005]) wie folgt definiert:

"Web 2.0 is the network as platform, spanning all connected devices; Web 2.0 applications are those that make the most of the intrinsic advantages of that platform: delivering software as a continually-updated service that gets better the more people use it, consuming and remixing data from multiple sources, including individual users, while providing their own data and services in a form that allows remixing by others, creating network effects

[1] Tim O'Reilly ist Gründer und CEO der O'Reilly Media Inc., einem der größten EDV-Buch Verlage weltweit. Er ist sehr engagiert im Open Source Bereich und hat mit diversen Veranstaltungen zur Vereinheitlichung der Gedanken rund um Web 2.0 beigetragen

through an "architecture of participation", and going beyond the page metaphor of Web 1.0 to deliver rich user experiences."

Mit dem Begriff des Web 2.0 beginnt ein Umschwung und Umdenken im World Wide Web. Die Interaktion des Benutzers mit der Applikation, aber auch mit anderen Benutzern und somit auch das Gefühl der Zugehörigkeit wird bei den neuen Web-Portalen in den Vordergrund gestellt. Vor allem die Gedanken der steigenden Interaktion und der "Rich User Experience" führen dazu, dass sich Web-Entwickler neue Konzepte zur Kommunikation zwischen Server und Client überlegen müssen, um einerseits Bandbreite zu sparen, aber auch andererseits dem Benutzer mehr Komfort und Geschwindigkeit in ihren Applikationen bieten zu können. (vgl. O'Reilly [2007], S.1ff)

Gerhard Janisch hat in seiner Diplomarbeit (vgl. Janisch [2008]) einen guten Überblick über die verschiedenen Möglichkeiten geboten, wie im Moment Web-Anwendungen mit einem so genannten "Rich User Interface" umgesetzt werden können. Dabei unterteilt er in folgende 3 Kategorien:

1. Applikationen, die über einen Browser gestartet werden, aber unabhängig davon ablaufen (Beispiel: Java Web Start[2])
2. Applikationen, die in einem Browser-Plugin ablaufen (Beispiele: Adobe Flash[3]/Flex[4], Microsoft Silverlight[5], Java Applets[6])
3. Applikationen, die direkt im Browser ablaufen, so genannte AJAX Applikationen

Da externe Applikationen und Plugin-basierte Applikationen (Punkte 1 und 2) ohnehin eine Socket-Kommunikation[7] erlauben, ist hier das Problem der Nachrichten Zustellung vom *Server zum Client* nicht gegeben. Ganz im Gegensatz zu den AJAX Applikationen (Punkt 3), deren Kommunikation auf normalen HTTP-Requests (mittels dem

[2] siehe: http://java.sun.com/javase/technologies/desktop/javawebstart/index.jsp
[3] siehe: http://www.adobe.com/products/flash/
[4] siehe: http://www.adobe.com/products/flex/
[5] siehe: http://silverlight.net/
[6] siehe: http://java.sun.com/applets/
[7] Ein Socket ist ein Nachrichtenkanal zwischen Client und Server, mit dem diese, sobald der Socket geöffnet wurde, jederzeit in beide Richtungen miteinander kommunizieren können

XMLHttpRequest-Objekt) beruht. Wie bereits erwähnt, kann hier der Server nur auf Anfragen des Clients antworten, nicht aber ungefragt eine Nachricht direkt an den Client schicken.

2.2 AJAX - Grundlegende Interaktivität

AJAX ist die erste Technologie, die es auch den Web-Entwicklern ermöglicht, Desktop-ähnliche Applikationen zu schaffen, welche sowohl eine "Rich User Experience" bieten, als auch Daten ohne große Verzögerungen und im Hintergrund nachladen können. Dabei ist AJAX keine eigene Technologie für sich, sondern eine Definition für die gemeinsame Verwendung mehrerer Technologien (vgl. Garrett [2008]):

- Standardisierte Darstellung basierend auf *XHTML* und *CSS*
- Dynamische Darstellung und Interaktion unter Verwendung des *Document Object Models*
- Daten-Austausch und -Manipulation mittels *XML* und *XSLT*
- Asynchrones Laden von Daten basierend auf dem *XMLHttpRequest-Objekt*
- *JavaScript*: Die Technologie, die alles verbindet

Auch wenn dies eine weite verbreitete Definition von AJAX ist, so werden zumeist leicht von dieser Definition abgewandelte Technologien verwendet. XML ist zwar bereits im Namen von AJAX enthalten, jedoch verwenden viele AJAX-Libraries entweder eine komplett unabhängige Schnittstelle, die es dem Benutzer erlaubt aus verschiedenen Container-Formaten für Ihre Nachrichten zu wählen oder aber sie setzen auf die einfache und nahe liegende JavaScript Object Notation (JSON[8]) (vgl. Powell [2008], Kapitel 5).

[8] JavaScript Object Notation, eine Daten-Beschreibungs Sprache basierend auf ECMA-Script, Details siehe http://www.json.org/

2.2.1 Anwendungsgebiete

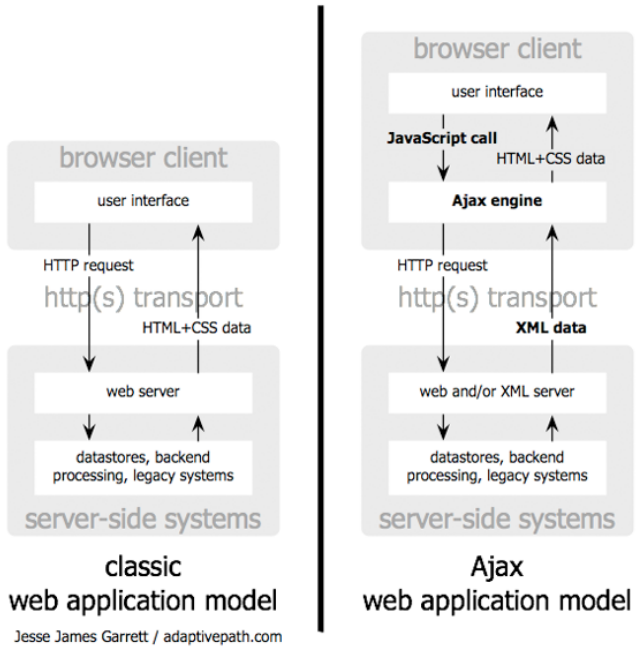
Wie Abbildung 2.1(a) zeigt, so führen AJAX Applikationen eine zusätzliche Programm-Schicht auf dem Client ein (in JavaScript), welche das Kernstück der AJAX Anwendung darstellt und die Asynchrone Kommunikation mit dem Server übernimmt. Diese Schicht beinhaltet auch einen signifikanten Teil an Logik, da auch hier Daten validiert, dargestellt, zum Server gesendet und vom Server empfangen werden können (vgl. Garrett [2008]).

Diese zusätzliche Schicht erlaubt es Web-Entwicklern, dem Benutzer das Gefühl zu geben, er arbeite mit einer normalen PC Anwendung (vgl. Garrett [2008]):

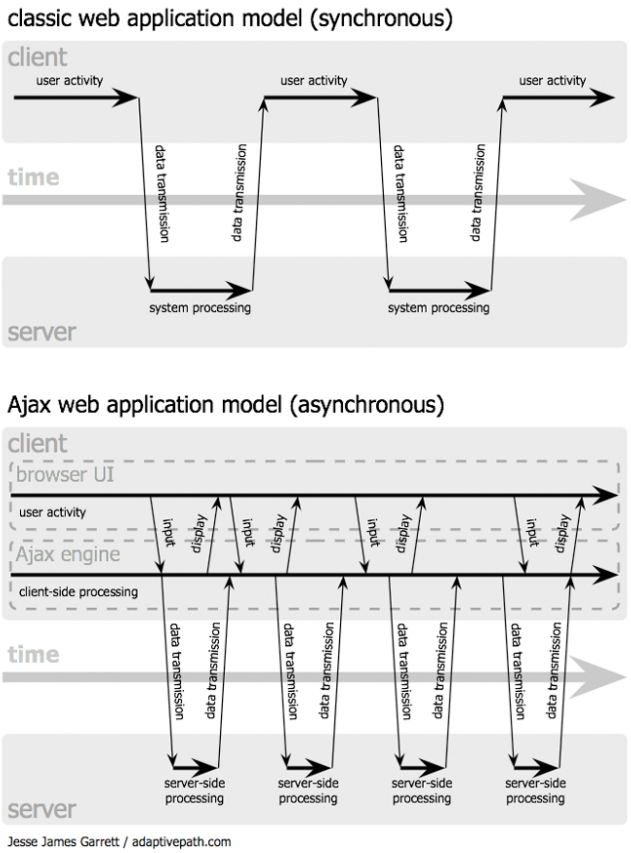
- Bei Interaktionen mit der Applikation wird nicht die gesamte Seite neu geladen und aufgebaut, sondern nur der relevante Teil aktualisiert
- Es kann direkt auf Benutzereingaben reagiert und dem Benutzer Hilfe angeboten werden (Client Side Form Validation)
- Es können völlig neue Steuer-Elemente kreiert werden, wie zum Beispiel Combo-Boxen/Eingabefelder mit Auto-Vervollständigung

Betrachtet man die neue Schicht auch von der zeitlichen Sicht her, so wird schnell klar, wo die asynchrone Kommunikation ihre Vorteile ausspielt (siehe Abbildung 2.1(b)). Die Benutzer Interaktion wird nicht durch lästiges Laden im Vordergrund gestört, dies passiert rein im Hintergrund, wodurch sich ein Eindruck einer wie von Desktop-Systemen bekannten Anwendung ergibt (vgl. Powell [2008], S3ff).

Abbildung 2.2(a) zeigt ein Vielen bekanntes Microsoft Office Excel 2007 Fenster. Im Vergleich hierzu zeigt Abbildung 2.2(b) eine sehr ähnliche Anwendung namens *Google Spreadsheet*. Diese ist aber keine Desktop Anwendung wie etwa Microsoft Excel, sondern eine Rich Internet Applikation. Auch wenn diese in Punkto Features nicht mit Excel oder dem Open Source Pendant *OpenOffice.org Calc* mithalten kann, so kann Google Spreadsheet trotzdem ein ähnliches Interface bieten, die Datenformate der beiden Desktop Produkte einlesen und auch teilweise bearbeiten. Und all dies ohne lokale Installation, von jedem Computer (mit Internetzugang), direkt im Web-Browser.

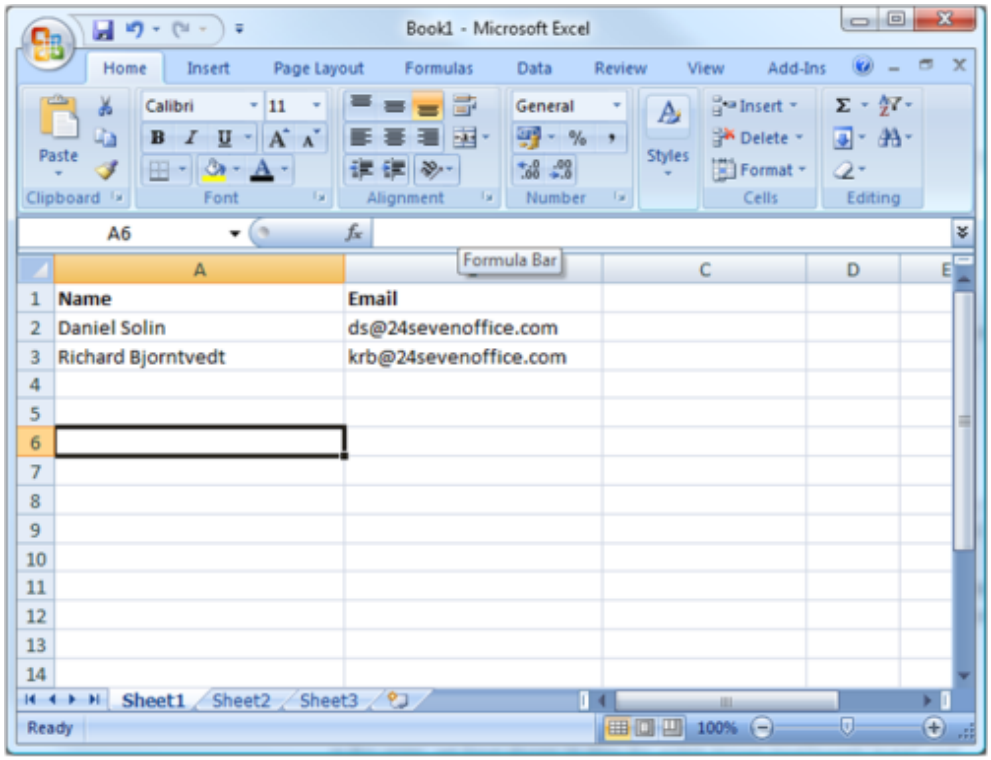


(a) Anwendungsschichten bei der Kommunikation

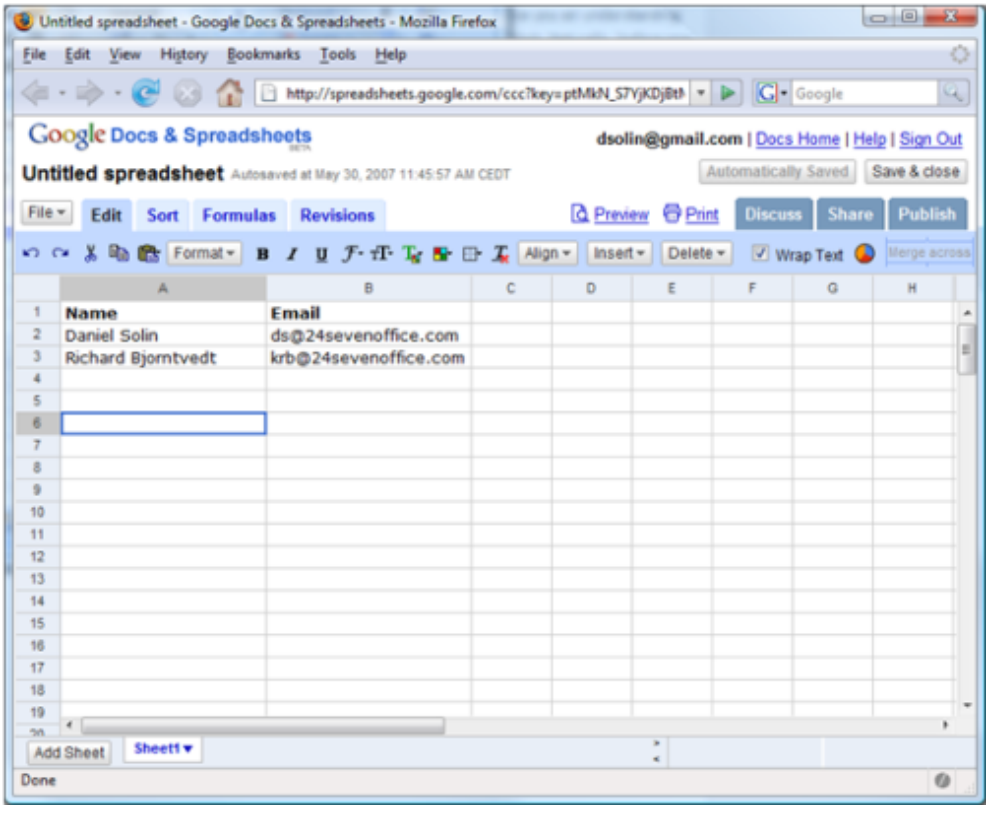


(b) Zeitlicher Ablauf der Kommunikation

Abbildung 2.1: Vergleich von klassischen Web Anwendungen mit AJAX Web Anwendungen (Quelle: Garrett [2008])



(a) Microsoft Office Excel 2007



(b) Google Spreadsheet

Abbildung 2.2: Vergleich von Desktop Anwendung mit einer Rich Internet Application

2.2.2 Limitierungen

Mit AJAX wurden zwar die Grundsteine für eindrucksvolle Web 2.0 Applikationen gelegt, jedoch nur für typische Einzelplatz-Anwendungen. Immer wenn mehrere Leute mit den selben Daten arbeiten, fehlt bei AJAX der Rück-Kanal vom Server, um Clients bei Änderungen schneller benachrichtigen zu können. Ein typisches Beispiel wäre eine Groupware: Mehrere Leute sehen die gleichen Notizen, Termine, Projekte. Folgendes Szenario zeigt aber die Problematik:

1. *Person X* und *Person Y* öffnen beide den Kalender ihrer Groupware, welche als Rich Internet Applikation entwickelt wurde und möchten eine Notiz zum morgigen Termin hinzufügen.
2. Während *Person X* bereits den Termin öffnet und bearbeitet, erhält *Person Y* einen Telefon-Anruf.
3. Nach 2 Minuten ist das Telefonat beendet und *Person Y* öffnet ebenfalls den Termin und bearbeitet diesen.
4. In der Zwischenzeit ist jedoch *Person X* bereits fertig und hat den neuen Termin gespeichert.
5. *Person Y* hat davon nichts mitbekommen, ändert den Termin ebenfalls ab und speichert - und überschreibt somit die Änderungen von *Person X*.
6. Keiner bekommt etwas von dem eben passierten Fehler mit, da der Server *Person Y* nicht warnen kann, dass in der Zwischenzeit der Termin geändert wurde. Könnte der Server von Sich aus anderen, von der Änderung betroffenen, Benutzern bescheid geben, so würde es zu keinem Fehler kommen.

Um bei derartigen Szenarien nicht nur mit serverseitigen Locking-Mechanismen entgegenwirken zu könne, haben sich Web-Entwickler bereits mehrer Lösungsansätze einfallen lassen, um auch die Clients über sich ändernde Zustände zu informieren.

2.3 Lösungsansätze

2.3.1 Comet oder Reverse AJAX

Ebenso wie Jesse James Garret im Jahr 2005 den Begriff AJAX definierte, kam Alex Russel 2006 mit dem Begriff "Comet" auf. Comet, manchmal auch "Reverse AJAX" genannt, ist der Name für eine Technik, welche es einem Web-Server erlaubt, direkt und jederzeit Nachrichten an einen Client zu senden. Basierend auf Garrets Netzwerk-Diagramm für Ajax (Abbildung 2.1(b)) kann man eine Comet-Kommunikation wie in Abbildung 2.3 darstellen. Der Unterschied besteht darin, dass nicht der Client immer beim Web-Server nach Änderungen fragen muss, sondern der Web-Server diese von sich aus an den Client senden kann.

Basierend auf Comet können Mehrbenutzer-Applikationen viel einfacher erstellt werden, als dies mit normalem AJAX möglich wäre. Es gibt mehrere Möglichkeiten, Comet zu implementieren, eine davon ist Long-Polling, wie im folgendem Kapitel erklärt (vgl. Russel [2008b]) wird.

2.3.2 Short- und Long-Polling

Eine mögliche Lösung für das Daten-Aktualisierungs-Problem wäre, den Datenabgleich mit dem Server durch Polling[9] durchzuführen, jedoch ist hier der zu erzielende Erfolg abhängig vom gewählten Intervall:

- Bei *sehr geringem* Intervall (< 5 Sekunden) kann es, vor allem wenn viele Benutzer mit einer Applikation arbeiten, dazu führen, dass der Server unter der starken Last nicht mehr reagiert. Jedoch sind hierbei die Daten sehr aktuell.
- Bei *eher großem* Intervall (> 30 Sekunden) wird zwar die Belastung an den Server gesenkt, jedoch können hier auch leichter Probleme durch inkonsistente Daten auftreten

[9] Unter Polling versteht man ständige das Senden von Anfragen an einen Server in einem bestimmten, festgelegten Intervall

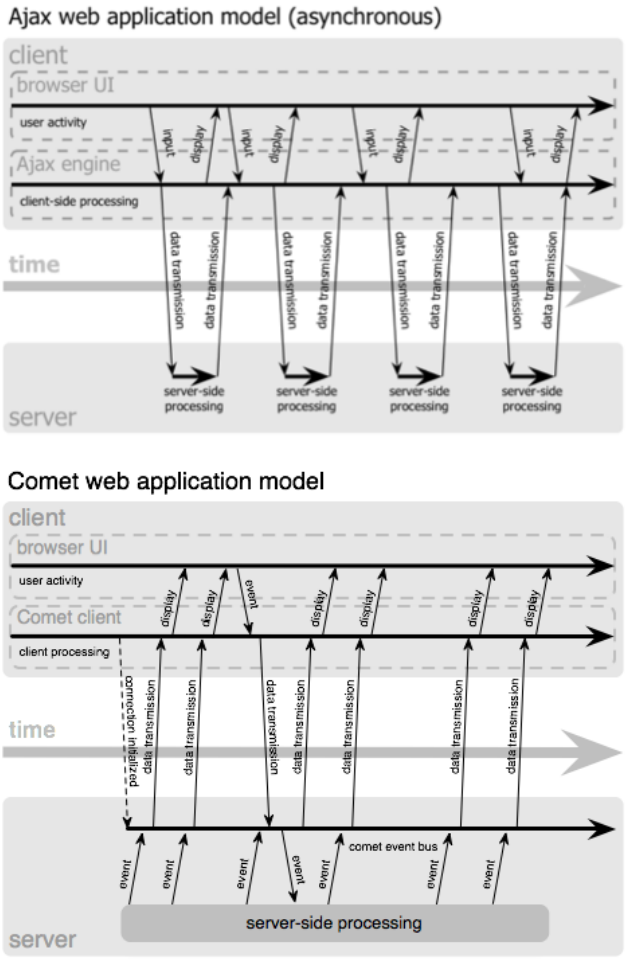


Abbildung 2.3: Vergleich von AJAX Requests und Comet Requests (Quelle: Russel [2008b])

Hier muss je nach Applikation das richtige Intervall gefunden, festgelegt und gegebenenfalls, je nach Benutzeraufkommen, nachjustiert werden. Unabhängig vom gewählten Intervall spricht man bei dieser Vorgehensweise vom so genannten *Short-Polling*.

Eine bessere Methode ist das so genannte *Long-Polling*. Hierbei wird eine HTTP Verbindung zum Web-Server aufgebaut und vom Server so lange offen gehalten, bis neue Daten für den Client vorhanden sind. Zwar wird hierdurch nicht durch ständiges (und oft auch unnötiges) Polling der Server penetriert, jedoch benötigt auch für dieses Verfahren der Server erhebliche Ressourcen, da pro offen gehaltener Verbindung zum Client ein Applikations-Thread am Server blockiert wird. Die Anzahl dieser Server-Threads ist in den meisten

Fällen aber limitiert, wodurch auch der Web-Server bei großer Benutzer Anzahl reaktionsunfähig werden kann (vgl. Wang [2008], S. 2).

