WebSockets: Spezifikation / Implementierung

Benjamin Ullrich
Betreuer: Philipp Fehre
Hauptseminar: Innovative Internettechnologien und Mobilkommunikation WS2011/12
Lehrstuhl Netzarchitekturen und Netzdienste
Fakultät für Informatik, Technische Universität München
Email: bu@elunic.com

KURZFASSUNG

Webseiten gehen heutzutage oft über die reine Darstellung statischer Hypertexte hinaus. Entwickler nutzen die Vorteile browserbasierter Anwendungen und schaffen interaktive Web-Applikationen mit dynamischen Inhalten, die vom Anwender unabhängig von Ort und Endgerät genutzt werden können. Das HTTP-Protokoll genügt den Anforderungen an die Kommunikation zwischen Client und Server dabei nur bedingt. Durch eine Vielzahl an notwendigen Serverabfragen erzeugt es einen großen Overhead beim Datenaustausch und verursacht unnötige zeitliche Verzögerungen.

Mit der Einführung von HTML5 soll auch das neue WebSocket-Protokoll in modernen Webbrowsern zum Einsatz kommen, das bidirektionale Verbindungen zwischen Webbrowser und Webserver ermöglicht und den notwendigen Netzwerktraffic minimieren soll. Im ersten Teil dieser Arbeit werden das JavaScript-Interface, technische Aspekte der Protokoll-Spezifikation und der Einsatzzweck von WebSockets untersucht. Im zweiten Teil wird die konkrete Implementierung von WebSockets aufgezeigt und die Fragestellung beantwortet, wie diese neue Technologie trotz eingeschränkter Browserkompatibilität schon heute eingesetzt werden kann.

Schlüsselworte

WebSocket Protokoll, HTML5, HTTP Header-Overhead, bidirektionale Webbrowser-Kommunikation, Comet

1. MOTIVATION

Der Einsatzzweck von Webseiten geht längst über den Abruf statischer Informationen hinaus. Technische Fortschritte der Webbrowser, wie leistungsstärkere JavaScript-Engines, eingehaltene browserübergreifende Konventionen und Bibliotheken wie jQuery ermöglichen die Entwicklung dynamischer Web-Applikationen. Beispiele wie Facebook oder Google-Docs zeigen, dass moderne Webapplikationen in vielen Bereichen auch als Alternative zu klassischen Desktop-Anwendungen eingesetzt werden können. Vorteile sind etwa:

- Keine clientseitige Installation der Anwendung notwendig
- Ortsunabhängige Verfügbarkeit der Services
- Client-Kompatibilität (verschiedene OS, Smartphones, Tablets...)
- Lizenzmodelle, die schwer oder gar nicht umgangen werden können

Nachteile sind dagegen:

- Notwendigkeit einer Internetverbindung
- Webbrowser-Kompatibilitätsprobleme durch Abweichungen von Standards (z.B. Internet Explorer 6)
- eingeschränkte Performance mit clientseitiger JavaScript-Programmierung
- eingeschränkter Zugriff auf Rechner-Ressourcen mit clientseitiger JavaScript-Programmierung im Gegensatz zu Desktop-Programmiersprachen (z.B. Grafikkarte, Netzwerksockets)

Nach wie vor stellen vor allem die Einschränkungen der JavaScript-Programmierung einen Nachteil gegenüber Desktopapplikationen dar. Mit der Einführung von HTML5 wurden daher neue Standards für Webbrowser definiert, die die Möglichkeiten der clientseitigen JavaScript-Programmierung erweitern.

Eine dieser Neuerungen sind “WebSockets”, deren Spezifikation in dieser Arbeit untersucht werden soll. Während Desktop-Entwickler Verbindungen zwischen Endgeräten aufbauen und über diese bidirektional kommunizieren können, beschränkt sich das HTTP-Protokoll auf die serverseitige Beantwortung clientseitiger Anfragen. Dies hat zur Folge, dass der Webbrowser für jede Aktualisierung von Daten eine neue Anfrage an den Server stellen muss. Anstatt je nach Bedarf Daten über eine offene Verbindung kommunizieren zu können, muss außerdem für jede Anfrage eine neue Verbindung aufgebaut werden. Dadurch entsteht ein Daten-Overhead, unnötige Rechenleistung wird beansprucht und die Aktualisierung der Daten wird unnötig verzögert.