2.3.3 Jetty Continuation und Suspendable Servlets

Die Entwickler des Jetty Web-Servers erkannten dieses Ressourcen Problem und haben in Ihren aktuellen Versionen 7 des Web-Servers (zur Zeit des Chreibens nur als Beta Version verfügbar) Möglichkeiten implementiert, um dieses Problem zu umgehen.

Bisher reichte der Ansatz, dass pro HTTP Verbindung zum Client ein eigener Server Thread angelegt wird. Aber langlebige HTTP Verbindungen können bei großem Benutzeraufkommen schnell zu einem Problem werden. Deswegen verfolgen die Entwickler hier den Ansatz, nicht pro *Verbindung* einen Thread anzulegen, sondern nur pro *Request*. Das heißt erst, wenn tatsächlich Daten für einen Client verarbeitet und gesendet werden, wird auch ein Applikations-Thread aus dem Thread-Pool der HTTP Verbindung zugeordnet und die Daten werden verarbeitet. Liegen in der Zwischenzeit (bei HTTP-Keep-Alive Connections) keine Requests an, so werden die Verbindungen in einen Select-Pool abgelegt, wo diese warten, bis neue Requests anliegen.

Mit der zunehmenden Verwendung von AJAX Polling (genau genommen Short-Polling) wurde aber auch das *Thread-Per-Request* Modell an seine Grenzen getrieben. Abhilfe schaffte die Kombination aus *AJAX Long Polling* und dem als *Continuation* getauften Feature von Jetty. Die bereits beschriebenen Long-Polling Verbindungen werden also, ähnlich dem Thread-per-Request Konzept, in einen Schlaf-Zustand versetzt, bis Daten für den Client anliegen. Der Unterschied hierbei ist, dass keine leere HTTP Verbindung schlafen gelegt wird, sondern ein bereits vom Client initiierter, aktiver HTTP-Request (siehe Wilkins [2008b]).

Dies ist eine der wenigen Implementierungen von Comet (siehe Kapitel 2.3.1), welche meist gemeinsam mit dem von der DOJO Foundation entwickelten Bayeux-Protokoll (siehe Kapitel 3) zum Einsatz kommt und somit zeitnahe bidirektionale Kommunikation zwischen Web-Server und -Client bietet.

Neben den Continuations in Jetty 6.0 wurde auch für die Servlet 3.0 API Spezifikation bereits ein JSR eingereicht (JSR-315: Asynchronous Servlet 3.0), welcher "suspendable" und "resumeable" Java-Servlets vorsieht (siehe Wilkins [2007]). Eine Implementierung dieser Spezifikation ist bereits in der ersten Beta-Version von Jetty 7.0 durchgeführt worden, wird aber in dieser Arbeit aufgrund des frühen Entwicklungsstadiums nicht näher behandelt.

Kapitel 3

Das Bayeux-Protokoll

Selbst wenn sich Comet noch so gut realisieren ließe, bliebe noch immer ein Problem, dass vom Design des HTTP-Protokolls stammt (siehe Fielding [1999], Kapitel 8.1.4 - Practical Considerations):

"*Clients that use persistent connections SHOULD limit the number of simultaneous connections that they maintain to a given server. A single-user client SHOULD NOT maintain more than 2 connections with any server or proxy. A proxy SHOULD use up to 2*N connections to another server or proxy, where N is the number of simultaneously active users. These guidelines are intended to improve HTTP response times and avoid congestion.*"

Im RFC 2616, welcher das HTTP Protokoll definiert, wird also darauf hingewiesen, dass zur Ressourcenschonung der Web-Server und zur Erhöhung der allgemeinen HTTP Antwortzeiten ein Client nur maximal 2 gleichzeitige Verbindungen zu einem Web-Server öffnen sollte.

Greg Wilkins[1] hat in einem Online Bericht (vgl. Wilkins [2008a]) erklärt, warum diese Limitierung der Client-Verbindungen zu einem Problem werden kann:

Auch wenn es sich nur um eine Richtlinie handelt, so wird diese doch heute in den meisten HTTP-Implementierungen umgesetzt, so auch in den aktuellen Web-Browsern. Da

[1] Greg Wilkins ist CTO des Java Spezialisten Webtide, welcher unter anderem hinter der Entwicklung des Web-Servers Jetty steckt. Außerdem hat Wilkins an der Erstellung des Bayeux-Protokolls mitgewirkt.

eine Verbindung für allgemeine Kommunikation mit dem Server frei bleiben sollte, bleibt auch nur eine Verbindung für einen Comet-Request übrig. Viele Szenarien würden jedoch mehrere Comet-Requests erfordern, wie zum Beispiel:

- 2 Frameworks innerhalb einer Seite, die beide unabhängig von einander einen Comet-Kanal aufbauen möchten
- Mehrer Frames auf einer Seite, die Ihre eigene Comet-Instanz aufbauen möchten
- Mehrer Tabs/Fenster, welche auf die selbe Web-Applikation zeigen

Vor allem, wenn innerhalb eines Browser-Fensters mehrere Applikationen eine Comet-Verbindung zum selben Server aufbauen, werden sich diese gegenseitig blockieren. Dies trifft nicht nur auf Comet-Verbindungen zu; auch bei Seiten, welche viele AJAX Requests senden, müssen diese Requests durch eine Client-Seitige Logik in einer Warteschlange verwaltet und nacheinander abgearbeitet werden (in der Praxis übernehmen dies die verwendeten Frameworks). Auch wenn somit die Kommunikation geordnet abläuft, so entsteht doch eine zeitliche Verzögerung, welche sich auf die Antwort-Zeiten am Client und somit auch auf die Usability der Web-Applikation auswirkt.

3.1 Aufbau, Zielsetzung

Wilkins führt in der Publikation (Wilkins [2008a]) fort, dass es sich beim Bayeux Protokoll an sich um kein Framework handelt, das die Verbindungen zum Webserver managen soll, sondern um eine Spezifikation, die es ermöglicht, die zuvor genannten Limitierungen zu umgehen. Die folgenden Haupt-Kriterien beim Entwurf des Protokolls zeigen bereits, worauf es ankommt:

1. Das Protokoll spezifiziert eine publish/subscribe Kommunikation, welche im Vergleich zu normalen AJAX XHR-Requests flexibler ist, besser skaliert und auch eine gemeinsame Nutzung der Ressourcen erlaubt.

2. Bayuex spezifiziert, dass der Übertragungs-Modus zwischen Client und Server verhandelbar ist und auch nicht notwendigerweise über HTTP passieren muss.

3. Obwohl der Inhalt der Bayeux-Nachrichten in JSON definiert ist, so muss dies nicht zwingend heißen, dass alle Implementierungen die Nachrichten auch per JSON senden.

Um das Problem mit den nur 2 verfügbaren Verbindungen zu lösen, werden im Bayeux-Protokoll alle Ereignisse der zu Grunde liegenden publish/subscribe Architektur in einem HTTP Request zusammengefasst (detaillierte Funktionsweise siehe Kapitel 3.3).

Eine grundlegende Frage ist die nach der dahinterstehenden Struktur, auf welche der publish/subscribe Mechanismus angewendet wird. Die Clients melden sich an einem so genannten *Channel* an. Diesen kann man als eine Art Chat-Raum verstehen. Jeder Client, der sich in einem Raum anmeldet, kann dort Nachrichten hinterlassen. Diese Nachrichten werden vom Server an alle anderen im Channel befindlichen Clients verteilt. Ein Client kann natürlich auch zu mehreren Channels subscriben und somit auch mehrere Empfangskanäle offen halten. All diese Kanäle werden laut der Bayeux-Spezifikation in einem HTTP Request gebündelt (mehr dazu in den technischen Details, Kapitel 3.3).

Die Nachrichtenkanäle sind von der Namensgebung her ähnlich einem Datei-Pfad. Wäre die Applikation ein Chat, so könnte man Räume in diesem Chat wie folgt abbilden: `/chat/raum1` bzw. `/chat/raum2`. Für manche Anwendungen kann es auch von Vorteil sein, alle Nachrichten eines bestimmten Pfades zu erhalten (z.B. ein Log-Dienst im Chat). Dies wird durch Wildcards in den Pfadnamen ermöglicht, wie zum Beispiel `/chat/*` (würde alle Nachrichten aus sämtlichen direkten Sub-Channels von `/chat` zugestellt bekommen; vgl. Crane [2008], S.87ff).

Durch diese Namenskonvention können beliebige Strukturen hierarchisch abgebildet werden. Ein kleines Beispiel einer Channel-Struktur für eine Groupware könnte wie folgt aussehen (nicht vollständig ausgeführt):

- /presence/users ... Zum nachverfolgen von Benutzerereignissen (Login/Logout)

- /projects/Projekt_A ... Benachrichtigungen zum Projekt A
- /projects/Projekt_X ... Benachrichtigungen zum Projekt X
- /calendar/2009/01/01 ... Nachrichten die den 01.01.2009 im Kalender betreffen
- /calendar/2009/01/* ... Nachrichten die den gesamten Januar 2009 betreffen
- /calendar/2009/** ... Nachrichten die das gesamte Jahr 2009 betreffen
- /private/user_xy/mail ... Channel der den User xy bei neuen Mails benachrichtigt

3.2 Schwächen

Obwohl das Protokoll bereits in einer Version 1.0 vorliegt, gibt es doch mehrere wichtige Dinge, welche bisher unbeachtet gelassen oder nicht näher spezifiziert wurden. Betrachtet man das vorangegangene Channel-Beispiel zur Groupware etwas näher, so fallen mehrere Dinge auf:

1. Wer verhindert, dass ein Benutzer, der nicht dem "Projekt A" angehört, nicht trotzdem dem Channel subscribed und somit Nachrichten bekommt, die er nicht bekommen dürfte?
2. Wie wird verhindert, dass man in den privaten Channel eines Benutzers subscribed?
3. Wenn ein Kalendereintrag bearbeitet wird, wäre es klug diesen in der Zwischenzeit für die anderen Personen zu sperren (auf deren GUI read-only zu setzen), anstatt ein serverseitiges Pessimistic/Optimistic Locking zu verwenden. Wer garantiert jedoch, dass diese Sperre bei allen nötigen Personen ankommt?

Auf die Punkte 1 und 2 wird im Entwurf des Bayuex-Protokolls (vgl. Russel [2008a]) nur bedingt eingegangen. Kapitel 7.1 weist *sehr kurz* auf die Tatsache hin, dass für die Channels eine Authentifizierung verwendet werden *kann*. Es wird jedoch weder ein Vorschlag zu zu verwendenden Technologien gemacht, noch wird beschrieben, wie die Authentifizierung bezüglich des Protokolls durchgeführt werden könnte.

Um diese Anforderungen erfüllen zu können, werden sich die Kapitel 4 und 5 dieser Arbeit mit den Möglichkeiten auseinander setzen, wie die fehlenden Punkte in das Protokoll integriert werden könnten. Bei erfolgreicher Forschung wird dies in einer Erweiterung des Protokolls resultieren.

Eine große Schwäche ist jedoch die derzeit mangelnde Dokumentation des Projektes an sich. Verlässt man sich rein auf die Protokoll-Referenz (Russel [2008a]), so sieht das Protokoll (in Version 1.0draft1) eher halb-fertig aus. Wenn man sich aber etwas näher mit den Referenzimplementierungen beschäftigt, erkennt man auch, dass vieles zwar bereits implementiert, aber in der Refernz oder Online-Dokumentation nicht erwähnt wurde (Beispiel: Security-Policy[2] in der cometd-jetty API oder die von Greg Wilkins kürzlich vorgestellte ACK-Extension[3])

Die Acknowledge-Extension von Greg Wilkins wurde erst kurz vor fertigstellung dieser Diplomarbeit veröffentlicht und kann daher bei den Implementierungen in dieser Arbeit nicht mehr berücksichtigt werden. Mit dieser Erweiterung reagiert Wilkins auf Feedback aus der Community, welche Zuverlässigkeits-Features in der Bayeux-Implementierung vermissten. Es wird serverseitig eine Nachrichten-Warteschlange eingerichtet, welche alle Nachrichten bis zum eintreffen eines ACKs zwischenspeichert. Dabei geht es vorrangig darum, nicht zugestellte Nachrichten nach einem Verbindungsabbruch erneut zuzustellen (vgl. Wilkins [2009]).

3.3 Technologie im Detail

Das Grundgerüst des Bayeux-Protokolls sind die Channels. Damit trotz vieler möglicher Channels nicht beliebig viele Verbindungen zum Server offen gehalten werden müssen, werden diese Channel-Verbindungen in einem Comet-Request gebündelt (siehe Abbildung 3.1). Für die Funktionalität dieses Konzeptes ist es notwendig, dass der Server einen Client eindeutig identifizieren kann, um somit auch eine Liste aufzubauen, welche Client-Channel Abbonnements beinhaltet.

[2] siehe http://svn.cometd.com/trunk/cometd-java/api/src/main/java/org/cometd/SecurityPolicy.java
[3] siehe http://blogs.webtide.com/gregw/entry/cometd_acknowledged_message_extension

All diese Initialisierungen und das Zuweisen der Client-ID passiert im initialen Handshake-Prozess zwischen Client und Server.

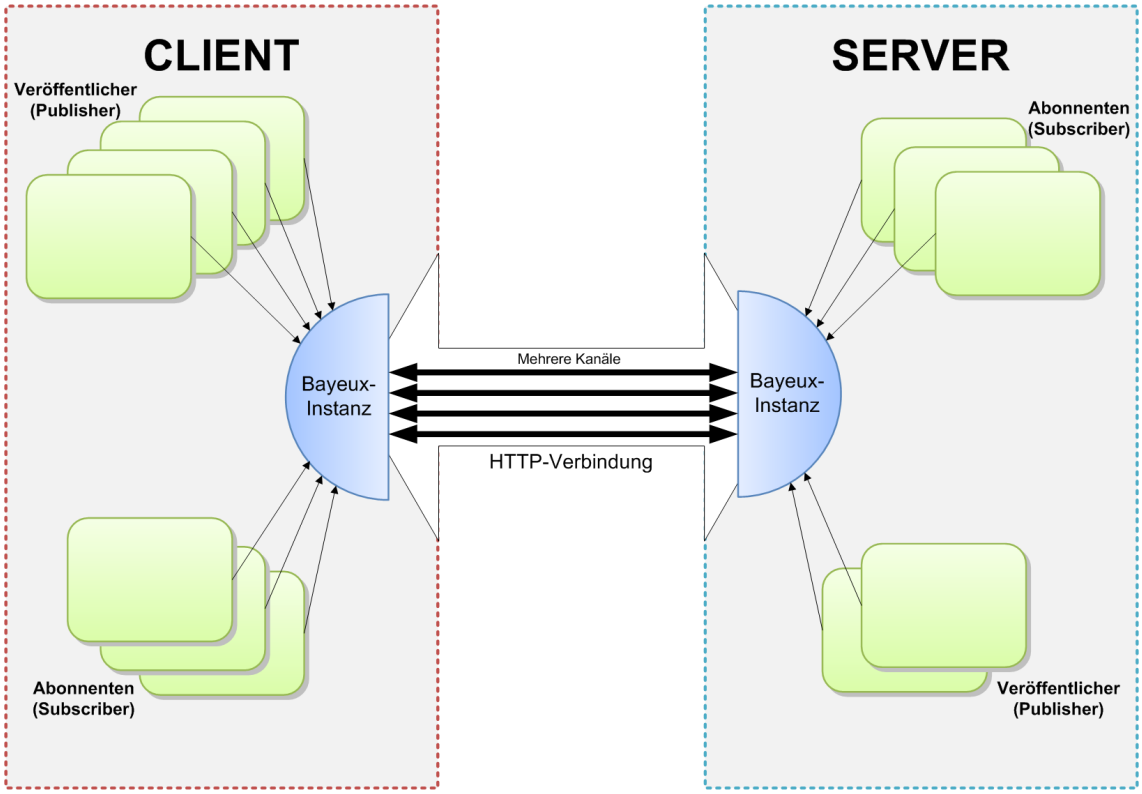


Abbildung 3.1: Channel-Kapselung in einem HTTP Request (vgl: Crane [2008])

3.3.1 Nachrichten und Nachrichtenformate

Nachrichten im Allgemeinen werden im Bayeux-Protokoll standardmäßig im JSON-Format zwischen Server und Client ausgetauscht und beinhalten typischerweise folgende Daten: (vgl. Crane [2008], S91f)

- *Inhalt* der Nachricht, kann ein einfacher Text (Zeichenkette), aber auch eine komplexe JSON Struktur sein
- Der *Channel*, an den die Nachricht gesendet wird
- Die *Unique-Client-ID*, die der Client vom Server zu Beginn zugeordnet bekommt

- Eine *Nachrichten-ID*, z.B ein Counter
- Ein `ext`-Feld für Protokoll-Erweiterungen

Wie eine Bayeux-Nachricht aussehen könnte, zeigt Listing 3.1.

```
[
    {
        "data": {
            'event' : 'sign-on',
            'user-id' : 'max'
        },
        "channel": "/application/presence",
        "clientID": "un1qu3Cl13nt1d",
        "id": "47",
        "ext": {
            "auth": { version: "0.1" }
        }
    }
]
```

Listing 3.1: Beispiel Bayeux-Nachricht im JSON Format

Das *ext*-Feld kann von jeder Protokoll-Erweiterung in jedem Nachrichten-Typ zum Übertragen von erweiterungsspezifischen Daten verwendet werden. Dabei sollte jede Erweiterung die Daten für sich in einem sub-Objekt der JSON Struktur des ext- Feldes anlegen (wie im Listing mit dem *auth*-Feld gezeigt).

3.3.2 Meta-Channels

Meta Channels sind jene Kanäle, über welche die System-Nachrichten zur Initialisierung der Bayeux-Kommunikation gesendet werden. Diese Kanäle haben ein paar Besonderheiten gegenüber normalen Channels: (vgl. Russel [2008a], Kapitel 2.2)

- Sie sind alle im Namensraum `/meta` untergebracht
- Clients *dürfen nicht* auf diese Channels subscriben
- Server *sollten nicht* auf diese Channels subscriben

- Nachrichten in Meta-Channels *dürfen nicht* zu anderen Clients verteilt werden
- Der Server-Handler des Meta Channels kann eine Antwort senden, wenn dann aber nur an den anfragenden Client
- Hat die Request-Nachricht ein ID Feld beinhaltet, so *muss* der Server diese ID auch im Antwort-Paket mitsenden

Die Standard-Implementierung des Bayeux-Protokolls sieht folgende Meta-Channels vor:

1. `/meta/connect`
   Über diesen Kanal stellt der Client eine Verbindung zum Bayeux-Server her
2. `/meta/disconnect`
   Über diesen Kanal kann der Client seine Verbindung zum Server beenden
3. `/meta/handshake`
   Über diesen Kanal werden Initialisierungsinformationen ausgetauscht
4. `/meta/ping`
   Kann als Keep-Alive Kanal verwendet werden, antwortet immer mit einer Kopie der gesendeten Nachricht, analog zu ICMP echo-request/echo-reply
5. `/meta/subscribe`
   Dieser Kanal wird zum Abonnieren von Channels verwendet
6. `/meta/unsubscribe`
   Mit diesem Kanal können Abonnements von Channels aufgehoben werden
7. `/meta/status`
   Dieser Kanal wurde zwar definiert, aber es wurde kein Verwendungszweck dazu vermerkt
8. `/meta/client`
   Dieser Kanal wurde zwar ebenfalls definiert, aber es wurde kein Verwendungszweck dazu vermerkt

3.3.3 Service-Channels

Im Entwurf des Protokolls (siehe Russel [2008a], Kapitel 2.2.3 und Kapitel 9) werden auch so genannte *Service Channels* erwähnt, welche typischerweise für die Interaktion mit genau einem Client gedacht sind (Service-Channels sind keine Broadcast-Channels, auf diese sollte auch nur ein Server subscriben).

Sie dienen dazu, um im Publish-Subscribe Konzept der Bayuex-CometD Architektur auch einfache Request-Response Abfragen zwischen einem Client und dem Server zu ermöglichen.

Service-Channels müssen im dafür vorgesehen Namensraum `/service` angelegt werden.

3.3.4 Transporttechnologien

Um die Nachrichten tatsächlich zwischen Client und Server austauschen zu können, muss eine geeignete Transporttechnologie eingesetzt werden. Die aktuelle Implementierung des Bayuex-Protokolls setzt dabei vorwiegend auf HTTP Also Transportweg bzw. CometD als Transporttechnologie. Da auch im Großteil des Protokolls außschließlich auf HTTP/Comet als Transporttechnologie eingegangen wird, wird sich auch die Forschung und Entwicklung in dieser Arbeit dem vorgegebenen Schema des HTTP Transports (da dieser im Moment die einzige brauchbare Implementierung darstellt) anschließen (vgl. Russel [2008a], Kapitel 1.4.2).

Jedoch hält das Bayeux-Protokoll hier alle Möglichkeiten offen, indem im aktuellen Entwurf jede Transporttechnologie, welche einen Request-Response Zyklus unterstützt, als für Bayuex geeignet genannt wird. Auch alternative Transports müssen sich an die semantischen Grundregeln des Bayeux-Protkolles halten, um Interoperabilität zu gewährleisten.

In aktuellen Referenz-Implementierungen wird der Transport an sich als Add-On Modul implementiert, um diesen somit vom Protokoll an sich unabhängig und somit auch leicht austauschbar zu gestalten.[4]

[4] siehe http://bugs.dojotoolkit.org/ticket/6997

3.3.5 Bayeux-Kommunikation - Ablauf

Der Ablauf einer typischen Bayeux-Kommunikation besteht im wesentlichen aus folgenden Schritten (wobei sich Schritte 4-6 beliebig oft während einer Session wiederholen ; siehe Russel [2008a], Kapitel 4 und 5):

1. Client und Server machen sich in einem Handshake *Transport-Parameter* aus
2. Während des Handshakes weist der Server dem Client seine *eindeutige ID* zu
3. Der Client führt ein *connect* durch
4. Der Client *subscribed* zu beliebigen Channels
5. Der Client kann ab jetzt Nachrichten *publishen* und in abonnierten Channels auch *empfangen*
6. Zum Stornieren eines Abonnements führt der Client ein *unsubscribe* auf den Channel durch
7. Schließlich führt der Client ein *disconnect* durch, um sich vom Server abzumelden

Im *Initialisierungshandshake* geht es vor allem darum, zwischen Client und Server den Verbindungstyp auszuhandeln und die eindeutige Client-ID (welche ab dem Zeitpunkt in jeder vom Client gesendeten Nachricht beinhaltet sein muss) zuzuweisen. Der genaue Inhalt wird am Besten anhand eines Beispieles erklärt: (vgl. Crane [2008], S.95f)

```
1  [
2      {
3          "version" : "1.0",
4          "minimumVersion" : "1.0",
5          "channel" : "/meta/handshake",
6          "id" : "0",
7          "ext" : { },
8          "supportedConnectionTypes" : [
9              "long-polling",
10             "callback-polling",
11         ]
12     }
13 ]
```

Listing 3.2: Bayeux-Handshake Nachricht vom Client

Kurz zur Erläuterung der gesendeten Felder (es wird nur näher auf jene Felder eingegangen, welche später im Rahmen dieser Arbeit auch von Belang sind; alle anderen Informationen können in der Protokoll-Referenz Russel [2008a] bzw. im Buch von Dave Crane Crane [2008] nachgelesen werden):

Die Initialisierungs-Nachricht wird an den Channel `/meta/handshake` gesendet, welcher für die Transport-Verhandlungen zuständig ist. Der wichtige Parameter hierfür ist `supportedConnectionTypes`. Dieser kann lt. aktueller Spezifikation folgende Werte annehmen: (vgl. Russel [2008a], Kapitel 3.4):

- `long-polling`: Die Nachrichten werden mittels des in Kapitel 2.3.2 beschriebenen Long-Polling Verfahrens zwischen Client und Server ausgetauscht. Dabei versendet der Client seine Nachrichten *application/x-www-form-urlencoded*[5] im Message-Teil eines POST Requestes. Die Antworten kommen direkt im POST Response-Body zurück.

- `callback-polling`: Die Nachrichten werden als Message-Parameter eines *urlencoded* GET Requests zum Server gesendet. Die Antworten werden in eine Java-Script Callback Funktion verpackt als Response zum Client gesendet[6].

- `iframe`: Die Nachrichten werden über den Body eines versteckten IFrame Elementes geladen und gesendet.

- `flash`: Die Nachrichten werden über ein Browser-Flash-Plugin im Binärformat übertragen.

Die Spezifikation ist vor allen bei den Angaben zu den Typen `iframe` und `flash` ziemlich ungenau, was den genauen Transport angeht. Vermutlich sind auch deshalb diese beiden Transportmöglichkeiten als optional markiert. `Long-Polling` ist das einzige Pflichtelement, welches jeder Client und Server unterstützen muss; `Callback-Polling` sollte ebenfalls in den Clients und Servern implementiert werden.

[5] Details zum URL Encoding siehe RFC 1738 Berners-Lee [1994]

[6] In Anlehnung an das JSON-P Pseudo-Protokoll, welches 2005 erstmals im MacPythonBlog Ippolito [2005] auftauchte. Es wird ein zusätzlicher Parameter `jsonp` angegeben, welcher den Namen der Callback-Funktion darstellt

Das `ext`-Feld kann, je nach verwendeten Erweiterungen verschiedene Inhalte haben, welche in der Dokumentation der jeweiligen Erweiterung nachzulesen sind. Da in diesem Fall keine Erweiterung geladen ist, ist das Feld auch leer (bzw. müsste nicht einmal vorhanden sein).

```
[
    {
        "id" : "0",
        "version" : "1.0",
        "minimumVersion" : "0.9",
        "channel" : "/meta/handshake",
        "successful" : true,
        "supportedConnectionTypes" : [
            "long-polling",
            "callback-polling",
        ],
        "advice" : {
            "reconnect" : "retry",
            "interval" : 0,
            "timeout" : 240000
        },
        "clientId" : "cl13nt1dg3n3r4t3d"
    }
]
```

Listing 3.3: Bayuex-Handshake Antwort vom Server

Der Server antwortet mit einer Nachricht wie in Listing 3.3. Auch der Server gibt hier die von ihm unterstützten Verbindungs-Typen im Feld `supportedConnectionTypes` an. Das Feld `successful` zeigt, dass der Server die Nachricht verstanden hat und erfolgreich verarbeiten konnte. Folgende 2 Felder sind vom Server neu dazugekommen:

1. `advice`: Das ist ein vom Server erstellter Vorschlag für den Verbindungsaufbau des Clients. In diesem Fall heißt es, dass der Client eine Verbindung für bis zu 4 Minuten (240.000 ms) öffnen soll und das Intervall zwischen dem Schließen einer und dem Öffnen der neuen Verbindung 0ms betragen soll (siehe Russel [2008a], Kapitel 3.6)

2. `clientId`: Der Server weist dem Client für die Dauer der Session eine eindeutige ID zu, die der Client ab sofort in jeder Nachricht mitzusenden hat, um wiedererkannt zu werden. Der Standard schlägt vor, zumindest eine Zeichenkette mit 128 Zufalls-Bits (entspricht 16 ASCII Zeichen) zu generieren (vgl. Russel [2008a], Kapitel 2.4).

Nachdem der Client die Entscheidung über den zu verwendenden Verbindungs-Typ getroffen hat, führt er ein `connect` durch:

```
[
    {
        "channel" : "/meta/connect",
        "clientId" : "cl13nt1dg3n3r4t3d",
        "connectionType" : "long-polling",
        "id" : "1"
    }
]
```

Listing 3.4: Bayeux-Connect Nachricht des Clients

Wenn der Server wiederum mit einem `'successful' : 'true'` antwortet, ist die Verbindung hergestellt und der Client kann beginnen, zu den Channels zu subscriben und Nachrichten zu publishen und zu empfangen.

Eine Subscription wird vom Client nicht wie vielleicht vermutet an den Channel direkt gesendet, sondern an einen eigenen Meta-Channel namens `/meta/subscribe`. Der Channel, welcher abonniert werden soll, steht als Wert des Attributes `subscription` in der Nachricht:

```
[
    {
        "channel" : "/meta/subscribe",
        "clientId" : "cl13nt1dg3n3r4t3d",
        "id" : "2",
        "ext" : {},
        "subscription" : "/calendar/2009/**"
    }
]
```

Listing 3.5: Bayeux-Subscribe Nachricht

Der Server antwortet mit einer fast gleichlautenden Nachricht, die um die Felder `successful` und `error` ergänzt wird. Das Feld `error` kann im Fehlerfall einen Fehlercode, einen Parameter und eine Fehlermeldung enthalten, zum Beispiel: `'error' : '403: /foo/bar:Subscription denied'` . Die Liste an Fehlermeldungen ist zum Zeitpunkt des

Schreibens noch nicht vollständig in den Protokoll-Entwurf aufgenommen worden. Es existiert jedoch um Versionierungssystem des Protokolls eine Datei, welche Fehlercodes auflistet, jedoch in einem völlig anderen Bereich als die direkt im Protokoll erwähnten[7]. Bei der finalen Version des Protokolls wird in diesem Bereich hoffentlich eine eindeutige Liste an definierten Fehlermeldungen und einem für Erweiterungen vorgesehenen Fehlernummernbereich definiert werden.

Hier kann man bereits eine weitere Ungereimtheit im Protokoll erkennen: obwohl zwar kaum auf den Punkt *Authentifizierung* eingegangen wird, wurde trotzdem eine passende Fehlermeldung definiert! (vgl. Russel [2008a], Kapitel 3.14 und bayeux_error_codes.json[7])

Der letzte wichtige Nachrichten-Type ist die *Publish-Message*, zum Senden von Nachrichten an einen Kanal (siehe Listing 3.6). Nach dem Standard beinhaltet eine publish-Nachricht den `channel` und das `data`-Feld. Die Parameter `clientId`, `id` und `ext` sind optional, wobei clientId und id nur in Spezialfällen weggelassen werden sollten (z.B. wenn der Server selbst eine Nachricht in einem Channel veröffentlicht).

```
1  [
2      {
3          "channel" : "/chat/room_a",
4          "clientId" : "cl13nt1dg3n3r4t3d",
5          "id" : "4",
6          "ext" : {
7              "reliab" : {
8                  "version" : "0.1"
9              },
10             "auth" : {
11                 "passphrase" : "p4ssphr4s3"
12             }
13         },
14         "data" : "Hallo! Das ist eine Testnachricht"
15     }
16 ]
```

Listing 3.6: Bayeux-Publish Nachricht

[7] siehe http://svn.cometd.org/trunk/bayeux/bayeux_error_codes.json

3.4 Architektur und Erweiterbarkeit

Das Bayeux-Protokoll basiert auf der Client-Server Architektur von Comet[8]:

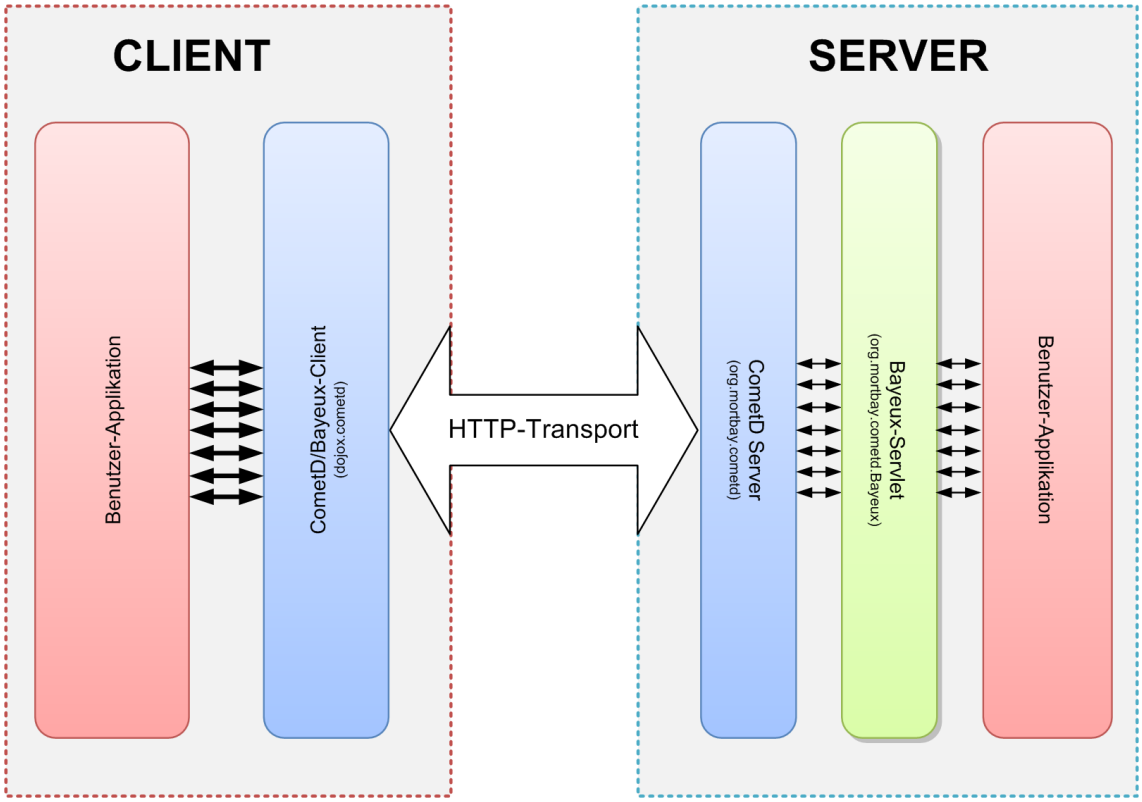


Abbildung 3.2: Architektur des Bayuex-Protokolls, aufbauend auf Comet

Auf beiden Seiten (Client & Server) läuft eine Implementierung des Bayuex-Protokolls, mittels welcher Benutzerapplikationen Bayeux-Services anbieten und auf diese zugreifen können. Noch eine Schicht tiefer, als Übertragungs-Kanal der Nachrichten, läuft eine CometD-Instanz. Um die Grundlagen des Protokolls (Meta-Handshakes, Connect, Disconnect) kümmert sich der Bayeux-Stack intern; erst wenn es um die Themen des *Abonnieren von Channels* (subscribe) oder *Veröffentlichen von Nachrichten* (publish) geht, kommt die Benutzerapplikation ins Spiel.

[8] In einem Eintrag des dojox-cometd Bug-Trackers (http://trac.dojotoolkit.org/ticket/6997) wird darauf hingewiesen, das entgegen der Definition(en) in der Spezifikation das verwendete Publish-Subscribe Prinzip nur das standardmäßige Nachrichten-Zustellungs-Verfahren darstellt, jedoch nicht das einzige. Weiters sind auch die zu verwendenden Transports als Add-On zu betrachten und nicht als fix vorgegeben.

Die weitere Beschreibung geht vertiefend auf die Implementierungen des cometd-java Servers (bzw. dessen jetty-Implementierung) und des dojoX-Clients ein. Die gezeigten und erklärten Features und Vorgehensweisen sollten jedoch auch in anderen Implementierungen analog anwendbar sein.

3.4.1 Anbieten eines Services

Um unter Jetty mittels CometD einen Bayeux-Service anbieten zu können, sind folgende Schritte nötig:

1. Erstellen eines Web-Applikations-Deskriptors (web.xml; siehe Listing 3.7)
   (a) Definieren des Comet-Servlets
   (b) Mapping für das Comet-Servlet definieren
   (c) Definieren eines Bayeux-Service Listeners
2. Erstellen des Bayeux-Service Listeners (Servlet)
3. Erstellen des Bayeux-Services (Servlet)

```
1  [ ... ]
2
3    <listener>
4      <listener-class>
5          BayeuxServicesListener
6      </listener-class>
7    </listener>
8
9    <servlet>
10     <servlet-name>cometd</servlet-name>
11     <servlet-class>org.mortbay.cometd.continuation.ContinuationCometdServlet</
           servlet-class>
12     <init-param>
13       <param-name>filters</param-name>
14       <param-value>/WEB-INF/filters.json</param-value>
15     </init-param>
16     <init-param>
17       <param-name>timeout</param-name>
18       <param-value>120000</param-value>
19     </init-param>
20     <init-param>
```

```
21        <param-name>interval</param-name>
22        <param-value>0</param-value>
23      </init-param>
24      <init-param>
25        <param-name>maxInterval</param-name>
26        <param-value>10000</param-value>
27      </init-param>
28      <init-param>
29        <param-name>multiFrameInterval</param-name>
30        <param-value>2000</param-value>
31      </init-param>
32      <init-param>
33        <param-name>logLevel</param-name>
34        <param-value>0</param-value>
35      </init-param>
36      <init-param>
37        <param-name>directDeliver</param-name>
38        <param-value>true</param-value>
39      </init-param>
40      <init-param>
41        <param-name>refsThreshold</param-name>
42        <param-value>10</param-value>
43      </init-param>
44      <load-on-startup>1</load-on-startup>
45    </servlet>
46
47    <servlet-mapping>
48      <servlet-name>cometd</servlet-name>
49      <url-pattern>/cometd/*</url-pattern>
50    </servlet-mapping>
51
52  [ ... ]
```

Listing 3.7: Beispiel einer web.xml für einen Bayeux-Service

Die *web.xml* sollte zumindest folgenden Elemente beinhalten (vgl. Listing 3.7):

Das Element **listener** definiert jenes Servlet, welches beim Instantiieren eines Bayeux-Servlets benachrichtigt wird (genauer gesagt wird es benachrichtigt, sobald dem Servlet-Context ein Attribut vom Typ *org.cometd.bayeux* hinzugefügt wird). In diesem Listener wird die erstellte Bayeux-Instanz abgerufen und die selbst erstellten Bayeux-Services registriert.

Das **cometd-servlet** definiert das Jetty-Comet-Continuation-Servlet mit diversen Parametern. Eine genaue Beschreibung aller Parameter findet sich in der Jetty-API[9].

Das **servlet-mapping** dient schließlich dazu, dem Comet-Servlet auch eine URL zuzuweisen, unter der später der Comet-Kanal angesprochen werden kann.

Damit beim Deployen der Web-Applikation auch der Bayeux-Service zur Verfügung gestellt wird, muss das **Bayeux-Service-Listener-Servlet** erstellt werden, wie in Listing 3.8 gezeigt. Dieser Listener würde auf das Hinzufügen einer Bayeux-Instanz zum Servlet-Context warten, und anschließend den in Listing 3.9 gezeigten Monitor-Service instantiieren, welcher keine andere Aufgabe hat, außer sämtliche Ereignisse der Meta-Channels im Jetty-Log mitzuprotokollieren.

```
1  public class BayeuxServicesListener implements ServletContextAttributeListener
2  {
3      public void initialize(Bayeux bayeux)
4      {
5            // Instantiieren der Services, Uebergeben der Bayeux-Instanz
6          new Monitor(bayeux);
7          bayeux.addExtension(new TimesyncExtension());
8          bayeux.addExtension(new AcknowledgedMessagesExtension());
9      }
10
11      // Wird vom Servlet-Container aufgerufen
12     public void attributeAdded(ServletContextAttributeEvent scab)
13     {
14           // Ueberpruefen, ob es sich bei dem uebergebenen Attribut
15           // um eine Bayeux-Instanz handelt
16         if (scab.getName().equals(Bayeux.DOJOX_COMETD_BAYEUX))
17         {
18                 // Bayeux-Instanz herauslesen\ldots
19             Bayeux bayeux=(Bayeux)scab.getValue();
20                 // \ldots und die Services instantiieren
21             initialize(bayeux);
22         }
23     }
24
25     public void attributeRemoved(ServletContextAttributeEvent scab) { ... }
26     public void attributeReplaced(ServletContextAttributeEvent scab) { ... }
27 }
```

Listing 3.8: Bayeux Service Listener Beispiel

[9] AbstractCometdServlet-API: http://www.mortbay.org/jetty/jetty-6/apidocs/index.html?org/mortbay/cometd/AbstractCometdServlet.html

```
1  public static class Monitor extends BayeuxService
2  {
3      public Monitor(Bayeux bayeux)
4      {
5          super(bayeux,"monitor");
6          subscribe("/meta/subscribe","monitorSubscribe");
7          subscribe("/meta/unsubscribe","monitorUnsubscribe");
8          subscribe("/meta/*","monitorMeta");
9      }
10
11     public void monitorSubscribe(Client client, Message message)
12     {
13         Log.info("Subscribe␣from␣"+client+"␣for␣"+message.get(Bayeux.
               SUBSCRIPTION_FIELD));
14     }
15
16     public void monitorUnsubscribe(Client client, Message message)
17     {
18         Log.info("Unsubscribe␣from␣"+client+"␣for␣"+message.get(Bayeux.
               SUBSCRIPTION_FIELD));
19     }
20
21     public void monitorMeta(Client client, Message message)
22     {
23         if (Log.isDebugEnabled())
24             Log.debug(message.toString());
25     }
26 }
```

Listing 3.9: Einfacher Bayeux-Monitor Service

Wie auch am Client, so kann auch der Server per `subscribe` einfach einen Channel abonnieren. Der zweite zu übergebende Parameter (siehe Listing 3.9, Zeilen 6-8) ist der Name der auszuführenden Methode. Diese muss eine der folgenden Signaturen aufweisen:

- `myMethod(Client fromClient,Object data)`

  fromClient beinhaltet eine Referenz zur Bayeux-Client-Instanz des Absenders der Nachricht, *data* kann entweder vom Typ `Message`, `Object` oder `Map<String,Object>` sein. Je nach Typ wird entweder die gesamte Nachricht oder nur der Daten-Teil (z.b. als `Map`) übergeben.

- `myMethod(Client fromClient,Object data,String id)`

  Als zusätzlicher Parameter kann auch die *id* der Nachricht empfangen werden.

- `myMethod(Client fromClient,String channel,Object data,String id)`
  Ebenfalls Optional, aber häufig gebraucht kann die Funktion den Channel-Namen direkt als Zeichenkette erwarten (v.a. bei Abonnements mit Wildcards interessant).

3.4.2 Grundlagen des Bayeux-Clients

Aus der Sicht der Anwendung ist der dojoX-CometD/Bayeux-Client sehr einfach gehalten (vgl. Listing 3.10):

- `dojo.require('dojox.cometd')` lädt die grundlegende Funktionsbibliothek.
- `dojox.cometd.init('/sample_app/cometd')` initialisiert den Comet-Kanal und das Bayeux-Protokoll (führt den Initialisierungshandshake und das anschließende Connect durch). Ab nun steht die CometD/Bayeux-Verbindung und es können `subscribe` und `publish`- Befehle abgesetzt werden.
- `dojox.cometd.subscribe('/sample/channel', object, 'func')` abonniert einen Channel, bei ankommenden Nachrichten wird die Funktion `func` des Objektes `object` aufgerufen. Parameter ist hierbei das gesamte Message-Objekt.
- `dojox.cometd.publish('/sample/channel', ...)` veröffentlicht die übergebene Nachricht im entsprechenden Kanal.