Ein Beispiel hierfür ist die Realisierung eines Webseiten-Chats. Da die Client-Anwendung nicht weiß, wann neue Chat-Nachrichten verfügbar sind, muss sie regelmäßig neue Anfragen an den Server senden. Diese enthalten dann gegebenenfalls Informationen über neue Nachrichten. Ist das Abfrageintervall lang, erscheinen Nachrichten erst mit einiger Verzögerung beim Chatpartner. Ist das Abfrageintervall kurz, werden u.U. viele unnötige Anfragen gesendet, obwohl

doi: 10.2313/NET-2012-04-1_8

in der Zwischenzeit keine neuen Nachrichten gesendet wurden.

Das HTTP-Protokoll genügt den Anforderungen moderner Webapplikationen an eine effiziente Kommunikation zwischen Client und Server daher nur bedingt. Mit HTML 5 wurde das WebSocket Protokoll vorgestellt. Dieses ermöglicht Entwicklern von Web-Anwendungen das Öffnen von TCP-Verbindungen für einen bidirektionalen Datenaustausch mit einem Webserver. Bisherige Technologien, die zu diesem Zweck mehrere HTTP-Verbindungen öffnen mussten, sollen dadurch überflüssig gemacht werden.

Der erste Teil dieser Arbeit beschreibt das WebSocket Protokoll und zeigt, wie dieses eine sichere Verbindung zwischen Webbrowser und Server ermöglicht. In Abschnitt 3 wird darauf aufbauend eine vom World Wide Web Consortium (W3C) definierte Client-API[4] für WebSockets vorgestellt[1]. Anhand einer Beispielanwendung wird demonstriert, welche Auswirkungen der Einsatz von WebSockets als Alternative zu HTTP-Anfragen auf die Netzlast haben kann. Zum Abschluss der Arbeit werden Einschränkungen in der Kompatibilität mit gängigen Webbrowsern beschrieben und gezeigt, wie die Vorteile von WebSockets bereits heute genutzt werden können.

2. WEBSOCKET PROTOKOLL

Das WebSocket Protokoll wurde seit seiner Bekanntgabe kontinuierlich weiterentwickelt und im Dezember 2011 von der IETF als offizieller Internetstandard eingeführt [7]. Dieser Abschnitt beschreibt, welche Funktionsweisen die Spezifikation für Webbrowser und Server vorgibt, um eine sichere Verbindung zu gewährleisten.

2.1 Verbindungsaufbau

Bei der Initialisierung neuer WebSockets baut der Webbrowser eine TCP-Verbindung zu der angegebenen Server-Adresse auf. Zu Beginn wird ein Handshake durchgeführt. Mit diesem stellt der Browser sicher, dass der Server das WebSocket-Protokoll versteht und es werden Parameter der Verbindung spezifiziert.

Eine Herausforderung bei der Einführung von WebSockets war die einfache Integrierbarkeit in die etablierte Infrastruktur des Internets. Das WebSocket-Protokoll wurde daher als Upgrade des HTTP-Protokolls konzipiert und kann so ohne aufwändige Neukonfigurationen auch in bestehenden Web-Servern über Port 80 genutzt werden. Der Opening Handshake einer WebSocket-Verbindung muss daher ein valider HTTP-Request sein. Die IETF weist aber darauf hin, dass in zukünftigen Versionen des Protokolls auch ein einfacherer Handshake über einen dedizierten Port denkbar ist. [3]

Header 1 zeigt den WebSocket Request-Header. Wie bei HTTP ist die erste Zeile nach dem Request-Line Format aufgebaut, bestimmt also Methode, angefragte Ressource (Request-URI), Protokoll und Protokollversion. Die Methode ist dabei auf den Wert „GET“ festgelegt. Die zweite Zeile gibt den adressierten Host und optional einen Port an, falls die Verbindung nicht über Port 80 laufen soll. Über den Wert „websocket“ des Upgrade-Feldes in Kombination mit dem Wert „Upgrade“ des Connection-Feldes wird der Host aufgefordert, auf das WebSocket-Protokoll zu wechseln. [3]

[1] s. http://dev.w3.org/html5/websockets

Header 1 Opening Handshake: Request-Header [3]