Die in Listing 3.10 verwendete Methodik `dojox.cometd.startBatch()` und `dojox.cometd.stopBatch()` dient zur Zusammenfassung und zum gemeinsamen Absetzen mehrerer Requests in einem Vorgang.

```
20          var loc = (new String(document.location).replace(/http
                :\/\/[^\/]*/, '').replace(/\/examples\/.*$/, '')) +
21          "/cometd";
22          dojox.cometd.init(loc);
31
32          // subscribe and join
33          dojox.cometd.startBatch();
34          dojox.cometd.subscribe("/chat/demo", room, "_chat");
35          dojox.cometd.publish("/chat/demo", {
36              user: room._username,
```

```
37              join: true,
38              chat: room._username + " has joined"
39          });
40          dojox.cometd.endBatch();
```

Listing 3.10: Auszug aus dem CometD-Chat Beispiel (Datei `chat.js`)

3.4.3 Der Extension-Stack

Erweiterungen im Bayeux-Protokoll werden bei der aktuellen Bayeux-Instanz registriert und verhalten sich anschließend wie ein Nachrichtenfilter, welcher bei allen Ein- und Ausgehenden Nachrichten durchlaufen wird:

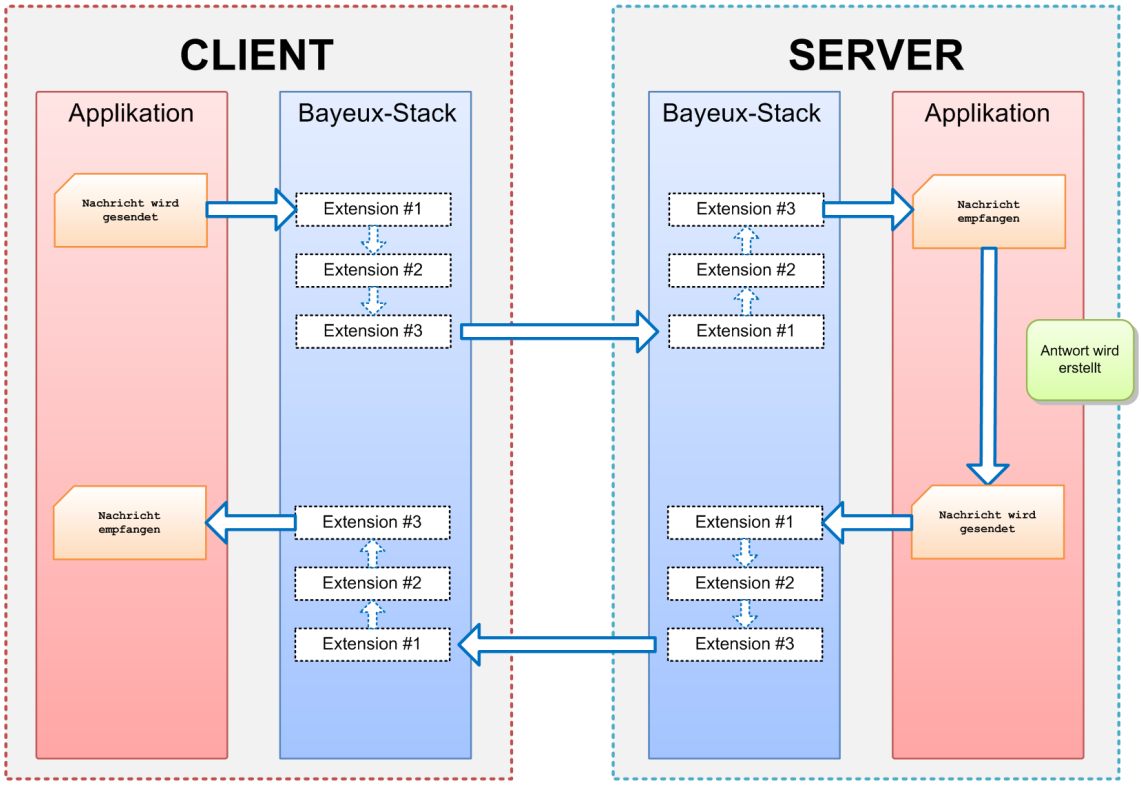


Abbildung 3.3: Aufbau des Bayeux-Extension-Stacks

3.4.3.1 Serverseitige Integration von Erweiterungen

Serverseitig muss eine Klasse erstellt werden, die das Interface *org.cometd.Extension* implementiert. Es gibt 4 Methoden, über welche die Erweiterung mit dem Bayeux-Protokoll-Stack verbunden wird:

1. `public Message rcv(Client from, Message message)`
   Wird bei eingehenden Nachrichten aufgerufen
2. `public Message rcvMeta(Client from, Message message)`
   Wird bei eingehenden Meta-Channel-Nachrichten aufgerufen
3. `public Message send(Client from, Message message)`
   Wird bei ausgehenden Nachrichten aufgerufen
4. `public Message sendMeta(Client from, Message message)`
   Wird bei ausgehenden Meta-Nachrichten aufgerufen

Die Methoden können dabei die Nachricht nach Belieben modifizieren, mit Ihren Attributen im *ext-Feld* versehen bzw. diese auslesen und verarbeiten. Anschließend muss die modifizierte Nachricht als return-Wert zurückgegeben werden. Wenn die Nachricht verworfen werden soll, muss `null` zurückgegeben werden. Eine Extension wird mittels der Methode `addExtension` der Bayeux-Klasse hinzugefügt (wie in Listing 3.8, Zeilen 7 und 8) gezeigt).

3.4.3.2 Clientseitige Implementierung

Clientseitig (dojox.cometd) ist die Extension-Verwaltung noch nicht so weit fortgeschritten. Zwar können auch hier Erweiterungen erstellt und in den Protokoll-Stack eingebunden werden, jedoch gibt es weder eine Basis-Klasse mit default-Handlern, noch Methoden zum organisierten Hinzufügen von Erweiterungen zum Stack. Um eine Erweiterung zu erstellen, muss eine Java-Script Klasse erstellt werden, die die Methoden `function _in(Message msg)` und `function _out(Message msg)` implementiert. Diese Klasse muss anschließend

händisch mittels `push` auf das Extension-Array hinzugefügt werden:

```
dojox.cometd._extendInList.push(dojo.hitch(dojox.cometd._ack, '_in'));
dojox.cometd._extendOutList.push(dojo.hitch(dojox.cometd._ack, '_out'));
```

(`dojo.hitch` ist eine Methode des DOJO-Toolkits, welche eine Wrapper-Funktion erstellt, die die als Zeichenkette übergebene Funktion des übergebenen Objektes aufruft)

Auch das Verwerfen von Nachrichten beim Durchlaufen des Extension-Stacks ist am Client nicht implementiert[10].

3.4.3.3 Das Ext-Feld zur Kommunikation

Die Extension-Module auf dem Client und Server kommunizieren über das ext-Feld der Bayeux-Nachrichten miteinander (siehe Kapitel 3.3.1). Dabei sollte jede Erweiterung Ihre Daten in einem Attribut des ext-Feldes, wiederum als JSON Objekt, ablegen:

```
1  [
2      {
3          "channel" : "/meta/connect",
4          "clientId" : "cl13nt1dg3n3r4t3d",
5          "connectionType" : "long-polling",
6          "id" : "1",
7          "ext" : {
8              auth : { user = "guest", password = "
                   b6b68e049f88aa99a320b4c3ec83aab9" }
9          }
10     }
11 ]
```

Listing 3.11: Beispiel einer Nachricht mit Ext-Feld

[10] In der zum Zeitpunkt des Schreibens aktuellen Version (http://trac.dojotoolkit.org/browser/dojox/trunk/cometd/_base.js?rev=16295#L515) wird, wenn eine Erweiterung `null` liefert, einfach die ursprüngliche Nachricht gesendet. Dies widerspricht zwar der serverseitigen Implementierung, aufgrund einer fehlenden Normierung im Protokoll ist dieses Vorgehen jedoch nicht als fehlerhaft zu beschreiben, sondern lediglich eine andere Art der Implementierung des Erweiterungs-Stacks. Hier sollte das Bayeux-Protokoll noch genauer ausgeführt werden.

Da der Extension-Stack bei jeder gesendeten und empfangenen Nachricht durchlaufen wird (auch bei Meta-Messages), können Erweiterungen in jeder Phase des Bayeux-Protokolls eingreifen und so zum Beispiel bereits beim initialen Handshake mit der Client-Extension Parameter aushandeln.

3.5 Aktuelle Implementierungen

Obwohl das Bayeux-Protokoll noch in einer frühen Phase steckt, existieren bereits mehrere Implementierungen in verschiedenen Technologien[11], welche in Tabelle 3.5 aufgelistet sind.

Programmiersprache	**Client-Implementierung**	**Server-Implementierung**
JavaScript	cometd-dojox (stable) cometd-jquery (unstable) ExtJS	cometd-js (unstable)
Java	cometd-jetty (stable)	cometd-java (stable) cometd-jetty (stable) Glassfish / Grizzly (andere) BEA (andere) IBM Websphere (andere)
Groovy	-	Cometd-Grails-Plugin
Python	-	cometd-twisted (unstable)
Perl	-	cometd-perl (unstable)
Flash / Flex	flexcomet (andere)	-

Tabelle 3.1: Übersicht über aktuelle Bayeux-Implementierungen

Hauptaugenmerk legt diese Arbeit auf die zum Zeitpunkt des Schreibens am weitesten fortgeschrittenen Implementierungen der Protokoll-Ersteller Alex Russel (`cometd-dojox`) und Greg Wilkins (`cometd-java` bzw. `cometd-jetty`). Basierend auf diesen Entwicklungen wird auch die Forschung und Erweiterung in den folgenden Kapiteln durchgeführt.

Andere Entwicklungen, welche zwar eine Comet-Technologie einsetzen, jedoch nicht auf dem Bayeux-Protokoll basieren, wären unter anderem *Orbited*[12] und *Liberator*[13].

[11] laut http://cometdproject.dojotoolkit.org/documentation/cometd haben unter anderem die Firmen IBM,Sun und BEA eigene Versionen des Protokolls implementiert

[12] siehe http://www.orbited.org/

[13] siehe http://www.freeliberator.com/

Kapitel 4

Authentifizierung und Autorisierung im Bayeux Protokoll

4.1 Einleitung

In der heutigen IT-Welt sind Sicherheitsaspekte kaum mehr wegzudenken. Vor allem in Mehrbenutzer-Anwendungen muss ein gewisses Level an Sicherheit geboten werden, um private Daten vor Fremdzugriffen zu schützen. So weisen viele der aktuell im Internet eingesetzten Protokolle ihre Authentifizierungs-Methoden auf:

- `HTTP`: Username/Passwort Authentifizierung über HTTP-Basic/Digest[1]
- `SMTP`: Das Simple-Mail-Transfer-Protocol bietet diverse Authentifizierungsmaßnahmen (User/Passwort, TLS)[2]

[1] siehe RFC 2617: http://www.ietf.org/rfc/rfc2617.txt
[2] siehe RFC 4954: http://tools.ietf.org/rfc/rfc4954.txt

- `XMPP`: Das eXtensible Messaging und Presence Protokoll bietet Authentifizierung basierend auf dem SASL-Protokoll[3] an[4].

Auch im Bayeux-Protokoll sollte daher, wie in Kapitel 3.2 bereits erläutert wurde, eine Möglichkeit zur Benutzerauthentifizierung und Serviceautorisierung angedacht werden. Dazu werden zunächst mögliche Implementierungs-Wege gesucht und evaluiert.

4.2 Evaluierung der Möglichkeiten

Als Basis für die Evaluierung muss auch die Frage des verwendeten Transports noch einmal aufgegriffen werden. Wie bereits erwähnt, schreibt das Bayeux-Protokoll zwar keinen speziellen Transport vor, jedoch wird in der Spezifikation[5] ausschließlich auf HTTP als verwendeten Transport eingegangen. Dies schränkt einerseits die in Frage kommenden Authentifizierungsmaßnahmen ein, bietet aber auf der anderen Seite auch eine Vereinfachung der Implementierung.

Ein weiterer wichtiger Punkt bei der Entscheidung über die zu verwendende Authentifizierungs methode ist die Frage, mit welcher Granularität eine Authentifizierung der Bayeux-Übertragungen notwendig ist (Beschränken des Zugriffs an sich oder beschränken auf Channel/Service-Ebene).

Nach erfolgter Analyse des Bayeux-Protokolls und Studium der Referenzimplementierungen wären folgende 2 Arten der Implementierung von Authentifizierung und Autorisierung naheliegend:

1. Authentifizierung auf Anwendungs-Ebene
2. Authentifizierung auf Protokoll-Ebene

[3] Simple Authentication and Security Layer, siehe RFC2222: http://www.ietf.org/rfc/rfc2222.txt
[4] siehe RFC 3920: http://www.ietf.org/rfc/rfc3920.txt
[5] siehe Russel [2008a], Kapitel 1.4.1/1.4.2

4.3 Authentifizierung auf Anwendungs-Ebene

Hierbei wird der Bayeux-Protokoll-Stack nicht verändert, sondern auf Anwendungs-Ebene eigene Authentifizierungs-Routinen eingeführt. Die Meta-Handshakes des Bayeux-Protokolls bleiben dabei völlig unberührt, mit dem Nachteil, dass keine spezifischen Authentifizierungen durchgeführt werden können, da der Protokoll-Inhalt mit diesen Methoden nicht verändert werden kann. Somit kann durch diese Formen der Authentifizierung prinzipiell der Zugang zum Comet-Kanal zwar reguliert werden, aber es können keine einzelnen Inhalte (Bayeux-Channels) gesperrt werden. Für eine einfache Applikation, bei der es darum geht, nur registrierten Benutzern Zugriff zu gewähren, wäre dies aber durchaus ausreichend.

4.3.1 HTTP-Basic / -Digest Authentifizierung

HTTP-Basic bzw. HTTP-Digest ist eine einfache Authentifizierungsform, die von jedem aktuellen Web-Browser unterstützt wird. Am einfachsten wird (im Falle eines Apache Web-Servers) per htaccess-Regeln[6] , welche auf bestimmte URLs matchen, eine Benutzer-/Passwortabfrage gelegt. Bei einem Jetty- Webserver kann über die `web.xml` mittels Security-Constraints[7] konfiguriert werden. Sinn dieser Beschränkungen ist es, den Zugriff auf den CometD Kanal nur dann freizugeben, wenn der Benutzer im Web-Browser die nötigen Zugangsdaten eingibt.

Im Falle einer CometD/Bayeux Applikation wäre diese Art der Authentifizierung zwar möglich, jedoch wenig zielführend; in den meisten Fällen wird es sich dabei um eine Rich-Internet-Applikation handeln und dadurch ist hier die HTTP-Basic/Digest Abfrage des Browser hier sowohl Designtechnisch, als auch Programmiertechnisch (da nicht kontrollierbar) nicht passend. Zwar können HTTP-Basic-/Digest-Authentifizierungen auch per XMLHttpRequest-Objekt durchgeführt werden, dies würde aber wiederum eine Änderung des CometD/Bayuex Source- Codes bedeuten und wäre somit keine reine Methode auf Anwendungsebene mehr.

[6] siehe http://httpd.apache.org/docs/2.2/howto/htaccess.html
[7] siehe http://docs.codehaus.org/display/JETTY/How+to+Configure+Security+with+Embedded+Jetty

4.3.2 Login-Service und SecurityPolicy

In der *java-server* Implementierung des Bayeux-Protokolls existiert eine (leider undokumentierte) Security-Policy-Klasse, welche für einfach Sicherheitsabfragen herangezogen werden kann. Um diese auch mit Benutzerdaten zu versorgen, kann zum Beispiel ein einfacher Login-Service kreiert werden.

Die `Security-Policy` bietet unterschiedliche Callback-Methoden, über welche Berechtigungsabfragen durchgeführt werden können:

- `public boolean canHandshake(Message msg)`
  wird aufgerufen, wenn eine Handshake-Nachricht von einem Client empfangen wird
- `public boolean canCreate(Client client, String channel, Message msg)`
  wird aufgerufen, wenn ein neuer Channel durch einen Client erstellt werden soll
- `public boolean canSubscribe(Client client, String channel, Message msg)`
  wird aufgerufen, wenn ein Client auf einen Channel subscriben möchte
- `public boolean canPublish(Client client, String channel, Message msg)`
  wird aufgerufen, wenn ein Client eine Nachricht in einem Channel veröffentlichen möchte

Alle Funktionen müssen entweder `true` für *erlauben* oder `false` für *verweigern* liefern. Es sollte jedoch bei selbstständiger Implementierung einer Security-Policy auch die Default-Policy mitberücksichtigt werden, da in dieser die Abfragen, ob zum Beispiel ein Client auf einen Meta-Channel subscriben darf, passieren.

Da die Aufrufe der Policy aber nur die Nachrichten und (bis auf `canHandshake`) die Bayeux-Client Instanz mitbekommen, kann hier von der Applikation nur schwer eine Entscheidung über eine Authentifizierung getroffen werden. Daher muss auf eine andere Art und Weise eine Benutzerkennung durch den Client übermittelt werden. Bis diese Kennung empfangen wurde, sollten alle Anfragen an die Security-Policy mittels `false` verweigert werden.

Eine Benutzerkennung könnte zum Beispiel über einen Service-Channel (z.B.: `/service/login`) empfangen werden.

Der Ablauf für die Client-Seite wäre Folgender:

1. Der Benutzer verbindet sich ganz normal zum Bayeux-Service
2. Vor etwaigen subscribe oder publish-Ereignissen wird eine Nachricht an den Service `/service/login` gesandt, welche Benutzername und -passwort enthält. (Passwort nach Möglichkeit nicht Plain-Text, z.B MD5)
3. Dieser Service teilt dem Client mit, ob die Authentifizierung erfolgreich war oder nicht. Serverseitig wird eine Map mit Client-ID und Credentials gefüllt, um später dem Client Berechtigungen zuordnen zu können.
4. Der Client kann versuchen, Channels zu abonnieren und Nachrichten zu veröffentlichen (sofern die nachfolgende Autorisierung gut geht).

Serverseitig müsste eine Applikation das Interface `Security-Policy` implementieren und die Default-Policy der Bayeux-Instanz ersetzen. Weiters muss ein Listener für den Service `/service/login` erstellt werden, welcher die Benutzerdaten validiert.

4.4 Authentifizierung auf Protokoll-Ebene

Auch wenn durch die vorhergehenden Maßnahmen grundlegender Schutz für die Applikation und den Comet-Kanal gewährleistet werden kann, so sollte für genügend Privatsphäre und Sicherheit eine Authentifizierung auf Protokoll-Ebene durchgeführt werden. Außerdem kann hier der generelle Ablauf besser kontrolliert und mehr Möglichkeiten in Punkto Sicherheit geboten werden.

4.4.1 Entwicklung einer Bayeux-Extension

Basierend auf dem in Kapitel 3.4.3 vorgestellten Erweiterungs-Stacks des Bayeux-Protokolls kann eine sichere und trotzdem sehr stark anpassbare Authentifizierungs-Lösung erstellt werden. Grundlage dafür ist das Verständnis des Bayeux-Verbindungsaufbaues, des Extension-Stacks und der Wege einer Nachricht.

Die Auth-Extension soll ähnliche Möglichkeiten wie die bereits in der Java Implementierung entwickelte Security-Policy bieten, jedoch implementierungsunabhängig in das Protokoll eingebunden werden. Damit auch beliebige Clients mit dieser Extension arbeiten können, sollte sowohl eine interne als auch eine externe Authentifizierung ermöglicht werden. Bereits beim initialen Handshake[8] prüft die Server-Extension, ob bereits Benutzerdaten (z.B. in Form einer Session/Cookie) in der Applikation vorliegen oder ob diese erst abgefragt werden müssen. Je nachdem wird der Client instruiert, die Benutzerdaten in der Connect-Nachricht[9] mitzusenden oder nicht.

Um die Bedürfnisse verschiedener Applikationen optimal abbilden zu können, sollte der Grad der Authentifizierung festgelegt werden können:

- **mandatory**: Eine Benutzer-Authentifizierung ist zwingend notwendig. Ohne gültige Daten wird die Verbindung verweigert.
- **optional**: Eine Benutzer-Authentifizierung kann durchgeführt werden, ist für eine Verbindungs-Herstellung aber nicht zwingend erforderlich (Verbindung mit eingeschränkten Rechten).
- **none**: Es ist keine Benutzer-Authentifizierung erforderlich. Dadurch ist nur die Channel-Authentifizierung aktiv.

Zur Autorisierung auf Channel-Ebene können 2 Möglichkeiten verwendet werden:

1. es können die in der Authentifizierung gewonnenen Benutzerdaten verwendet werden

[8] siehe Kapitel 3.3.1, Seite 24
[9] siehe Kapitel **??**, Seite 27

2. der Channel kann durch eine Pass-Phrase geschützt sein, die zur Laufzeit vom Client abgefragt wird

In beiden Fällen wird bei jeglichen Ereignissen auf den Channels (publish/subscribe) serverseitig eine Methode der Benutzer-Applikation aufgerufen, welche die Art der Autorisierung für den Kanal festlegt. Wird eine Passphrase verlangt, so wird die Benutzeraktion mit einem Authentifizierungs-Fehler und einem Request-der Passphrase abgebrochen. Der Client hat die Möglichkeit (nach Abfrage der Passphrase vom Benutzer) den Request zu wiederholen.

4.5 Bewertung der Alternativen

Auch wenn mittels der gebotenen Möglichkeiten in den Implementierungen von Java/DojoX eine Authentifizierung und Autorisierung ohne die Extension abgewickelt werden kann, so ist dies doch eine sehr spezielle Lösung, welche stark an diese Infrastruktur gebunden ist. Hier ist eine allgemeine Erweiterung, welche spezifisch für die verschiedenen Plattformen implementiert werden kann, wesentlich universeller einsetzbar.

Bezüglich des Grades der Sicherheit sind meiner Ansicht nach beide Möglichkeiten nahezu gleichzusetzen, auch wenn es sich als Nachteil erweist, auf Anwendungsebene Channels z.B. nicht durch eine Pass-Phrase schützen zu können.

Nachfolgend wird deshalb die Erweiterung des Protokolls zunächst allgemein (theoretisch) erläutert, und anschließend für das Client-Server Paar Jetty/Dojo implementiert.

4.6 Referenzimplementierung

4.6.1 Architektur

Die Implementierung der Extension wird in 2 Teile zerlegt, die aber miteinander kommunizieren können müssen. Diese sind die *Authentifizierung* und *Autorisierung*.

4.6.1.1 Authentifizierung

Die Authentifizierung ist der erste Schritt, den die Extension durchführt, um die Identität des Benutzer feststellen zu können. Hierfür müssen im initialen Handshake Client- und Server-Extension abgleichen, ob sie zueinander kompatibel sind (Version überprüfen). Dies kann, vor allem wenn zu einem späteren Zeitpunkt mehrere Authentifizierungs-Möglichkeiten implementiert werden, Fehler in der Authentifizierung vermeiden.

Dazu versenden sowohl Client als auch Server in den jeweiligen Handshake-Nachrichten eine entsprechende Struktur im ext-Field:

```
1    auth: {
2        version : '0.1'
3    }
```

Listing 4.1: Ext-Feld Inhalt für die Auth-Extension, Client-Anfrage

Der Server hat zu Beginn die Entscheidung über die Authentifizierung an die Applikation abzugeben. Hierfür wird eine Schnittstelle zur Applikation aufgerufen, welche eine der folgenden Antworten liefern muss:

- **ALREADY_AUTHENTICATED**: Die Applikation konnte den Benutzer bereits (z.B. aufgrund von Informationen in einer Session) identifizieren, es ist keine weitere Authentifizierung notwendig
- **MANDATORY**: Der Benutzer muss vom Protokoll authentifiziert werden
- **OPTIONAL**: Der Benutzer kann sich authentifizieren, muss aber nicht (z.B. eingeschränkte "Gast"-Verbindung)
- **NONE**: Es ist keine Benutzer-Authentifizierung notwendig und es liegen auch keine Informationen über den Benutzer in der Applikation vor

In den Fällen von MANDATORY und OPTIONAL wird dem Client folgende Antwort gegeben:

```
1    auth: {
2        version : '0.1',
3        type : 'mandatory',
4        secret : '47ef11ab08',
5        hashtype : 'md5'
6    }
```

Listing 4.2: Ext-Feld Inhalt für die Auth-Extension, Server-Antwort

Liefert die Schnittstelle der Benutzer-Applikation ALREADY_AUTHENTICATED, so wird dem Client mit `type : 'none'` geantwortet und eine weitere Schnittstelle zum Laden einer eindeutigen Benutzerkennung aufgerufen. Diese wird für nachfolgende Autorisierungsanfragen benötigt.

Wünscht die Applikation keine Authentifizierung (NONE) so wird dies auch dem Client durch `type : 'none'` mitgeteilt.

Wenn vom Client eine Authentifizierung gefordert wird, so ruft die dortige Erweiterung eine Schnittstellenfunktion auf, die die Benutzerdaten bereitstellt. Das Passwort wird mittels der angegebenen Hash-Funktion, unter Beigabe des Secrets, gehasht und gemeinsam mit dem Benutzernamen zum Server übertragen.

Die Serverseite gibt die empfangenen Daten zur Applikation weiter, welche die Überprüfung der Benutzerdaten durchführt. Stimmen die Daten überein, so liefert die Applikation als Rückgabewert einen eindeutigen Identifier für den Benutzer (z.B. dessen ID/GUID aus der Datenbank) für spätere Autorisierungsanfragen. Im Fehlerfall wird NULL (oder ein Äquivalent in der jeweiligen Programmiersprache) geliefert.

Dem Client wird der "connect-Request" mit einer entsprechenden Nachricht bestätigt oder abgelehnt.

Abbildung 4.1 erläutert diese Vorgehensweise graphisch zum besseren Verständnis.

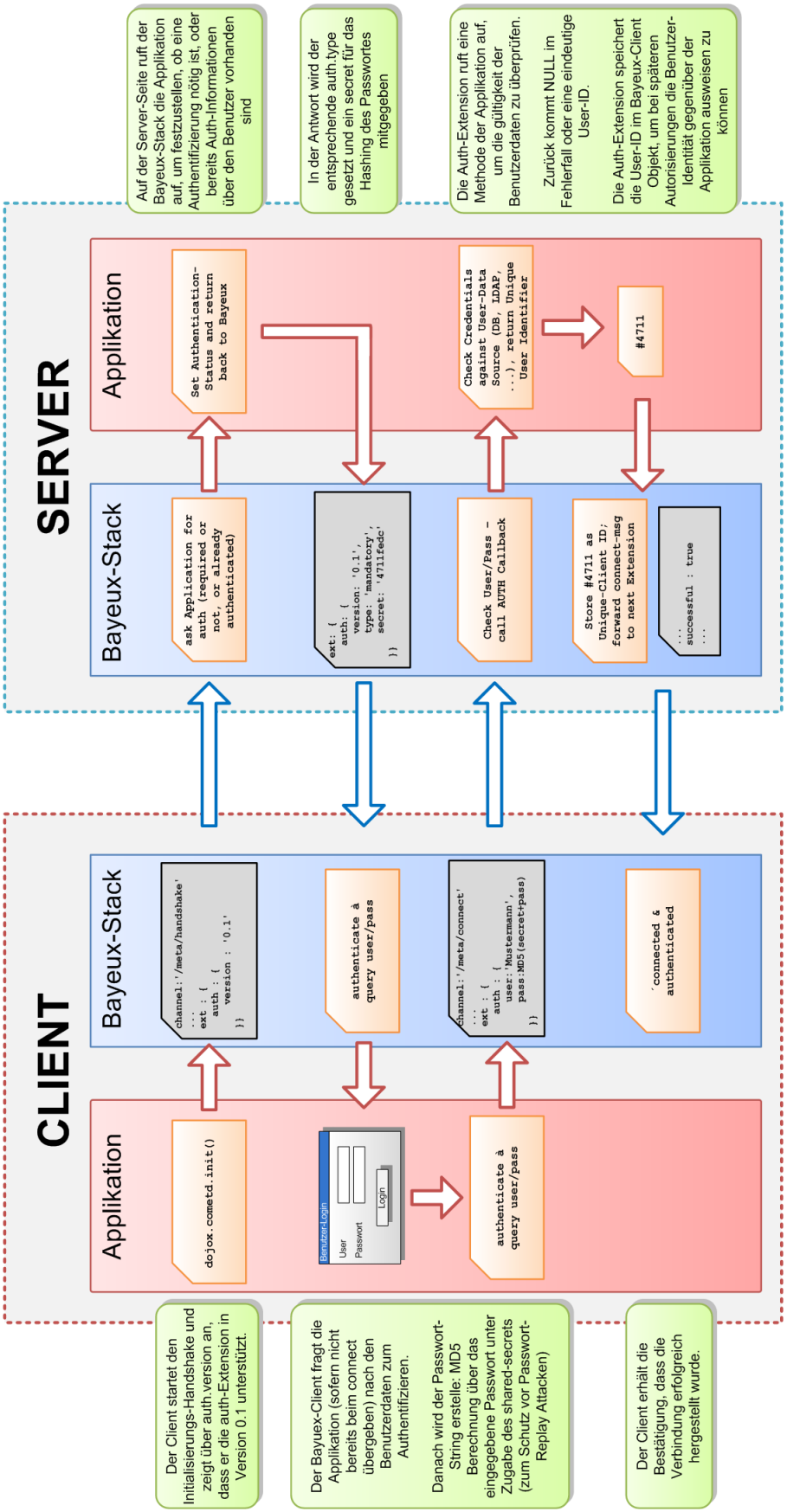


Abbildung 4.1: Authentifizierungs-Vorgang in der Auth-Extension

4.6.1.2 Autorisierung

Ist die Authentifizierung erfolgt, können auch Autorisierungsanfragen abgewickelt werden. Für die Aktionen subscribe und publish bietet die Server-Extension je eine Schnittstelle zur Applikation, welche die Autorisierung des Clients übernimmt. Damit diese auch weiß, um welchen Client es sich handelt, wird der jeweiligen Schnittstellenfunktion auch die eindeutige Client-ID, sowie der betroffene Kanal übergeben.

Keine vorhandene Client-ID bedeutet nicht automatisch keine Berechtigung, so muss auch ohne Client-ID (Parameter-Wert NULL) die Abfrage durchgeführt werden, da der Channel auch durch eine Passphrase geschützt sein könnte. Wurde die Passphrase vom Client übermittelt, wird diese als optionaler dritter Parameter ebenfalls an die Schnittstelle übergeben. Die Applikation liefert als Antwort einen der folgenden Werte:

- **AUTHORIZED**: Die Autorisierung wurde erfolgreich durchgeführt, der Benutzer darf die gewünschte Aktion durchführen.
- **PASSPHRASE_REQUIRED**: Die Autorisierung konnte nicht durchgeführt werden, es ist eine Passphrase notwendig bzw. die übergebene Passphrase ist ungültig!
- **DENIED**: Die Autorisierung wurde abgelehnt.

Wurde die Autorisierung erfolgreich durchgeführt, so gibt die Erweiterung die Nachricht des Clients zur weiteren Verarbeitung frei und die gewünschte Aktion wird durchgeführt. Wird hingegen eine Passphrase verlangt oder die Autorisierung vollständig abgelehnt, so wird die Nachricht verworfen und der Umstand dem Client entsprechend durch eine Fehlermeldung mitgeteilt. Die Client-Implementierung der Extension verarbeitet diese Fehlermeldung und ruft eine Callback Funktion der Applikation auf, um diese über die gescheiterte Autorisierung zu informieren (und dieser auch somit eine erneute Möglichkeit zur Autorisierung zu geben; vgl. Listing 4.3).

```
1    successful : false,
2    error : '412::Passphrase Required'
```

Listing 4.3: Kanal-Autorisierung: Fehlermeldung bei falscher/fehlender Passphrase

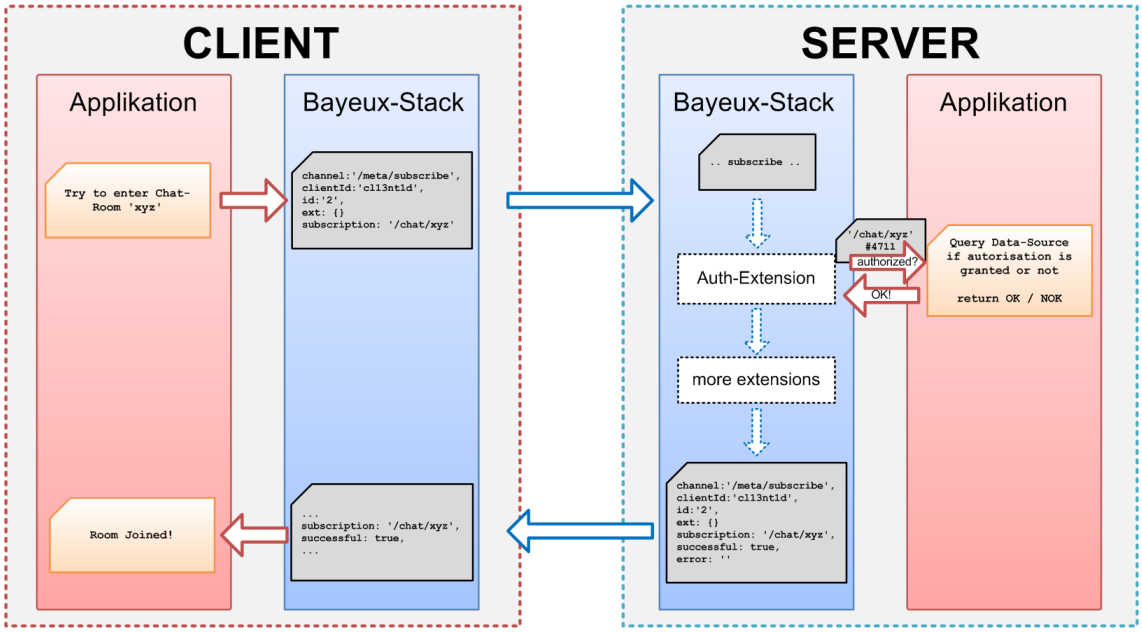


Abbildung 4.2: Autorisierungs-Vorgang in der Auth-Extension

Werden Berechtigungen während einer aktiven Benutzersitzung entzogen, so wirken sich diese *nicht* auf bereits autorisierte Clients aus, da dies Abfrage nur bei initialen Requests (subscribe) durchgeführt wird!

Die detaillierte Umsetzung dieses Konzeptes hängt von den jeweiligen Basis-Implementierungen des Bayeux-Protokolls ab. Je nach Möglichkeit sollte aber dem Applikations-Entwickler immer eine, diesen Anforderungen entsprechende, Schnittstelle für die Entwicklung geboten werden.

4.6.2 Server

4.6.2.1 Authentication & Authorization Extension

Als Basis für die serverseitige Extension muss zunächst eine allgemeine Extension implementiert werden, welche direkt in das Bayeux-Protokoll eingebunden wird, um bereits im Handshake-Prozess eingreifen zu können. Die wesentlichen Aufgaben dieser Erweiterung sind:

- Erkennen ob der Client Authentifizierung unterstützt
- Einfügen des Authentifizierungs-Typs und des Shared-Secrets
- Abwickeln der Authentifizierung mit der Benutzer-Applikation
- Intallieren eines Authorisierungs-Handlers in der Bayeux-Client Instanz

Da der eingehende Request auf `/meta/handshake` bzw. `/meta/connect` ohnehin keine Daten verändert, reicht es, direkt auf die jeweiligen Antworten in Richtung Client zu reagieren und zum Beispiel bei der Antwort auf den Connect-Request das Attribut `successful` im Fehlerfall auf `false` zu setzen. Somit wird in dieser Extension nur die Funktion `public Message sendMeta(Client from, Message msg)` implementiert (siehe AuthenticationAuthorizationExtension.java).

Für die Kommunikation mit der Applikation werden je ein Interface für Authentifizierung und Autorisierung erstellt (siehe Listing 4.4 und 4.5).

```
1  public interface AuthenticationListener
2  {
3      public static final int ALREADY_AUTHENTICATED = 0;
4      public static final int AUTHENTICATION_MANDATORY = 1;
5      public static final int AUTHENTICATION_OPTIONAL = 2;
6      public static final int NONE = 3;
7
8      public int getUserAuthenticationType(Client c);
9      public Object checkCredentials(String username, String password, String secret,
           String hashType);
10     public Object getUserAuthenticationToken(Client c);
11 }
```

Listing 4.4: Interface für Authentifizierung

```
1  public interface AuthorizationListener
2  {
3      public static final int DENIED = 0;
4      public static final int ALLOWED = 1;
5      public static final int PASSPHRASE_REQUIRED = 2;
6
7      public int subscriptionAllowed(Object token, String channel);
8      public int publishAllowed(Object token, String channel, Message msg);
9      public boolean checkPassPhrase(Object token,String channel,String passPhrase);
10 }
```

Listing 4.5: Interface für Autorisierung

Um dem Client über eventuell aufgetretene Fehler bei der Authentifizierung und Autorisierung zu informieren, wurden folgende Fehlermeldungen definiert:

- **411::Authentication Required** ... wird gesendet, wenn keine oder ungültige Benutzerdaten angegeben wurden
- **412::Passphrase Required** ... wird gesendet, wenn für die gewählte Aktion eine Passphrase notwendig ist oder die übergebene Passphrase ungültig ist
- **413::Access Denied** ... wird gesendet, wenn der Zugriff auf die gewählte Ressource verweigert wurde

4.6.2.2 Authorization Client Extension

Wurde die Authentifizierung erfolgreich durchgeführt, so wird der Bayeux-Client Instanz ebenfalls eine Extension installiert, welche ab nun die Aufgabe hat, sich um Autorisierungs-Anfragen zu kümmern. Dieser Instanz wird der aus dem Authentifizierungs-Handshake von der Applikation erhaltene Benutzer-Token mitgeteilt, um den Client bei folgenden Autorisierungen identifizieren zu können.

Der naheliegendste Ansatz für eine Implementierung wäre gewesen, direkt die jeweiligen Aktionen (publish/subscribe) über die Funktionen `public Message rcv(Client from, Message message)` und `public Message rcvMeta(Client from, Message message)` abzufangen und eine Autorisierungsabfrage durchzuführen. Da jedoch die aktuelle Bayeux-Implementierung keinen Eingriff in die Handshakes durch eine Extension zulässt (ein connect/subscribe kann **nicht** durch eine Extension abgebrochen werden, liefert die Extension `null`, so wirft die subscribe-Funktion eine Exception) muss auf die in Kapitel 4.3.2 erwähnte SecurityPolicy zurückgegriffen werden.

Diese wurde bereits von der generellen *AuthenticationAuthorizationExtension* implementiert und installiert, und leitet nun entsprechende Autorisierungs-Anfragen über die Client-Extension an die Applikation weiter. Je nach Ereignis wird entweder die Aktion zugelassen oder im Fehlerfall eine Fehlermeldung in der Client-Extension gespeichert, damit diese beim Senden der Antwort Nachricht eingefügt werden kann.

4.6.3 Client

4.6.3.1 Allgemeiner Aufbau

Die Client-Extension wurde aufgrund einer Eigenheit beim Laden von JavaScript Objekten in der dojo-Library (Objektnamen müssen gleich Pfadnamen sein) im Namespace `bayeux.cometd` erstellt, um diese nicht direkt in die Original-Library platzieren zu müssen.

Die Client-Extension beruht auf folgendem einfachen Funktionsprinzip:

- Der Client zeigt in der Meta-Handshake Nachricht an, dass er Authentifizierung unterstützt
- Der Server gibt die Authentifizierungs-Parameter bekannt
- Der Client sendet in der Meta-Connect Nachricht die Benutzerdaten zur Authentifizierung mit
- Der Server bestätigt oder verweigert die Authentifizierung
- Wenn notwendig sendet die Extension eine Passphrase für bestimmte Tätigkeiten mit

4.6.3.2 Authentifizierung

Wie auch bereits bei der Server-Implementierung, so sind auch auf der Client Seite einige Probleme bei der Implementierung aufgetreten, welche durch geschicktes Verändern bestimmter Eigenschaften des Protokolles umgangen werden konnten.

So sollten zum Beispiel Benutzerdaten für eine Authentifizierung bereits *vor* dem Aufruf der Funktion `dojox.cometd.init` an die Extension übergeben werden, da eine Abfrage der Benutzerdaten während des Handshakes nicht ohne weiteres durchgeführt werden kann (da in Java-Script keine Blocking-Events möglich sind und ein asynchrones Aussetzen und Wiederaufnehmen des Connnection-Handshakes durch eine Extension nicht möglich

ist). Hierfür wurde eine einfache Funktion `dojox.cometd._auth.setCredentials( username,password)` zur Verfügung gestellt. Die Durchführung der Authentifizierung zeigt Listing 4.6.

```
1  if (msg.channel == "/meta/handshake")
2  {
3      if (!msg.ext)
4          msg.ext = {};
5      msg.ext.auth = { version : '0.1' };
6  }
7
8  if(msg.channel == "/meta/connect")
9  {
10     // we have to provide the auth credentials if neccessary
11     if(authType == 'mandatory' || (authType == 'optional' && username != ''))
12     {
13         if(!msg.ext)
14             msg.ext = {};
15         msg.ext.auth = { username : username, password : MD5(password+secret) };
16
17     }
18 }
```

Listing 4.6: Auszug der _out Method der Client-Authentifizierungs-Extension

Eingehende Fehlermeldungen werden im entsprechenden Teil der `_in` Methode verarbeitet (siehe Listing 4.7). Hier musste bereits das erste kleinere Problem umgangen werden. Damit durch die standardmäßig vom Server gesendete Advice-Policy nicht ein erneuter Verbindungsversuch gestartet wird (da der Bayeux-Client nicht weiß, dass die Authentifizierung fehlgeschlagen ist) muss im Fehlerfall der gesamte CometD Kanal abgebrochen werden. Der Fehler wird per `onError` Schnittstelle an die Applikation weitergegeben.

```
1  // handle responses to the connect request (only if not already connected)
2  if(msg.channel == "/meta/connect" && dojox.cometd._status != 'connected')
3  {
4      // if we've got an error
5      if(!msg.successful)
6      {
7          // cancel the whole cometd connection
8          dojox.cometd.disconnect();
9          // get the error
10         var err = msg.error.split(':');
11         // and pass it to the application
12         this.onError(err[0], err[2], msg);
13     }
```

```
14      else
15      {
16          // fire the connect-handler, not a very clean solution
17          // yet, has to be improved! (setTimeout is neccesary for
18          // the bayeux-client to set up the connection-internals)
19          window.setTimeout(dojo.hitch(this, 'onConnect'), 500);
20      }
21  }
```

Listing 4.7: Auszug der _in Method der Client-Authentifizierungs-Extension

4.6.3.3 Autorisierung

Autorisierungsanfragen werden nach erfolgter Authentifizierung am Server direkt abgehandelt. Wird der Zugriff auf eine Ressource aber verweigert oder ist eine Passphrase für den Zugriff notwendig, so wird eine Fehlermeldung an den Client propagiert. Da die Bayeux-Client Implementierung sehr restriktiv programmiert ist (beinahe alle `successful : false` resultieren in einem kompletten Verbindungsabbruch), muss wieder ein Workaraound geschaffen werden: In der Extension wird zunächst der Channel aus der Nachricht gelöscht (z.B. auf 'none' gesetzt) und anschließend das Attribut `successful` wieder auf `true` gesetzt. Dadurch kann die Nachricht vom Bayeux-Protokoll ohne Fehler weiterverarbeitet werden (welche aber keinen Effekt hat, da der Channel ohnehin nicht zugeordnet werden kann) und die Extension kann über die `onError`-Schnittstelle die Applikation über den Fehler informieren (vgl. Listing 4.8).

```
1   // look for errors...
2   if(!msg.successful && msg.channel != "/meta/connect" && msg.channel != "/meta/
        handshake")
3   {
4       // ... and check if it is an auth error
5       if(msg.error)
6       {
7           var err = msg.error.split(':');
8           if(err[0] == "411" || err[0] == "412" || err[0] == "413"))
9           {
10              // clear the channel
11              msg.channel = "none";
12              // reset successful to true
13              msg.successful = true;
```

```
14              // pass the error to the application
15              this.onError(err[0], err[2], msg);
16          }
17      }
18  }
```

Listing 4.8: Fehlbehandlung in der _in Method der Client-Authentifizierungs-Extension

Um Autorisierungen über die Passphrase abwickeln zu können, bietet die Client-Extension die Möglichkeit, per `bayeux.cometd._auth.setPassPhrase(channel,phrase)` diese für einen bestimmten Kanal zu setzen. Ausgehende Nachrichten für den entsprechenden Channel werden automatisch mit der gesetzten Passphrase versehen.

Kapitel 5

Zuverlässigkeit und Integrität der Daten im Bayeux Protokoll

5.1 Einleitung

Bei manchen Anwendungsgebieten kann es nötig sein, dass eine fehlerfreie Übertragung und Zustellung aller Nachrichten sichergestellt werden muss. In der aktuellen Version des Bayeux-Protokolls wurde für solche Fälle jedoch noch keine Vorgehensweise definiert Mit der Frage, wie man trotzdem zu den erwähnten Features kommt und mit den Möglichkeiten der Implementierung im Bayeux-Protokoll beschäftigen sich nachfolgende Forschungen und Analysen.

5.2 Evaluierung der Möglichkeiten

Generell gibt es verschiedene Ansätze, welche eine zuverlässige Nachrichtenzustellung ermöglichen können. Dabei kommt es darauf an, welche Ziele man verfolgt und wie viele Ressourcen man für das Erreichen dieser Ziele opfern möchte.

Um die nachfolgende Möglichkeiten möglichst Ressourcen schonend einsetzen zu können, sollte jeder Client für sich entscheiden können, welches Maß an Zuverlässigkeit er für seine

Unterhaltungen für nötig hält.