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Origin: http://example.com
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
Sec-WebSocket-Protocol: chat, superchat
```

Header 2 Opening Handshake: Response-Header [3]

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Daneben definiert das Protokoll die Verwendung einiger zusätzlicher Felder, die eine sichere Verbindung gewährleisten sollen oder mit denen Parameter der Verbindung spezifiziert werden können [3]:

Origin

Das Feld muss vom Webbrowser mitgeschickt werden, um vor unerlaubten Cross-Domain Anfragen in Webbrowsern zu schützen. Der Server kann durch die Auswertung des Felds also selbst entscheiden, von welchen Adressen er Anfragen zulassen möchte.

Sec-WebSocket-Key

Das Feld enthält einen zufälligen 16-Byte-Wert der base64-kodiert wurde.

Sec-WebSocket-Version

Die Version des Protokolls; nach aktueller Spezifikation muss das Feld den Wert „13“ besitzen.

Sec-WebSocket-Protocol (optional)

Das Feld zeigt an, welche Anwendungsschicht-Unterprotokolle vom Client unterstützt werden.

Sec-WebSocket-Extension (optional)

Protokoll-Erweiterungen, die vom Client unterstützt werden. Diese Angabe ermöglicht zukünftig die Integration von Erweiterungen des Protokolls, wie eine Frame-Komprimierung oder Multiplexing mehrerer WebSocket-Verbindungen[2], ohne sie für alle Anwendungsfälle zwingend vorauszusetzen.

Optional können analog zu HTTP weitere Felder zwischen Client und Server kommuniziert werden, wie beispielsweise Cookies.

[2] s. auch http://tools.ietf.org/html/draft-tamplin-hybi-google-mux-01

doi: 10.2313/NET-2012-04-1_8

Der korrekte Response-Header des Servers (vgl. Header 2) zeigt dem Client an, dass der Server das WebSocket-Protokoll versteht. Die erste Zeile ist nach dem HTTP Status-Line Format aufgebaut, enthält also Protokoll, Version und Status-Code. Ist der Status-Code ungleich „101“, so kann der Webbrowser die Antwort analog zu HTTP behandeln, um beispielsweise eine Autorisierung bei Code „401“ durchzuführen. [3]

Der Server muss den HTTP-Upgrade Prozess durch Angabe derselben Felder-Werte für „Connection“ und „Upgrade“ wie im Request-Header vervollständigen. Weitere durch das Protokoll bestimmte Felder sind außerdem [3]:

Sec-WebSocket-Accept

Mit dem Feld muss ein umgeformter Wert des im Request-Header empfangenen „Sec-WebSocket-Key“ übergeben werden. Der empfangene Wert muss hierzu mit der GUID „258EAFA5-E914-47DA-95CA-C5AB0DC85B11“ verknüpft werden. Der Server muss dann einen SHA-1 Hash des verketteten Wertes generieren, diesen base64-kodieren und das Ergebnis als Wert des Header-Feldes „Sec-WebSocket-Accept“ zurückgeben.

Damit kann der Webbrowser sicherstellen, dass der Server die Anfrage wirklich gelesen und verstanden hat. Sollte der Browser einen falschen Wert für Sec-WebSocket-Accept empfangen, muss er die Antwort laut Protokoll als serverseitige Ablehnung der Verbindung werten und darf daher keine zusätzlichen Frames senden.

Sec-WebSocket-Protocol (optional)

Wurden vom Client ein oder mehrere Unterprotokolle empfangen, so kann der Server maximal eines davon für die Verbindung auswählen. Durch Zurücksenden des Wertes im „Sec-WebSocket-Protocol“ wird dem Client die Wahl des Unterprotokolls bestätigt.

Sec-WebSocket-Extension (optional)

Wurden vom Client unterstützte Extensions empfangen, so kann der Server durch Zurücksenden einzelner oder aller Extensions deren Verwendung für die WebSocket-Verbindung bestätigen.