5.2.1 Delayed Acknowledge

Beim so genannten *Delayed Acknowledge*[1] wird nicht jede Nachricht sofort mit einem Acknowledge bestätigt, sondern es wird bei der nächsten Übertragung die höchste Empfangene Nachrichten-ID bestätigt. Bei diesem Ansatz werden die verwendeten Ressourcen die für das Acknowledge-Verfahren benötigt werden minimal gehalten, da das ACK-Bit nur als Zusatz- Attribut zu einer normalen Daten-Payload angehängt wird. Serverseitig werden alle unbestätigten Nachrichten in einer Warteschlange solange aufgehoben, bis eine Bestätigung eingeht (Nachrichten-IDs werden im Bayeux-Protokoll aufsteigend vergeben und müssen pro Verbindung zwischen Client und Server eindeutig sein). Nachfolgende Szenarien zeigen verschiedene Fehler und wie das Delayed Acknowledge diese (nicht) beheben kann.

5.2.1.1 Recovery fehlender Nachricht(en)

Wenn ein ACK für eine zu niedrige ID vom Client gesendet wird, so veranlasst der Server ein erneutes Senden *aller unbestätigten* Nachrichten (siehe Abbildung 5.1):

1. Der Client sendet eine Nachricht an den Server und ein ACK für die zuletzt erhaltene Nachricht (*#47*)
2. Der Server sendet Nachricht #48 zum Client und fügt diese zur Liste nicht bestätigter Nachrichten hinzu
3. Der Client merkt sich #48 als höchste unbestätigte Nachricht
4. Der Server sendet Nachrichten #49, #50 zum Client und fügt diese wiederum zur Liste der nicht bestätigten Nachrichten hinzu

[1] Ein ähnlicher, ebenfalls mit delayed Acknowledge benannter Algorithmus kommt unter anderem beim TCP Protokoll zur Anwendung, siehe RFC 1122 bzw. RFC 2581

5. Nach Nachricht 48 bricht die Verbindung ab, Nachricht #49 und #50 kommen nicht an
6. Der Client sendet eine Nachricht an den Server und ein ACK für *#48*
7. Der Server bemerkt die zu niedrige ACK-ID und sendet die Nachrichten #49 und #50 erneut

Wie in diesem Beispiel ersichtlich wird, muss der Server für den Delayed-ACK Mechanismus pro Client eine Warteschlange zur Verfügung stellen. Je nach Zeitpunkt des ACKs vom Client wird diese Liste ständig wachsen. Deshalb sollte hier von der Implementierung entweder eine Beschränkung des Puffers (z.B. Ringpuffer mit fixer Größe) oder aber eine zeitliche Limitierung für das ACK einer Nachricht eingeführt werden. Diese Parameter sollten konfigurierbar sein, um je nach Applikation / Server-Ressourcen eine optimale Einstellung vornehmen zu können.

Auch der Client sollte sich mit einem ACK nicht unbedingt bis zur nächsten zu sendenden Nachricht Zeit lassen, sondern ein Timeout implementieren, welches zum Beispiel nach spätestens 10 Sekunden ein ACK für erhaltene Nachrichten sendet. Dadurch kann die Puffergröße am Server wiederum kleiner gewählt werden und somit ressourcensparender gearbeitet werden.

5.2.1.2 Verlorene Nachrichten

Geht eine Nachricht aus einer Serie von Nachrichten verloren, so wird dies durch das Delayed Acknowledge nicht bemerkt:

Der Ablauf in Abbildung 5.2 ist ähnlich dem zuvor, jedoch geht diesmal nur eine Nachricht verloren und die letzte Nachricht wird zugestellt. Da der Delayed-ACK Mechanismus auf der aufsteigenden Nachrichten-ID aufbaut und die letzte Nachricht per ACK bestätigt wird, fällt die verloren gegangene Nachricht weder am Client noch am Server auf. Somit ist diese Methode zwar Ressourcen schonend, kann aber zwischendurch verloren gegangene Nachrichten nicht erkennen.

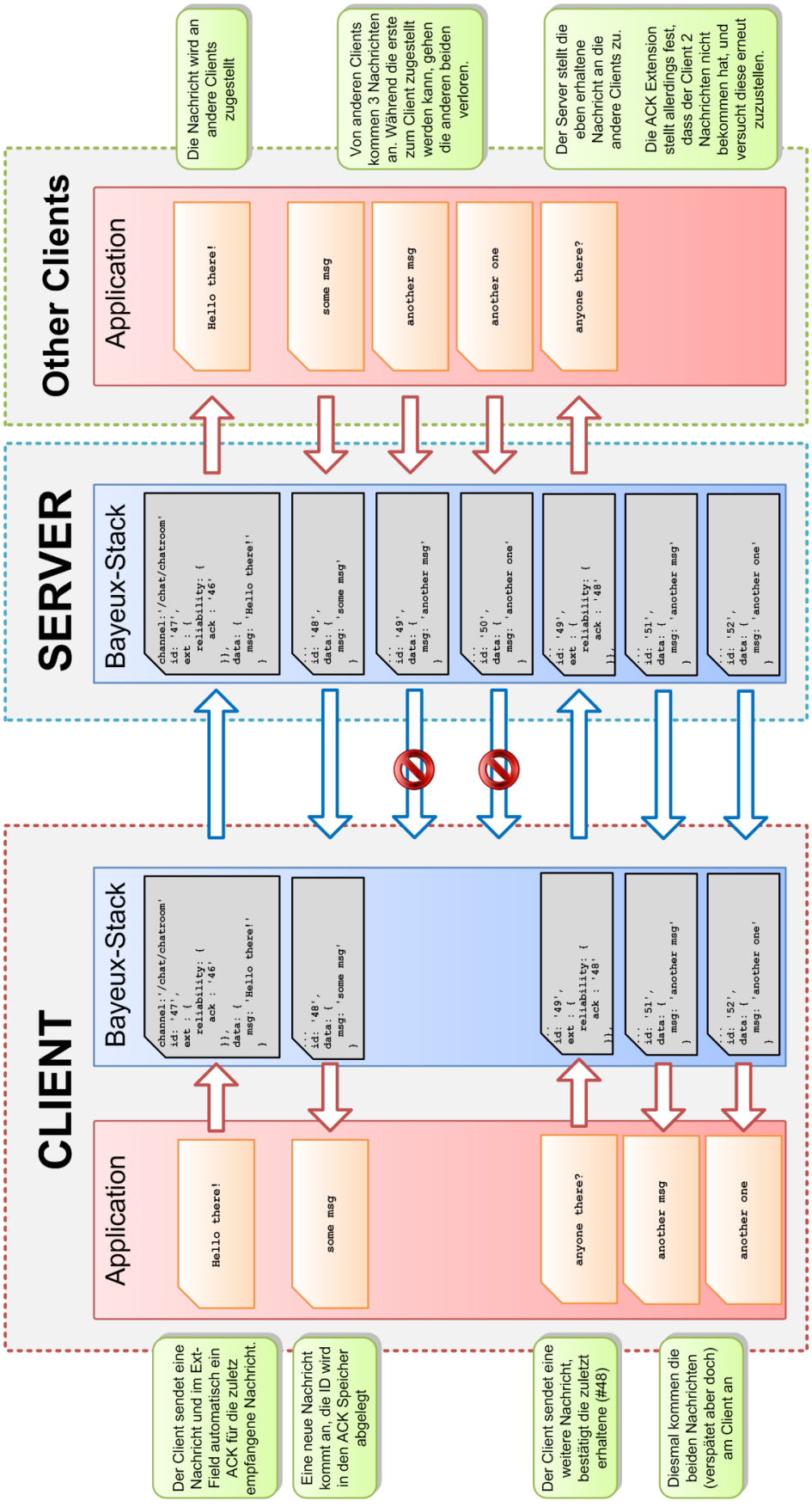


Abbildung 5.1: Recovery verlorengegangener Nachrichten beim Delayed Acknowledge

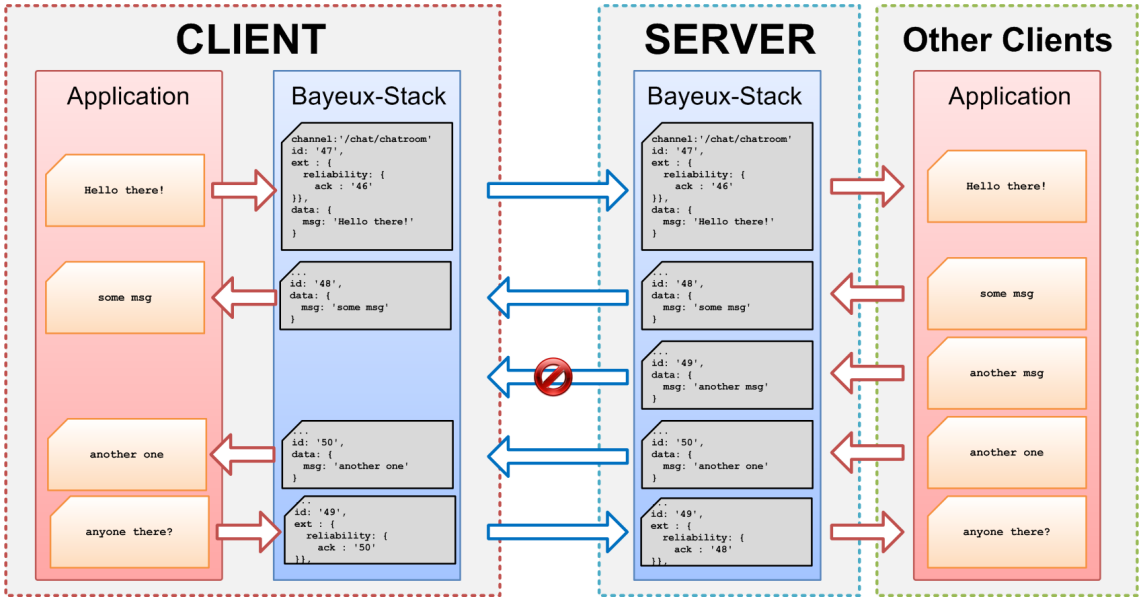


Abbildung 5.2: Unwiderruflich verlorene Nachricht beim Delayed Acknowledge

5.2.2 Per-Message Acknowledge

Eine weitere naheliegende Implementierungsform wäre ein *per-Message Acknowledge*, also ein direktes Bestätigen jeder Nachricht. Ziel dieses Verfahrens ist es, auch den vorhergehenden Fall eine verloren gegangenen Nachricht aus einer Serie von Nachrichten erkennen und beheben zu können. Jede empfangene Nachricht muss unmittelbar mit einer ACK-Nachricht (zum Beispiel an einen eigenen Meta-Channel `/meta/ack`) bestätigt werden. Erst nach Bestätigung der zuletzt gesendeten Nachricht kann die nächste Nachricht zum Client zugestellt werden.

Diese Methode benötigt aber wiederum mehr Ressourcen, nicht nur im Bezug auf Server-Ressourcen im Speicher, da ein entsprechend großer Nachrichtenpuffer vorhanden sein muss, sondern auch betreffend Requests, da jede Nachricht für sich in einer separaten ACK-Nachricht bestätigt wird. Um eine verloren gegangene Nachricht auch erneut zustellen zu können, muss der Server ein konfiguriertes Timeout besitzen, nach dessen Ablauf versucht wird, die erste Nachricht aus der Queue erneut zuzustellen. Abbildung 5.3 beschreibt diesen Vorgang genauer:

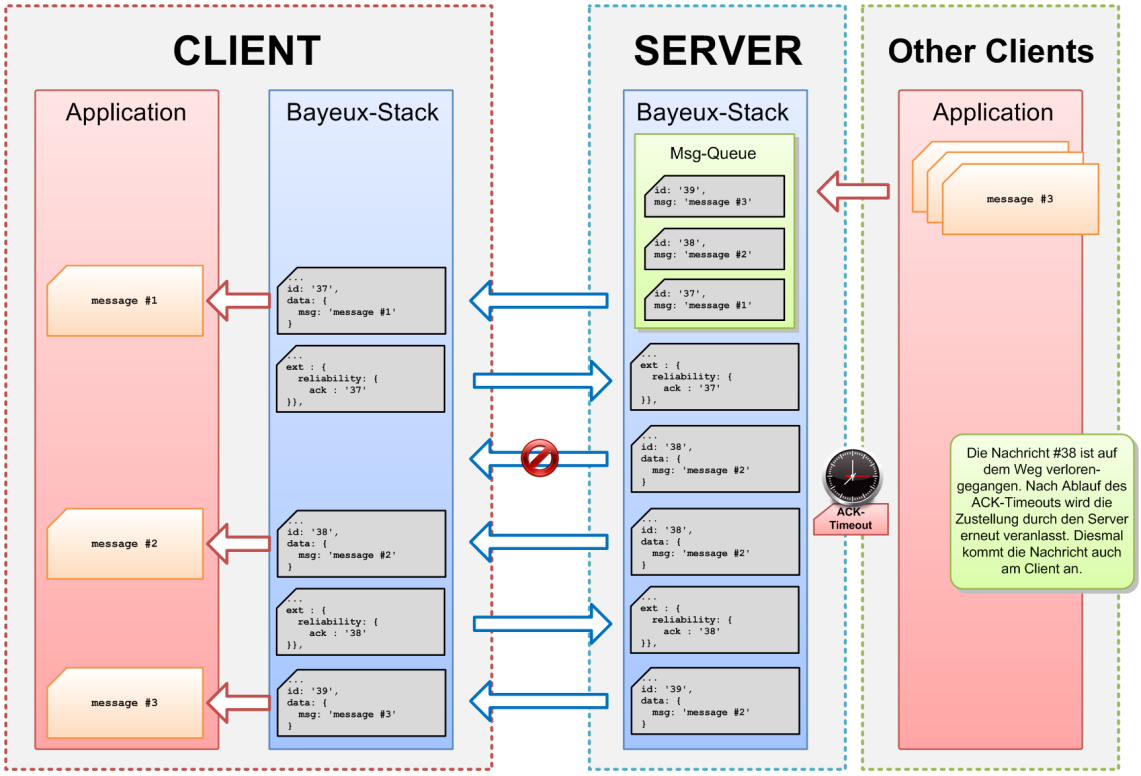


Abbildung 5.3: Zustellung von Nachrichten mit per-Message Acknowledge

1. Der Client subscribed auf einen Channel am Server
2. Der Server möchte Nachrichten #37 bis #40 verteilen, diese werden in einer Queue gehalten, Nachricht #37 wird zugestellt
3. Der Client bestätigt Nachricht #37
4. Direkt nach Erhalt der Bestätigung sendet der Server Nachricht #38
5. Die Nachricht geht verloren, der Client bestätigt nicht
6. Am Server läuft das Timeout ab, der Server sendet erneut Nachricht #38
7. Diesmal kommt die Nachricht erfolgreich an, der Client bestätigt die Nachricht und die nächste Nachricht wird vom Server zugestellt

Für eine Transport unabhängige Implementierung wären sowohl *Delayed ACK* als auch *Per-Message ACK* denkbar und auch durchführbar. Wenn man jedoch speziell auf die

aktuellen Implementierungen basierend auf dem HTTP Protokoll und Comet eingeht, so wird klar, dass die Architektur des Comet Protokolls eine Per-Message (oder besser per-Message-Collection) Acknowledge impliziert:

- Der Client baut die Comet-Verbindung zum Server auf, die Verbindung wird über `/meta/connect` hergestellt und vom Server auf unbestimmte Zeit (bis Daten anliegen oder ein timeout kommt) offen gehalten
- Der Server stellt über diese Verbindung 2 Nachrichten an den Client zu, beide Nachrichten bekommen von der cometd-java Implementierung **die selbe** Nachrichten-ID zugewiesen und werden gleichzeitig zum Client übertragen
- Der Client erhält die Daten und verarbeitet diese. Da der Comet-Kanal geschlossen wurde, wird dieser vom Client neu aufgebaut. Im Zuge dieser Verbindung zu `/meta/connect` wird auch sofort die ACK-Nachricht für die eben erhaltenen Nachrichten zum Server übertragen, sodass diese sofort bestätigt werden

Auch wenn durch diese Tatsache die Einstellung *Delayed-Acknowledge* unsinnig wäre, so muss die Implementierung nicht unbedingt auf dem HTTP-Kanal und Comet beruhen und somit wird die Grundlegende Idee aufrecht erhalten, jedoch nicht in der Referenz-Implementierung implementiert werden, da diese ohnehin auf Jetty/DojoX und somit CometD basiert.

5.2.3 Delivery Notification

Während die vorhergehenden Maßnahmen lediglich eine Erhöhung der Zuverlässigkeit zwischen Server und Client gebracht haben, so weiß der Absender einer Nachricht allerdings nicht, ob diese auch tatsächlich angekommen ist. Für manche Anwendungsfälle kann dies aber nötig sein (zum Beispiel eine wichtige Chat-Sitzung, eine Lock-Nachricht in einer Applikation, etc.).

Im Gegensatz zu den anderen beiden Möglichkeiten sollte auch eine Möglichkeit für die Zustellbenachrichtigung per-Channel existieren, womit der Channel selbst vorschreiben

kann, welches Maß an Zuverlässigkeit für Nachrichten dieses Bereiches maßgeblich sind.

Da es sich bei dieser Maßnahme nicht mehr um einen Client <-> Server, sondern viel mehr um einen Client <-> Server <-> Client Prozess handelt, steigt auch der Aufwand und der Ressourcen-Anspruch entsprechend an.

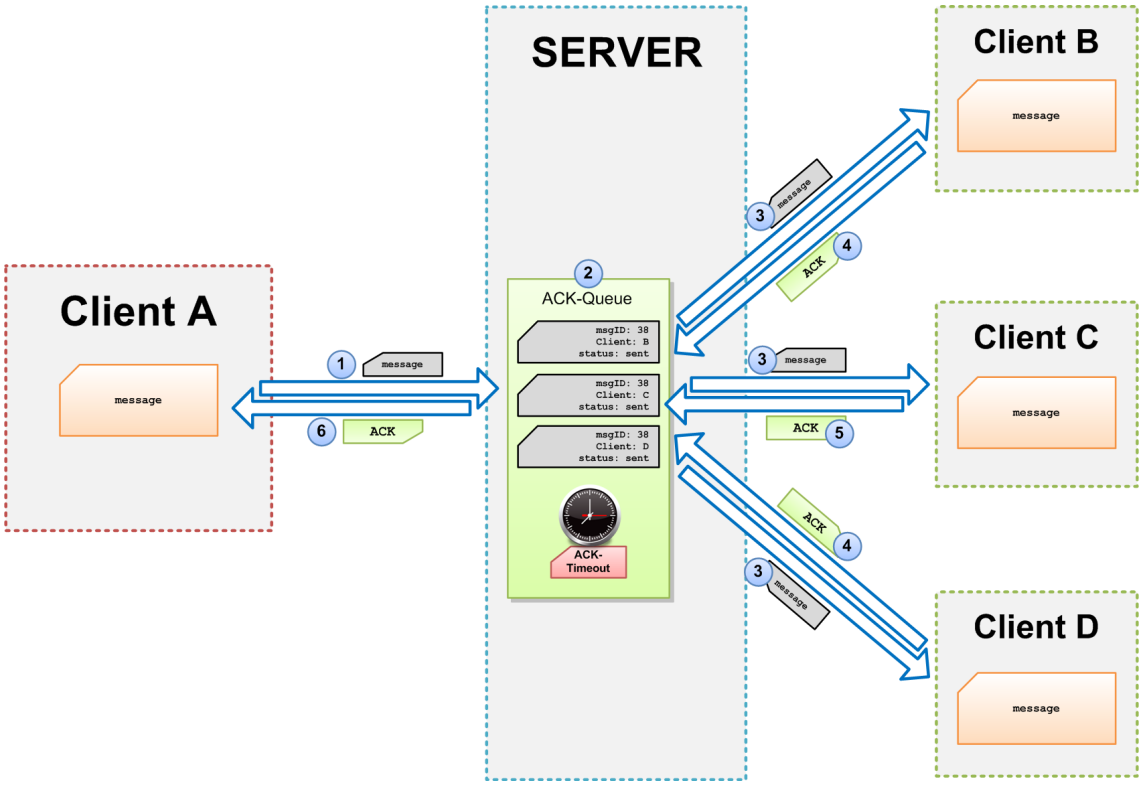


Abbildung 5.4: Benachrichtigung über erfolgreiche Nachrichtenzustellung

1. *Client A* sendet eine Nachricht an einen Channel
2. Der Server empfängt diese und die Extension merkt sich alle potentiellen Empfänger dieser Nachricht in einer ACK-Queue und erstellt einen Timer, welches eine NOT-ACK Nachricht an Client A sendet, wenn bis dahin noch nicht alle ausgehenden Nachrichten bestätigt wurden.
3. Die Nachricht wird an die einzelnen Clients im Channel verteilt. Beim Senden der Nachricht wird der Status in der ACK-Queue auf *sent* gesetzt, das bedeutet, dass auf ein ACK von diesem Client gewartet wird.

4. Client B und D antworten sofort mit einem ACK und werden in der Liste abgehakt.
5. Client C antwortet etwas verzögert, aber immer noch vor Ablauf des ACK Timers.
6. Alle Clients wurden in der Queue abgehakt, somit kann dem ursprünglichen Sender ein ACK gesendet werden. Der Timer und die ACK-Queue werden wieder entfernt.

Würde von einem Client kein ACK kommen und somit der Timer ablaufen, so würde die Queue gelöscht werden und dem ursprünglichen Sender der Nachricht ein `NAK` gesendet werden (verspätet eintreffende ACK-Nachrichten, welche keiner ACK-Queue zugeordnet werden können, werden von der Erweiterung ignoriert).

5.2.4 Integrität der Nachrichten

Ein weiteres Sicherheitsfeature ist die Sicherstellung und Gewährleistung der Integrität der Nachrichten. Client und Server sollten sicherstellen können, dass die übertragene Nachricht unversehrt angekommen ist, das heißt weder durch Übertragungsfehler, noch durch Einflüsse Dritter manipuliert wurde.

Dafür verwenden Netzwerkprotokolle einen so genannten MAC (Message Authentication Code), welcher im Normalfall durch einen Hash-Algorithmus über den Inhalt der Nachricht deren Identität darstellt. Da der Hash-Algorithmus aber ein Teil der Protokoll-Erweiterung und somit öffentlich zugänglich ist, kann ohne weiters ein Angreifer die gesendete Nachricht verändern, mit einem neuen MAC versehen und die Nachricht zustellen. Um dies zu verhindern, sollte in die Berechnung des MAC ein shared-secret einfließen, das nur Client und Server bekannt ist, pro Verbindung vergeben und evtl. auch mehrmals während einer Verbindung aktualisiert wird (vgl. Ferguson [2003], S. 97ff).

Die Berechnung der Prüfsumme über eine normale Zeichenkette oder einen Binären Datenblock wäre auch keine allzu große Herausforderung, da hier die Positionen der einzelnen Bestandteile nicht variabel sind. Nicht jedoch im Falle der im Bayeux Protokoll verwendeten JSON Nachrichten. Objekte, welche mittels JSON beschrieben werden, haben keine eindeutige Darstellung. Für das Senden und Empfangen muss eine JSON Nachricht codiert

und decodiert werden, dabei existiert aber keine Norm, wie die Nachricht zerlegt und anschließend wieder zusammengesetzt wird.

Ein Client könnte beispielsweise folgende Nachricht versenden (das Listing zeigt nur die Daten-Payload einer Bayeux-Nachricht, da auch nur für diese der MAC berechnet werden soll):

```
1 data : {
2   'event' : 'createAppointment',
3   'dateTime' : {
4       'time' : '12:00'
5       'date' : '2009-07-05',
6    },
7   'text' : 'Manuels Geburtstag'
8 }
```

Listing 5.1: JSON Nachricht, wie sie von einem Client gesendet wird

Der Server könnte die Nachricht wie folgt zusammensetzen:

```
1 data : {
2   'dateTime' : {
3       'date' : '2009-07-05',
4       'time' : '12:00'
5    },
6   'event' : 'createAppointment',
7   'text' : 'Manuels Geburtstag'
8 }
```

Listing 5.2: Die in Listing 5.1 gesendete Nachricht, zusammengestellt am Server

Während zum Beispiel also der Client die Attribute in der Reihenfolge der Zuweisung wiedergibt, sortiert der Server beim Zusammensetzen nach den Schlüsselwerten. Auch wenn die Nachricht nach außen hin gleich aussieht, stellt dies bei der Berechnung eines MAC ein großes Problem dar, da die geläufigen HASH Algorithmen (z.B. MD5, SHA1) nicht Positions unabhägig agieren und somit die Reihenfolge der Attribute / Werte in der String-Darstellung der Nachricht eine Rolle spielen.

Könnte die Nachricht unmittelbar vor dem Versenden bzw. nach dem Empfangen zur Berechnung des MAC herangezogen werden, könnte dieses Problem umgangen werden. Da

sich diese Arbeit aber mit der Erweiterung des Bayuex-Protokolls auseinander setzt und auf dieser Ebene ein solcher Eingriff nicht möglich ist, müssen die Nachrichten anderweitig normiert werden.

Eine Möglichkeit zur einheitlichen MAC Berechnung wäre folgende:

- Die Schlüssel einer Hierarchie-Ebene der JSON Nachricht werden aufsteigend sortiert
- Für jedes Schlüssel-Werte-Paar wird eine Zeichenkette erstellt, welche sich aus "<Schlüssel>:<Wert>" zusammensetzt
- Die resultierenden Strings werden direkt konkateniert und ein Shared-Secret beigemengt
- Der resultierende String sollte, unabhängig vom erfolgten JSON-Decoding, für diese Nachricht immer der selbe sein und kann als Grundlage für einen HASH Algorithmus herangezogen werden
- Ist der Wert eines Schlüssels selbst ein JSON-Objekt, so wird für diesen die Funktion rekursiv aufgerufen, das Ergebnis der Funktion (HASH-Wert in ASCII Form) als Wert herangezogen

Für die Nachricht aus den Listings 5.1 / 5.2 würde die Basis für eine MAC Berechnung (basierend auf dem Hash Algorithmus MD5) wie folgt aussehen:

```
dateTime:02d500f9f2e705ff4672b491632b2c7bevent:createAppointmenttext:Manuels
Geburtstag
```

Resultiert in folgendem MD5-Hash: `a7ffc9a9ab55b76ec9acfc951d85199c`

Wird vom Client oder Server ein nicht passender MAC entdeckt, so wird die Nachricht sofort mit einem NAK quittiert und erneut angefordert. Auch hierfür ist wiederum auf der Gegenseite ein Nachrichtenpuffer notwendig und somit eine direkte Verbindung des Integrity-Checks mit den Acknowledge Implementierungen von Vorteil.

5.3 Referenzimplementierung

5.3.1 Architektur

Da die in Kapitel 5.2 beschriebenen Techniken des Per-Message-Acknowledge und der Nachricht-Integritäts-Prüfung auf einer serverseitigen Zwischenspeicherung der Nachrichten beruhen, ist eine gemeinsame Implementierung in einer Extension die naheliegenste Alternative. Das heißt es wird auf Basis des Bayeux-Extension Stacks eine Erweiterung entwickelt, welche nach Bedürfnissen des Benutzers konfigurierbar ist und sämtliche der in Kapitel 5.2 erarbeiteten Features bieten kann.

Bereits beim initialen Handshake müssen die unterstützten Features (Version der Reliability Extension) abgeklärt werden. Dazu wird vom Client in der initialen Handshake-Nachricht ein Eintrag im ext-Feld mit folgendem Inhalt hinzugefügt (um Kollisionen mit der vor kurzem erschienenen ACK-Extension von Greg Wilkins zu vermeiden, welche ein ext-Field namens `ack` verwendet, werden in dieser Erweiterungen Informationen über ein Feld Namens `reliab` im EXT-Teil der Nachrichten ausgetauscht).

```
1 {
2   reliab : { version : '0.1' },
3   mac : { version : '0.1' },
4 }
```

In der Antwort zur Handshake Message fügt die Server-Extension ebenfalls ihre unterstütze Version ein, und somit können Client- und Server-Extension feststellen, ob und welche Features aktiviert werden. Weiters wird vom Server bei der Antwort auch das für die weitere Berechnung des MAC Codes zu verwendende shared-Secret beigefügt (z.B.: `mac : version : '0.1', secret : 'sh4r3dS3cr3t'`). Auch wenn hiermit das shared-Secret selbst über einen (im Falle von normalen HTTP Transaktionen) unsicheren Kanal übertragen wird, so wird für den weiteren Verlauf die MAC Berechnung vor Man-in-the-Middle (MitM) Attacken geschützt (vgl. Edney [2005], S37f). Bei Übertragungen über HTTP besteht jedoch ohnehin die Möglichkeit, den gesamen Daten-Traffic verschlüsselt über HTTPS abzuwickeln, wodurch MitM Attacken komplett ausgeschlossen werden kön-

nen. Für hochsichere Implementierungen, welche nicht auf HTTPS Aufbauen können, sollte der shared-secret austausch mittels sicherer kryptografischer Methoden wie Diffie-Hellman (vgl. Ferguson [2003], Kapitel 12) oder dergleichen durchgeführt werden. Da diese einen zu tiefgreifenden Ausflug in die Kryptografie bedeuten würde, wird auf eine nähere Ausführung in dieser Arbeit verzichtet. Eine Implementierung kann jedoch im Bedarfsfall anhand der angegebenen Quelle durchgeführt werden.

Nach dem Herstellen der Verbindung wird auf dem Server für das Client-Objekt eine Instanz der serverseitigen Client-Extension angelegt und zugewiesen. Diese hat im weiteren Verlauf die Aufgabe, die Nachrichten zu cachen, den MAC zu berechnen und im Falle einer fehlgeschlagenen Zustellung diese erneut durchzuführen.

5.3.2 Server

5.3.2.1 Allgemeine Erweiterung

Ähnlich wie bei der Authentifizierungs- und Autorisierungserweiterung wird auch hier wiederum zuerst eine allgemeine Erweiterungsklasse implementiert, welche überprüft, ob der Client überhaupt das Reliability-Protokoll unterstützt und gegebenenfalls die, für die jeweilige Bayeux-Client Instanz am Server zuständige, spezifische Erweiterung installiert. Für die Delivery Notifications hat die allgemeine Erweiterung außerdem noch die Aufgabe, die ACKs an die jeweiligen Client-Extensions weiterzuleiten (siehe Listing 5.3).

```
1  public Message sendMeta(Client from, Message message)
2  {
3      // get the original message, sent by the user
4      Message rcv = message.getAssociated();
5
6      // get the ext-field
7      Map<String, Object> ext = rcv.getExt(true);
8      HashMap<String, Object> reply = new HashMap();
9
10     reply.put("version", this.protocolVersion);
11     if (message.getChannel().equals(Bayeux.META_HANDSHAKE) && Boolean.TRUE.equals(
           message.get(Bayeux.SUCCESSFUL_FIELD)))
12     {
13         // if client supports reliability, install client-extension
14         if(ext.get("reliab") != null)
```

```
15          {
16              from.addExtension(new ReliabilityClientExtension(from, this, _listener)
                    );
17          }
18
19          ext.put("reliab", reply);
20          message.put(Bayeux.EXT_FIELD, ext);
21      }
22      return message;
23  }
24
25  // pass through the received acks to delivery notification listeners
26  public void ackReceived(Client from, Message message)
27  {
28      List<DeliveryNotifier> snapshot = new ArrayList(deliveryNotifiers);
29      for(DeliveryNotifier notify : snapshot)
30          notify.ackReceived(from, message);
31  }
```

Listing 5.3: Allgemeine serverseitige Reliability Extension

5.3.2.2 Clientspezifische Erweiterung

Die grundlegende Aufgabe der Client-spezifischen Erweiterung ist die Verwaltung des Nachrichtenzwischenspeichers. Sowohl für die Integritätsprüfung, als auch für die Zuverlässigkeit müssen die Nachrichten eine bestimmte Zeit (bis die jeweilige Bestätigung einlangt, oder das Timeout abläuft) zwischengespeichert werden. Da die Bayeux-Implementierung das gleichzeitige Senden von mehreren Nachrichten innerhalb eines Requests ermöglicht (Message-Batches), muss hier eine geeignete Lösung für die Speicherung der Nachrichten gefunden werden, da alle Nachrichten in einem Batch auch ein und die selbe Nachrichten-ID zugewiesen bekommen.

Hierfür wurde eine kleine Wrapper-Klasse konzipiert, welche mehrere Nachrichten mit der selben ID zusammenfasst und auch den zugehörigen ACK-Timer verwaltet. Nach dem Hinzufügen der ersten Nachricht wird auch der ACK-Timer mit einem Standardwert von 10 Sekunden initialisiert. Die Erweiterung versucht dem Client jede Nachricht bis zu 3 mal erneut zuzustellen, bevor die gesamte betroffene Nachrichten-Sammlung verworfen wird, sofern nicht eine Empfangsbestätigung eintrifft (siehe Listing 5.4).

```
1  class MessageCollection extends TimerTask
2  {
3      Vector<Message> messages = new Vector();
4      String _collectionID = null;
5      ClientImpl _client = null;
6      final ReliabilityClientExtension _rce;
7
8       /* Initializes the collection */
9      MessageCollection(ReliabilityClientExtension rce,String baseID);
10      /* adds a message to the collection, starts the ACK timer */
11     void addMessage(Message m);
12      /* returns the list of messages in this collection */
13     Vector<Message> getMessages();
14      /* stops the timer (if active) **/
15     void stopTimer();
16      /* resends a given message */
17     void resendMessage(Message m)
18      /* requeues a given message (not sent) */
19     void requeueMessage(Message m)
20      /* sends a dummy message so the client-queue gets sent */
21     void fireQueue()
22 }
```

Listing 5.4: Hilfs-Klasse zur Verwaltung von Nachrichten-Batches

Die Überprüfung ob und welche Reliability-Optionen für eine Bestimmte Nachricht zum Einsatz kommen, wird in der Funktion `public void send(Client, Message)` durchgeführt. Wenn erforderlich, wird für die Nutzdaten der Nachricht (im Feld `data` gespeicherte Daten) ein MAC berechnet und der Nachricht hinzugefügt. Anschließend wird für die Nachrichten-ID eine vorhandene Sammlung im Nachrichtenspeicher gesucht oder eine neue erstellt, wenn noch keine vorhanden ist. Für sämtliche "Meta"-Nachrichten wird keine Reliability angeboten, da sonst sehr leicht eine Endlosschleife und somit auch eine Überlastung entstehen könnte.

Eingehende ACK-Nachrichten werden im `ext`-Feld einer `/meta/connect`-Nachricht mitgesendet und in der Funktion `public void rcvMeta(Client,Message)` ausgewertet. Wird eine ACK-ID empfangen, so wird die zugehörige Nachrichten-Sammlung abgehakt und entfernt. Wird hingegen ein Fehlermeldung bezüglich eines nicht passenden MACs empfangen, so wird die Nachricht anhand des original-MACs aus der Sammlung gesucht und erneut zugestellt.

Wird eine Zustellbestätigung gewünscht, so wird bereits bei der eingehenden Nachricht eine Hilfs-Klasse namens `DeliveryNotification` instantiiert (siehe Listing 5.5). Diese Klasse registriert sich selbst bei der allgemeinen Reliability Extension für eingehende ACK-Nachrichten, liest die Anzahl an Abonnenten des Ziel-Channels aus und startet einen Ablauf-Timer. Jedes eingehende ACK wird von den Client-Extensions an die allgemeine Extension weitergereicht, wodurch auch jeweilige DeliveryNotification-Listener darüber informiert werden und den ACK-Counter reduzieren können. Wurden alle Nachrichten zugestellt oder aber wurde das Timeout erreicht, wird dem Sender der Original-Nachricht eine Antwort zugestellt, welche eine der folgenden Daten-Payloads enthält:

- `421:<msgID>:Delivery Successful`
- `422:<msgID>:Delivery Failed:<unackedCount>`

```
1  class DeliveryNotifier extends TimerTask
2  {
3     ReliabilityExtension _re;
4     ReliabilityClientExtension _rce;
5     ClientImpl _from;
6     MessageImpl _msg;
7     TimerTask _timer;
8     int ackCount = 0;
9     boolean canceled = false;
10
11     /* Constructor for initialization */
12    DeliveryNotifier(Client from, Message m, ReliabilityExtension re,
          ReliabilityClientExtension rce)
13     { ... }
14
15     /* Handles a received Client-ACK, if ackCount reaches 0, the delivery was
           successful */
16    public void ackReceived(Client from, Message msg)
17     { ... }
18
19     /* Timer-Method for ACK-timeout */
20    public void run()
21     { ... }
22 }
```

Listing 5.5: Hilfs-Klasse für Zustellbenachrichtigungen

Zur Kommunikation mit der Applikation wurde, wie auch für die Auth-Extension, ein Interface erstellt, welches von der Applikation implementiert werden muss (vgl. Listing 5.6).

```
1  public interface ReliabilityListener
2  {
3     public static final int RELIAB_NONE = 0;
4     public static final int RELIAB_DELAYED = 1;
5     public static final int RELIAB_PER_MESSAGE = 2;
6     public static final int RELIAB_LIKE_CLIENT = 4;
7     public static final int RELIAB_INTEGRITY = 8;
8
9     public int getReliabilityLevel(String channel);
10 }
```

Listing 5.6: ReliabilityListener Interface, serverseitge Referenzimplementierung

5.3.3 Client

Die Client Implementierung ist hier die kleinere Aufgabe, der Client muss lediglich erhaltene Nachrichten bestätigen und gegebenenfalls eine MAC-Code Berechnung und Überprüfung durchführen. Wie auch bei der Auth-Extension wurde eine eigene dojo-Klasse `bayeux.cometd._reliability` erstellt, welche die beiden Methoden `_in` und `_out` implementiert und im Extension-Stack registriert. Zusätzlich wurde noch ein Event-Handler `function onDelivery (errno, errmsg, msgid, unackedCount)` definiert, welcher bei Erhalt einer Zustell-Benachrichtigung aufgerufen wird. Damit die Zustell-Benachrichtigung aktiviert wird, muss die Applikation die Methode `setDeliveryInfo(channel, state)` aufrufen und den gewünschten Kanal und `true` übergeben. Beim Senden von Nachrichten wird überprüft, ob ein ACK zu senden ist und ob eine deliveryNotification angefordert werden soll (siehe Listing 5.7).

```
1  this._out = function(msg)
2  {
3      // show the server, that we can handle reliability
4      if (msg.channel == "/meta/handshake")
5      {
6          if (!msg.ext) msg.ext = {};
```

```
7           msg.ext.reliab = { version : '0.1' };
8       }
9       else
10      {
11          // check if there is an ack to send
12          if(ackToSend)
13          {
14              if(!msg.ext) msg.ext = {}
15              if(!msg.ext.reliab) msg.ext.reliab = {};
16
17              msg.ext.reliab.ack = lastID;
18              ackToSend = false;
19          }
20
21          // activate deliveryInfo for the specified channel if set
22          if(typeof deliveryInfo[msg.channel] != "undefined" && deliveryInfo[msg.
                channel] === true)
23          {
24              if(!msg.ext) msg.ext = {}
25              if(!msg.ext.reliab) msg.ext.reliab = {};
26
27              msg.ext.reliab.deliveryInfo = true;
28          }
29      }
30      return msg;
31  }
```

Listing 5.7: Setzen von Reliability-Infos am Client

In der Eingangsfunktion wird primär auf eingehende Reliability-Informationen gewartet, diese ausgewertet und darauf entsprechend reagiert. Eine der empfangenen Informationen kann ein vom Server generierter MAC für die Nachricht sein, welcher vom Client verifiziert und bestätigt werden muss. Alternativ kann der Server auch einfach nur ein ACK für die Nachricht fordern. Eingehende Zustell-Benachrichtigungen werden ebenfalls über die `_in` Methode abgewickelt und an die Applikation weitergegeben (vgl. Listing 5.8).

```
1   this._in = function(msg)
2   {
3       /* Check if we have receive a delivery notification */
4       if(msg.channel == "/meta/ack")
5       {
6           var data = msg.data.split(":");
7           this.onDelivery(data[0], data[2], data[1], (data.length == 4 ? data[3] : 0)
                );
8           return msg;
9       }
10
```

```
11      // Check if there is reliability info included in the message
12      if(msg.ext && msg.ext.reliab)
13      {
14          var errMsg = null;
15          lastID = msg.id;
16          ackToSend = true;
17
18          if(msg.ext.reliab.mac)
19          {
20              var myHash = this._getCheckSum(msg);
21
22              if(myHash != msg.ext.reliab.mac)
23              {
24                  // in order to dispose the message, we would have to remove the
                        channel attribute
25                  // or at least let it point to some nonesense...
26                  msg.channel = "nonesense";
27                  // we have to inform the server about this!!
28                  errMsg = "macmissmatch";
29              }
30          }
31
32          // force immediate ack/nak (ACK-ID gets inserted later in the _out func)
33          if(errMsg)
34              dojox.cometd.publish('/meta/ack', {}, { ext : { reliab : { error :
                    errMsg, msgid : msg.id, msgmac : msg.ext.reliab.mac } } });
35          else
36              dojox.cometd.publish('/meta/ack', {}, { ext : { reliab : { } } });
37      }
38
39      return msg;
40  }
```

Listing 5.8: Verarbeiten von Reliability-Infos am Client

In der praktischen Umsetzung bekommen die Enduser-Applikationen nichts von einer vorhandenen Reliability-Extension mit, da diese ohne jeglichen Kontakt, rein im Hintergrund arbeitet und eine fehlerfreie Zustellung der Nachrichten gewährleistet. Lediglich bei aktivierter Zustell-Benachrichtigung wird auch die Applikation über erfolgte oder fehlgeschlagene Übermittlungen von Nachrichten informiert.

Kapitel 6

Implementierung einer Teststellung

6.1 Zielsetzung / Architektur

Die Zielsetzung dieser Teststellung ist es, eine reine *Proof-Of-Concept* Lösung zu erstellen, also eine einfache unvollständige Applikation zu schaffen, welche lediglich die in den Kapitelnd 4 und 5 erforschten und entwickelten Algorithmen und deren Funktion unter Beweis stellen soll. Es ist *nicht* das Ziel, eine vollständige, sinnvolle, verwendbare Applikation zu entwerfen, da alleine dafür der Aufwand einer eigenen Diplomarbeit gerecht wäre.

Als Grundlage der Teststellung wird das Chat-Demo aus dem Comet/Bayeux Repository herangezogen. Dieses wird um die neu hinzugekommenen Features des Protokolls erweitert:

- Da sich in der bisherigen Demo Applikation jeder Benutzer mit beliebigen Nicknamen einloggen konnte, wird die Anmeldung zu einer echten Authentifizierung umgebaut. Serverseitig wird eine Benutzerdatenbank simuliert.
- Es wird einen privaten "moderator-channel" geben, wo nur bestimmte User subscriben/publishen dürfen.

- Der Client wird für diesen "moderator-channel" die Zustell-Info aktivieren, um über (nicht) zugestellte Nachrichten benachrichtigt zu werden.
- Es wird auch einen, nur mit Kennwort (Passphrase) betretbaren Channel geben.
- Alle Nachrichten werden per Integritäts-Check vor MitM Angriffen und Übertragungsfehlern geschützt.