Optional können wiederum weitere Felder zwischen Server und Client kommuniziert werden, wie beispielsweise „Set-Cookie“ um das Cookie zu überschreiben.

2.2 Datenübertragung

Nach einem erfolgreichen Opening Handshake wird clientseitig das WebSocket-Event „onopen“ angestoßen und es können bidirektional Nachrichten ausgetauscht werden. Die Nachrichten können jeweils aus einem oder mehreren Frames bestehen. So wird einerseits ermöglicht, Nachrichten ohne vorheriges Puffern zu senden. Andererseits kann ein Teilen der Nachrichten in mehrere kleinere Frames vorteilhaft für ein zukünftiges Multiplexing mehrerer Socket-Verbindungen sein.[3] Um die Kommunikation vor „Cache Poisoning“ zu schützen, müssen alle vom Client an den Server gesendeten Frames maskiert [5] werden.

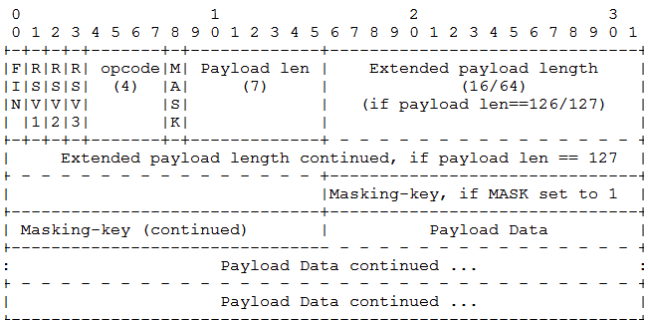

Abbildung 1: Framing-Modell

Eine Grundphilosophie des WebSocket-Protokolls ist es, den Overhead beim Datenaustausch möglichst gering zu halten. Daran orientiert sich auch das Framing der einzelnen Pakete. Abbildung 1 zeigt den Aufbau eines WebSocket-Frames. Das „FIN“-Bit gibt an, ob es sich um das letzte Frame der Nachricht handelt. Die folgenden drei Bits sind für Protokoll-Erweiterungen reserviert. Bits vier bis sieben beinhalten den OP-Code, der angibt, wie der Nachrichteninhalt interpretiert werden muss. Beispielsweise sind Textframes durch den OP-Code „0x1“ gekennzeichnet. Das achte Bit gibt an, ob das Frame maskiert ist. Die nächsten Bits geben die Länge das Nachrichteninhalts an. Je nach Umfang des Nachrichtinhalts werden für diese Angabe 7, 23 oder 71 Bits verwendet. Falls das Maskierungsbit gesetzt wurde, müssen die nächsten 32 Bit den Maskierungskey angeben. Alle weiteren Bits werden für den Nachrichteninhalt verwendet. [3]

Je nach Nachrichtenlänge und Maskierung werden pro Frame also 2 - 14 Bytes an Headern benötigt.

2.3 Verbindungsabbau

Um die Verbindung zu schließen, wird ein Closing-Handshake durchgeführt (vgl. Abbildung 2). Die Seite, die die Verbindung schließen möchte, muss dazu ein Frame mit dem Befehlscode „0x8“ senden. Optional kann im Body der Nachricht der Grund für das Schließen enthalten sein. Die ersten zwei Bytes des Bodys müssen dann einen Statuscode für den Grund enthalten. [3]

Empfängt eine Seite einen Close-Frame, ohne zuvor selbst einen Close-Frame gesendet zu haben, muss sie einen Close-Frame als Antwort senden. Nachdem eine Seite sowohl einen Close-Frame empfangen als auch gesendet hat, muss sie die TCP-Verbindung trennen. Durch dieses Vorgehen stellt es auch kein Problem dar, falls beide Seiten zur selben Zeit die Verbindung trennen möchten: Da beide Seiten parallel einen Close-Frame senden und kurz darauf den der jeweils anderen Seite empfangen, können sie die Verbindung direkt trennen. [3]

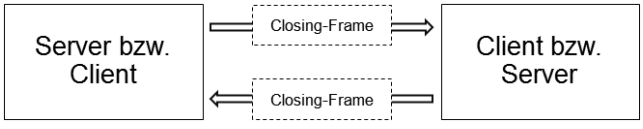

Abbildung 2: Closing Handshake

doi: 10.2313/NET-2012-04-1_8

3. WEBSOCKET API

3.1 Das WebSocket Interface

Quelltext 1 zeigt das Interface eines WebSockets nach [4], das dem Anwendungsentwickler clientseitig zur Verfügung steht.

3.1.1 Constructor

Der Constructor eines WebSockets besitzt den Parameter „url" und den optionalen Parameter „protocols". Für die Adressierung des WebSocket-Servers wurden zwei URI-Schemen analog zu HTTP eingeführt [4]:

ws-URI: **ws://host[:port][/path][?query]**
und wss-URI (secure): **wss://host[:port][/path][?query]**

Mit dem zweiten Parameter „protocols" können ein oder mehrere bevorzugte Protokolle für die Kommunikation angegeben werden.

Quelltext 1 WebSocket Interface (vereinfacht) [4]

```
[Constructor(url, optional protocols)]

interface WebSocket : EventTarget {

        readonly attribute url;

        // ready state
        const CONNECTING = 0;
        const OPEN = 1;
        const CLOSING = 2;
        const CLOSED = 3;

        readonly attribute readyState;
        readonly attribute bufferedAmount;

        // networking
        attribute Function onopen;
        attribute Function onerror;
        attribute Function onclose;

        readonly attribute extensions;
        readonly attribute protocol;

        void close(optional code, optional reason);

        // messaging
        attribute Function onmessage;

        attribute binaryType;

        void send(data);
};
```

3.1.2 Methoden

Nach der Initialisierung stehen dem JavaScript-Entwickler die beiden Methoden „send" und „close" zur Verfügung [4]:

send(data:mixed) Durch den „send"-Aufruf werden die Daten des Parameters „data" an den Server gesendet. Mögliche Formate des „data"-Parameters sind:

String Text-Nachricht (Textframes)

ArrayBuffer Daten des ArrayBuffers (Binärframes)

Blob Rohdaten (Binärframes)

close(optional code, optional reason) Mit dem Aufruf der Methode „close" wird die TCP-Verbindung des WebSockets geschlossen.

Quelltext 2 MessageEvent Interface [1]

```
interface MessageEvent : Event {

        readonly attribute data;
        readonly attribute origin;
        readonly attribute lastEventId;
        readonly attribute source;

        void initMessageEvent(...);
        void initMessageEventNS(...);
};
```

3.1.3 Events

Um auf Ereignisse des WebSockets zu reagieren, kann der Entwickler Callbacks für vier Events definieren [4]:

onopen(event:Event) Der Callback wird aufgerufen, sobald die TCP-Verbindung geöffnet und ein Handshake mit dem Server erfolgreich durchgeführt wurde.

onmessage(event:MessageEvent) Der Callback wird jedes Mal dann aufgerufen, wenn eine neue Nachricht vom WebSocket-Server eingetroffen ist. Das „data"-Attribut des „event"-Parameters (vgl. Quelltext 2) enthält den Inhalt der Nachricht, der Wert steht abhängig vom übermittelten Datentyp im Format String, ArrayBuffer oder Blob zur Verfügung.

onerror() Der Callback wird aufgerufen, wenn ein ungültiges Frame empfangen und die WebSocket-Verbindung daher geschlossen wurde.

onclose(event:CloseEvent) Der Callback wird aufgerufen, wenn die Verbindung getrennt wurde. Der Parameter „event" enthält das Attribut „code", über das der Statuscode für das Schließen der Verbindung abgefragt werden kann.

doi: 10.2313/NET-2012-04-1_8

3.1.4 Attribute

Jeder WebSocket besitzt außerdem folgende readonly-Attribute [4]:

url Die im Constructor übergebene Server-URL.

readyState Der Statuscode der Verbindung als Zahl zwischen 0 und 3. Quelltext 1 zeigt die entsprechenden Konstanten der Statuscodes.

bufferedAmount Anzahl der aktuell zu sendenden Bytes, die noch nicht übertragen wurden.

extensions Alle Erweiterungen des WebSocket-Protokolls, die im WebSocket verwendet werden.

protocol Das eingesetzte Unterprotokoll, falls vorhanden.

3.2 Anwendungsbeispiel

Mit Hilfe der WebSocket-API können Entwickler von Web-Applikationen effiziente bidirektionale Verbindungen mit WebSocket-Servern aufbauen. Der in Abschnitt 1 angesprochene Overhead durch den Einsatz von HTTP-basierten Technologien entfällt. Dies lässt sich am Beispiel des Browserchats veranschaulichen.

Quelltext 3 zeigt den JavaScript-Code eines einfachen Browserchats. Um den Code übersichtlich zu halten wurde die Bibliothek jQuery[3] verwendet. Der Code enthält eine Funktion „addMessage“, um Nachrichten in einem Container des DOMs anzuhängen, einen Listener, der bei Klicks auf einen „send“-Button eine Nachricht versendet und eine Poll-Funktion, die jede Sekunde einen Ajax-Request an einen Server sendet, um - falls vorhanden - neue Nachrichten zu erhalten.

Quelltext 4 zeigt den selben Chat-Client unter Verwendung von WebSockets. Auf den ersten Blick fällt auf, dass der Code etwas kompakter ist als die Polling-Variante. Die Funktion „wsChat.ondata“ kann per Server-Push neue Nachrichten unmittelbar empfangen, sobald sie im Server eintreffen. Dadurch kommen im Gegensatz zu Quelltext 3 keine leeren Nachrichten mehr an und mehrere Nachrichten müssen nicht mehr als Array gebündelt versendet werden. Prüfung und Iteration über „data“ entfallen daher.

Ressourcenverwendung und Performance

Der eigentliche Mehrwert zeigt sich aber bei der Ressourcenverwendung. In Variante 1 muss jede Sekunde ein neuer Request gesendet werden, unabhängig davon, ob tatsächlich neue Daten vorliegen. Jeder Request erzeugt dabei einen HTTP Header-Overhead. Abbildung 3 zeigt den Anfrage- und Antwortheader einer typischen Ajax-Anfrage, die in diesem Beispiel zusammengenommen eine Länge von 871 Bytes aufweisen [8]. Da die Headerlänge von Anzahl und Umfang der übertragenen Metainformationen abhängt, kann sie je nach Anwendungs-Szenario auch deutlich länger oder kürzer ausfallen. Abbildung 4 zeigt, welchen Netzwerktraffic dieser Headeroverhead in Abhängigkeit von der Zahl der gleichzeitig auf einer Webseite aktiven Benutzer unter folgenden Annahmen generiert:

[3] s. http://jquery.com

Quelltext 3 Browserchat via Ajax-Polling