6.2 Durchführung

6.2.1 Server

Um eine möglichst realistische Benutzerverwaltung simulieren zu können, werden 2 Hilfs-Klassen implementiert. Für die allgemeine Verwaltung wird ein `UserManager` (siehe Listing 6.1) und für den Benutzer selbst die Klasse `User` (siehe Listing 6.2) erstellt.

```
1  class UserManager
2  {
3     private static HashMap<String,User> users;
4
5      /* gets a user by his id */
6     static User getUser(String uid)
7      { ... }
8
9      /* gets a user by his name */
10    static User getUserByName(String name)
11     { ... }
12
13     /* a simple password verification function */
14    boolean checkPassword(String pwd,String secret,String hashType)
15     { ... }
16
17     /* static initializer to fill the 'user-db' */
18    static
19    {
20        users = new HashMap();
21        users.put("1001", new User("1001", "manii", "test", true));
22        users.put("1002", new User("1002", "hugo",  "foo",  false));
23        users.put("1003", new User("1003", "asdf",  "bar",  false));
24        users.put("1004", new User("1004", "some",  "one",  true));
25    }
26 }
```

Listing 6.1: Benutzerverwaltungs-Klasse zur Simulation einer Benutzerdatenbank

```
1  class User
2  {
3     private String _id;
4     private String _name;
5     private String _password;
6     private boolean _moderator;
7
8      /* constructor for initialization */
9     User ( String id, String name, String password, boolean moderator)
10     { ... }
11
12     /* some getter methods for the properties */
13    String getID()
14     { ... }
15    String getName()
16     { ... }
17     String getPassword()
18     { ... }
19    boolean isModerator()
20     { ... }
21 }
```

Listing 6.2: Benutzer-Klasse zur Simulation eines Chat-Benutzers

Damit die Erweiterungen überhaupt aktiv werden, müssen diese im Konstruktor des `ChatService` zuerst dem allgemeinen Bayeux-Protokoll Extension-Stack hinzugefügt werden:

```
1  [ ... ]
2
3  public ChatService(Bayeux bayeux)
4  {
5      super(bayeux, "chat");
6      subscribe("/chat/**", "trackMembers");
7      subscribe("/service/privatechat", "privateChat");
8      bayeux.addExtension(new AuthenticationAuthorizationExtension(this, bayeux));
9      bayeux.addExtension(new ReliabilityExtension(this, bayeux));
10 }
11
12 [ ... ]
```

Listing 6.3: Hinzufügen der Erweiterungen zum Chat-Service

Um die Schnittstellen zur Applikation herstellen zu können, müssen die verschiedenen Event-Listener implementiert werden (`AuthorizationListener` und `AuthenticationListener`, siehe Kapitel 4.6.2, Seite 51 bzw. `ReliabilityListener`, siehe Kapitel 5.3.2,

Seite 73). Die in den Event-Listenern definierten Funktionen werden dabei, soweit nötig, mittels diverser Zugriffe auf die simulierte Benutzerdatenbank ausprogrammiert. Im Falle der Test-Applikation implementiert die Haupt-Klasse `ChatService.java` alle 3 Interfaces, und wird in den jeweiligen Konstruktoren als Schnittstelle (vgl. Listing 6.3) den Erweiterungen übergeben.

```
1  [ ... ]
2
3  /* ~~~~~~~~ AUTHORIZATION LISTENER INTERFACE ~~~~~~~~ */
4
5  public int subscriptionAllowed(Object token, String channel)
6  {
7      User user = UserManager.getUser((String)token);
8      if(user == null)
9          return DENIED;
10
11     if("/chat/protected".equals(channel))
12         return PASSPHRASE_REQUIRED;
13
14     if("/chat/moderators".equals(channel))
15         return user.isModerator() ? ALLOWED : DENIED;
16
17     return ALLOWED;
18 }
19
20 public int publishAllowed(Object token,String channel,Message msg)
21 {
22     return subscriptionAllowed(token, channel);
23 }
24
25 public boolean checkPassPhrase(Object token, String channel, String passPhrase)
26 {
27     return "testPhrase".equals(passPhrase);
28 }
29
30 [ ... ]
```

Listing 6.4: Implementierung des AuthorizationListener Interfaces

Die restlichen Features des Chats, wie verteilen der Member-Listen oder der Private-Chat wurden nicht abgeändert.

6.2.2 Client

Am Client musste der Chat etwas mehr umstrukturiert werden. Da hier viele Aktivitäten asynchron ablaufen, musste zum Beispiel der gesamte Initialisierungs-Prozess in mehrere Funktionen aufgeteilt werden. Damit auch hier die Erweiterungen aktiv werden, müssen diese mittels `dojo.require` geladen weren.

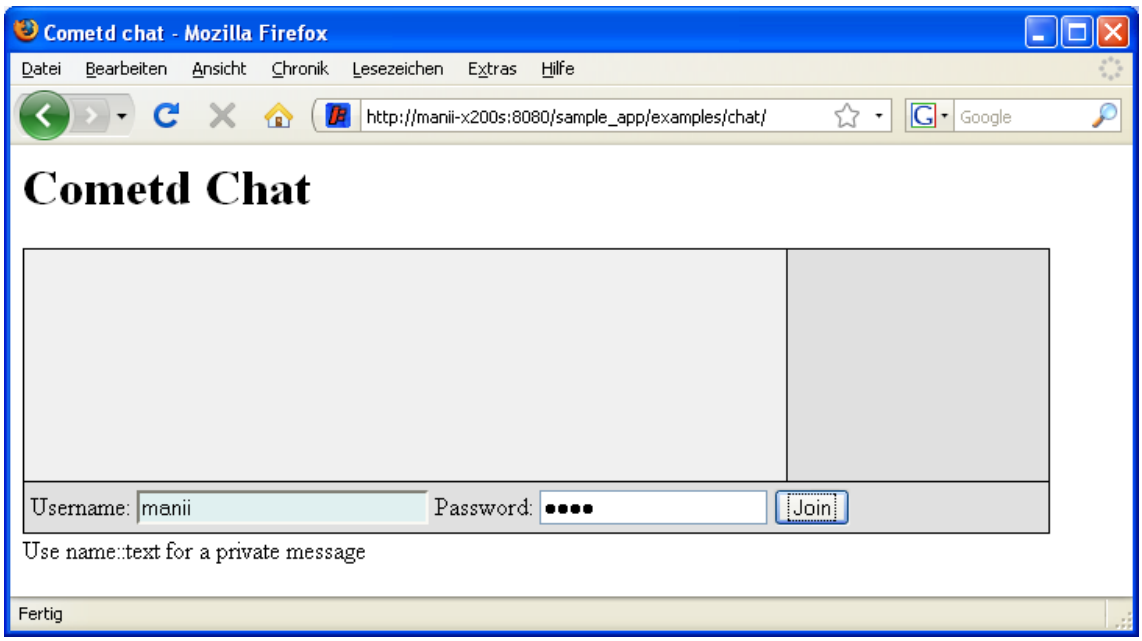


Abbildung 6.1: Login-Bildschirm der Chat-Testapplikation

Dem Chat-Formular wurde zu aller erst ein Passwort-Feld hinzugefügt. Nach Klicken auf den "Join"-Schaltfläche wird die Initialisierung des Comet-Kanals und die Authentifizierung begonnen. In der Funktion `join` werden auch die diversen Event-Handler und Error-Handler registriert (siehe Listing 6.5).

```
1  [ ... ]
2
3  join : function(name, pwd)
4  {
5      /* check for a valid user name */
6      if (name == null || name.length == 0) {
7          alert('Please enter a username!');
8          return;
9      }
10
```

```
11      /* create the cometd url which we connect to */
12      var loc = (new String(document.location).replace(/http:\/\/[^\/]*/, '').replace
            (/\/examples\/.*$/, '')) + "/cometd";
13
14      /* set username/password in the auth extension */
15      dojox.cometd._auth.setCredentials(name, pwd);
16
17      /* install auth-error handler */
18      dojox.cometd._auth.onError = function(errno, msg) {
19          alert("auth failure: \n("+errno+") "+msg);
20      };
21
22      /* activate delivery notification for moderators channel */
23      dojox.cometd._reliability.setDeliveryInfo("/chat/moderators",true);
24
25      /* install delivery notification event handler */
26      dojox.cometd._reliability.onDelivery = function(errno,errmsg,msgid,unacked) {
27          if(errno == "421")
28              alert("<"+errno+"> -- Nachricht ID["+msgid+"] wurde erfolgreich an alle
                    Clients zugestellt! ("+errmsg+")");
29          else
30              alert("<"+errno+"> -- Nachricht ID["+msgid+"] konnte NICHT erfolgreich
                    an alle Clients zugestellt werden! ("+errmsg+"; "+unacked+" clients
                    nicht erreicht!)");
31      };
32
33      /* install auth-connect handler, called when auth was successful */
34      dojox.cometd._auth.onConnect = function () {
35          room._connected = true;
36          room._username = name;
37
38          dojo.addClass("join", "hidden");
39          dojo.removeClass("joined", "hidden");
40          dojo.byId("phrase").focus();
41
42          room.joinRoom("lobby");
43      };
44      dojox.cometd.init(loc);
45  }
46
47  [ ... ]
```

Listing 6.5: Initialisierung des Chat-Clients

Da nun auch mehrere Räume zur Auswahl stehen, wurde die Client-Klasse um eine Variable `_activeRoom` erweitert. Zusätzlich wurde die Subscription in eine eigene Funktion ausgelagert, welche auch eine Passphrase für Räume verarbeiten kann (vgl. Listing 6.6).

```
1  [ ... ]
2
3  joinRoom : function(roomName, passPhrase)
4  {
5      // set a passphrase if given
6      if(typeof passPhrase != undefined)
7          dojox.cometd._auth.setPassPhrase('/chat/'+roomName, passPhrase);
8      // subscribe and join a room, on success the "joinedRoom" func should be called
9      dojox.cometd.subscribe("/chat/"+roomName, room, "_chat").addCallback(dojo.hitch
           (room, 'joinedRoom', roomName));
10 }
11
12 [ ... ]
```

Listing 6.6: Funktion zum Betreten von Chat-Räumen am Client

Wurde ein Raum erfolgreich betreten, wird über die Funktion `joinedRoom` eine Welcome-Nachricht verschickt und auch eine eventuelle Subscription für einen anderen Raum gelöscht. Kann der Raum aufgrund fehlender Berechtigung nicht betreten werden, so wird der installierte Auth-Error Handler aufgerufen. Die Applikation gibt in diesem Fall einfach eine simple Fehlermeldung per `alert` aus (siehe Abbildung 6.2).

Um auch die Zustellbenachrichtigung zu testen, wurde diese für den Raum `/chat/moderators` aktiviert. Hierfür ist lediglich ein Funktionsaufruf zu `dojox.cometd._reliability.setDeliveryInfo('/chat/moderators/', true)` nötig (siehe Listing 6.5). Danach wird für jede gesendete Nachricht in diesem Kanal vom Server eine Benachrichtigung gesendet, ob diese Nachricht alle im Raum befindlichen Benutzer erreicht hat oder nicht. Für Testzwecke wird die Zustellbenachrichtigung direkt per `alert` ausgegeben und nicht weiter verarbeitet (siehe Abbildung 6.3). Die restlichen Features (Message-ACK und Berechnung bzw. Check des MAC) der Reliability-Erweiterung laufen ohne weiters zutun im Hintergrund ab.

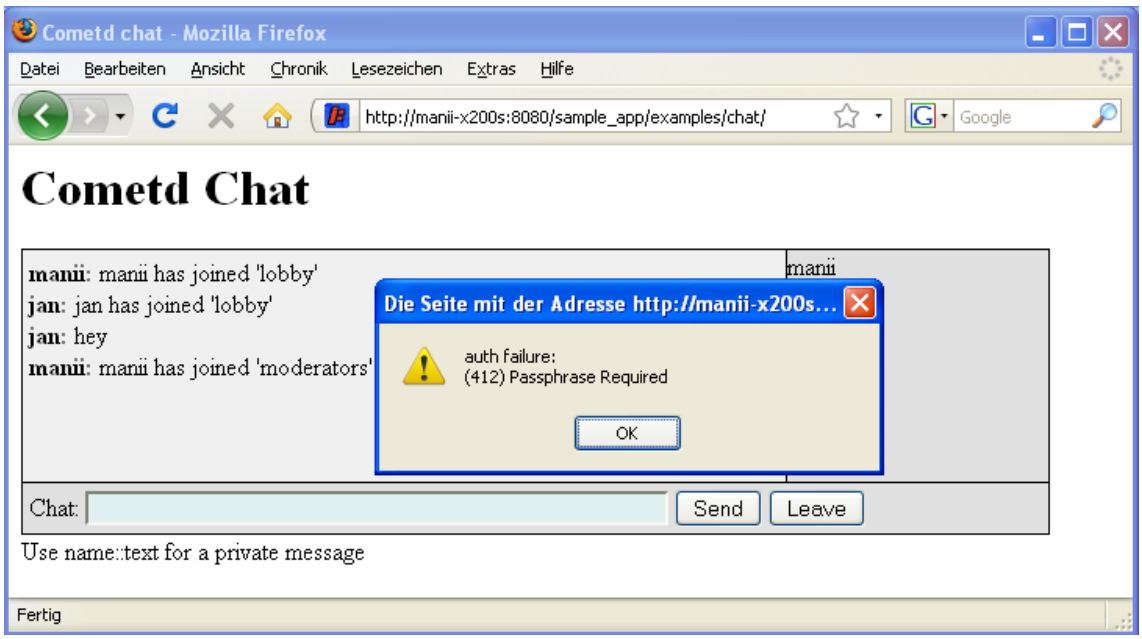


Abbildung 6.2: Fehler beim Betreten des geschützten Raumes “protected”

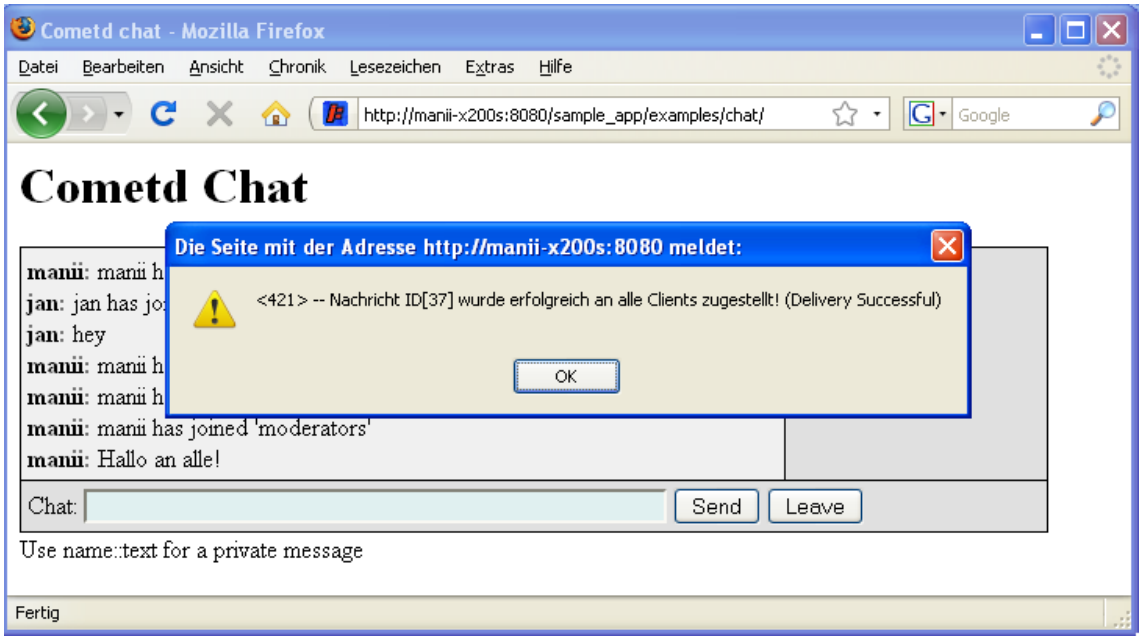


Abbildung 6.3: Benachrichtigung über erfolreiche Zustellung der letzten Nachricht

6.3 Resultate

Obwohl die erste Implementierung ein paar Schwierigkeiten bereitete, konnten schließlich durchaus brauchbare Ergebnisse erzielt werden:

- Die Authentifizierung und Autorisierung funktioniert vollständig, wie erwartet
- Die Autorisierung mittels Passphrase funktioniert einwandfrei
- Die server- und clientseitige Berechnung des MAC liefert die selben Ergebnisse, wodurch auch die Integritätsprüfung erfolgreich arbeitet
- Die Reliability-Extension stellt Nachrichten bis zu 3 mal zu. Wird ein Browser in den Offline-Modus versetzt, oder verlässt ein Teilnehmer den Chat ohne sich abzumelden, so liefert die Zustell-Benachrichtigung einen entsprechenden Fehler

Während das Einbinden der Reliability-Erweiterung für die Applikation völlig unbemerkt von statten geht, müssen für die Authentifizierungs und Autorisierungs-Erweiterung einige umstrukturierungen vorgenommen werden. So kann nach Herstellen der Verbindung nicht auch automatisch davon ausgegangen werden, dass sobald eine Nachricht vom Server kommt, die Verbindung auch erfolgreich hergestellt wurde. Hier sind asynchrone Handler, sowohl für Fehler- als auch Erfolgsfall zu erstellen (vgl. Listing 6.6, `dojox.cometd._auth.onError` und `onConnect`). Auch das Betreten von Räumen und Senden von Nachrichten kann nicht mehr als selbstverständlich beachtet werden, sondern es muss auf etwaiige Autorisierungs Probleme mit Passwort-Abfragen oder Fehlermeldungen reagiert und der Benutzer darüber in Kenntnis gesetzt werden.

Kurz zusammengefasst funktionieren alle implementierten Features wie erwartet. Lediglich kleinere Feinheiten sollten vor der praktischen Anwendung noch näher überprüft und implementiert werden (wie zum Beispiel eine saubere Synchronisation der Zugriffe auf gemeinsam genutzte Ressourcen am Server).

Kapitel 7

Abschließendes Fazit

7.1 Erkenntnisse der Arbeit

Der prinzipielle Ansatz von CometD und dem Bayeux-Protokoll geht in eine sehr gute Richtung, jedoch konnte in dieser Arbeit erfolgreich gezeigt werden, dass es durchaus noch einige verbesserungswürdige Punkte gibt.

Die erstellten Erweiterungen bringen weitere wichtige Features eines Netzwerkprotokolls und ermöglichen somit auch noch sichereres Arbeiten mit dieser sehr jungen Technologie.

Die Grundidee des Erweiterungs-Stacks, Erweiterungen wie eine Filterkette zu durchlaufen, ist zwar sehr gut, jedoch sollte den Erweiterungen selbst noch etwas mehr Mitspracherecht in den wichtigen Phasen des Bayeux-Protokolls gewährt werden. Trotz der Tatsache, dass diese Arbeit mit diversen Workarounds auch zum Ziel kam, wäre es zum Beispiel wesentlich einfacher gewesen, wenn eine Erweiterung einfach `NULL` zurückliefern könnte und damit alle folgenden Tätigkeiten automatisch abgebrochen werden.

Viele Features, Funktionsweisen und Programmierrichtlinien des Protokolls konnten auch erst nach intensivem Studium der diversen Source-Codes und nicht nur durch Lesen der Dokumentation herausgefunden werden.

7.2 Ausblicke für weiterführende Arbeiten

Im Moment wird sehr viel in die Richtung von CometD/Bayeux entwickelt. Die in Kapitel 2.3.3 erwähnte Servlet API 3.0 wird die Technologie, welche bereits in Jetty 6 per Workaround implementiert wurde, standardisieren und es für alle Servlet basierten Web-Server ermöglichen, Dienste basierend auf CometD und Bayeux anzubieten.

Eine weitere Alternative bietet der, aktuell in Entwicklung befindliche HTML Standard 5.0, welcher viele interessante Features für Rich Internet Applikationen bringt (siehe W3C [2009]). Darunter sind auch vor allem für die CometD und Bayeux Entwickler interessante Features wie *asynchrone server-sent Events* und *Cross Document Messaging*, welche ähnlich wie Comet eine asynchrone Benachrichtigung durch den Web-Server ermöglichen oder im Falle des *Cross Document Messaging* sogar direkte Kommunikation zwischen Browser-Fenstern erlauben.

Fakt ist, dass die in dieser Arbeit behandelten Technologien gerade erst am Anfang ihrer Entwicklung stehen, aber sehr großes Potential bieten.

Anhang A

Abbildungsverzeichnis

Anhang B

Listingverzeichnis

Anhang C

Literaturverzeichnis

[Ballard 2008] Ballard, Phil: Sams Teach Yourself Ajax, Javascript, and Php All in One. Sams, 2008. – ISBN 9780672329654

[Berners-Lee 1994] Berners-Lee, T.: Uniform Ressource Locators (RL). December 1994. – http://www.ietf.org/rfc/rfc1738.txt

[Crane 2008] Crane, Dave: Comet and Reverse Ajax: the Next-Generation Ajax 2.0. Berkeley : APress, 2008. – ISBN 9781590599983

[Edney 2005] Edney, Arbaugh: Real 802.11 Security, Wifi Protected Access and 802.11i. Addison Wesley, 2005. – ISBN 0321136209

[Ferguson 2003] Ferguson, Schneier: Practical Cryptography. Wiley, 2003. – ISBN 0471223573

[Fielding 1999] Fielding, R.: Hyper Text Transfer Protocol - HTTP/1.1. June 1999. – http://www.w3.org/Protocols/rfc2616/rfc2616.html

[Garrett 2008] Garrett, Jesse J.: AJAX: A New Approach To Web Applications. December 2008. – http://www.adaptivepath.com/ideas/essays/archives/000385.php

[Ippolito 2005] Ippolito, Bob: Remote JSON - JSONP. December 2005. – http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/

[Janisch 2008] Janisch, G.: Analyse von Rich Internet Application Frameworks am Beispiel einer Thesaurusverwaltung. (2008). – http://staff.fh-hagenberg.at/kurschl/pubs/advisedDT/Janisch.Gerhard.2008.pdf

[O'Reilly 2005] O'Reilly, Tim: Web 2.0: Compact Definition? October 2005. – http://radar.oreilly.com/archives/2005/10/web-20-compact-definition.html

[O'Reilly 2007] O'Reilly, Tim: What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. In: Communications & Strategies 65 (2007), S. 17–37

[Powell 2008] Powell, Thomas: Ajax: the Complete Reference. McGraw-Hill Osborne Media, 2008. – ISBN 9780071492164

[Russel 2008a] Russel, Alex: Bayeux Protocol – Bayeux 1.0draft1. December 2008. – http://svn.cometd.com/trunk/bayeux/bayeux.html

[Russel 2008b] Russel, Alex: Comet: Low Latency Data for the Browser. December 2008. – http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/

[Stats 2008] Stats, Internet W.: Internet World Statistics, Overall Population compared to Internet Users. December 2008. – http://www.internetworldstats.com/stats.htm

[W3C 2009] W3C: HTML 5 differences from HTML 4. February 2009. – http://www.w3.org/TR/html5-diff/

[Wang 2008] Wang, Weigang: Powermeeting on common ground: web based synchronous groupware with rich user experience. In: WebScience '08: Proceedings of the hypertext 2008 workshop on Collaboration and collective intelligence. New York, NY, USA : ACM, 2008, S. 35–39. – ISBN 978-1-60558-171-2

[Wilkins 2007] Wilkins, Greg: JSR 315 – Servlet 3.0, Asynchronous Servlet Proposal. July 2007. – http://dist.codehaus.org/jetty/misc/AsyncServlet3.0-draft0.html

[Wilkins 2008a] Wilkins, Greg: Colliding Comets: Battle of the Bayeux Part 1. February 2008. – http://cometdaily.com/2008/02/07/colliding-comets-battle-of-the-bayeux-part-1/

[Wilkins 2008b] Wilkins, Greg: Jetty Continuations. June 2008. – http://docs.codehaus.org/display/JETTY/Continuations

[Wilkins 2009] Wilkins, Greg: Acknowledged Messages Extension. February 2009. – http://blogs.webtide.com/gregw/entry/cometd_acknowledged_message_extension