```
var chatCon = $('#chatContainer'),
    chatInput = $('#chatInput');

function addMessage(message) {
    $('<div class="msg">' + message + '</div>')
        .appendTo(chatCon);
}

function pollForMessages() {
    $.get(
        'http://server.de/chat',
        {},
        function(data) {
            if($.isArray(data))
                return;
            for(var n = 0; n < data.length; n++)
                addMessage(data[n]);
        }
    );
}

setInterval(pollForMessages, 1000);

$('#send').on('click', function() {
    addMessage(chatInput.val());
    $.post(
        'http://server.de/chat',
        {input: chatInput.val()}
    );
    chatInput.val('');
});
```

Quelltext 4 Browserchat via WebSockets

```
var chatCon = $('#chatContainer'),
    chatInput = $('#chatInput'),
    wsChat = new WebSocket('ws://server.de:80/chat');

function addMessage(message) {
    $('<div class="msg">' + message + '</div>')
        .appendTo(chatCon);
}

wsChat.ondata = function(event) {
    addMessage(event.data);
};

$('#send').on('click', function() {
    addMessage(chatInput.val());
    wsChat.send(chatInput.val());
    chatInput.val('');
});
```


doi: 10.2313/NET-2012-04-1_8

- Polling jede Sekunde
- Neue Nachricht alle 2 Sekunden
- Nachrichtenlänge 50 Bytes (entspricht 4 Byte Frame-Header)
- Ein Frame pro Nachricht

Fazit: Der Netzwerktraffic kann bei diesen Header-Längen durch den Einsatz von WebSockets von 83MB/s auf ca. 195 KB/s verringert werden. Die Einsparung von 99,77% der benötigten Netzwerkkapazitäten resultiert aus dem geringen Header-Overhead des WebSocket-Protokolls von nur 4 Byte. Durch das Polling entsteht außerdem eine Verzögerung, bis die Nachrichten beim Chatpartner angezeigt werden, der selbst bei Vernachlässigung der Übertragungszeiten bis zu eine Sekunde dauern kann. Die Verzögerung könnte zwar verkürzt werden, indem das Abfrageintervall verringert wird, allerdings würde dies den Ressourcenaufwand noch weiter erhöhen.

4. WEBSOCKETS „IN THE WILD“

Das WebSocket-Protokoll ist zum aktuellen Zeitpunkt noch sehr jung und wurde von der IETF im vergangenen Jahr mehrfach überarbeitet. Erst am 11.12.2011 wurde das Protokoll als IETF-Standard veröffentlicht. Dementsprechend gering fällt zum aktuellen Zeitpunkt die Unterstützung der Webbrowser aus. Tabelle 1 zeigt die geplante oder realisierte Implementierung entscheidender Protokoll-Entwicklungsstufen in gängigen Webbrowsern.

Protokoll	hixie-76 (Mai 2010)	hybi-10 (Jul. 2011)	RFC 6455 (Dez. 2011)
Chrome	6	14	16
Safari	5.0.1	-	-
Firefox	4.0 (deaktiviert)	6	11
Opera	11.0 (deaktiviert)	-	-
IE	-	IE 10 developer preview	-

Tabelle 1: WebSocket Browserunterstützung [2]

In diesem Abschnitt wird gezeigt, wie WebSockets dennoch schon heute browserübergreifend einen Mehrwert für Webapplikationen bringen können.

4.1 Alternative Technologien

Wie bereits dargelegt wurde, ermöglichen WebSockets gegenüber HTTP eine bidirektionale Kommunikation zwischen Webbrowser und Webserver. In der Praxis haben Webentwickler aber bereits verschiedene technische Alternativen auf der Basis von HTTP eingesetzt, die eine ähnliche Kommunikation ermöglichen.

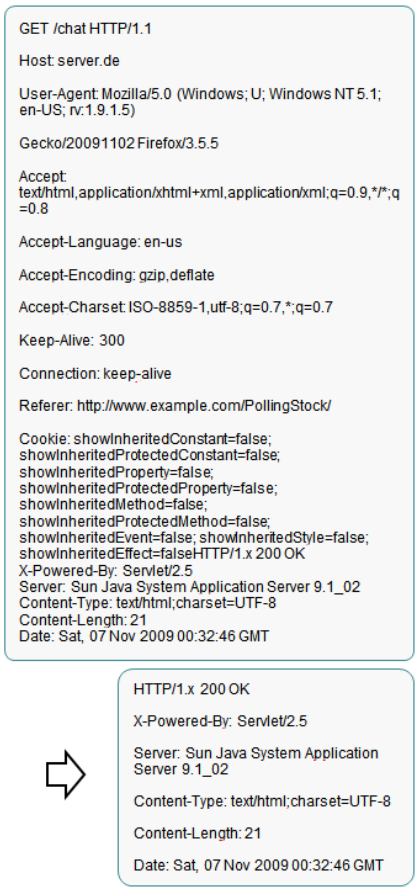

Abbildung 3: Beispiel HTTP-Header [8]

Abbildung 4: Netzwerktraffic Polling / WebSockets (in Bit/s) bei einem Abfrageintervall von 1 Sekunde; logarithmische Skalierung (Basis 10); nach eigener Berechnung

doi: 10.2313/NET-2012-04-1_8

4.1.1 Comet

Ein Ansatz, einen Server-Push ohne WebSockets zu realisieren, ist der Einsatz des Web-Application Models „Comet“[4]. Dieses umfasst verschiedene technische Methoden, HTTP-Requests an einen Server zu schicken, ohne dass dieser die Anfrage sofort beantwortet. Stattdessen wird die Verbindung so lange aufrechterhalten, bis dem Server neue, für die Anfrage relevante Daten zur Verfügung stehen, die dann über den bereits geöffneten Request an den Client geschickt werden können [9]:

Streaming

Eine dieser Varianten ist „Streaming“. Hierzu wird ein HTTP-Request z.B. als Ziel-URL eines iFrames an den Server gesendet. Dieser beantwortet die Anfrage aber nie vollständig, sondern sendet neue Daten immer scheibchenweise, sobald diese verfügbar sind. In einem iFrame lässt sich dies z.B. dadurch realisieren, dass die einzelnen Datenblöcke in JavaScript-Code eingefasst und gesendet werden, der dann direkt im Browser ausgeführt wird. Die Daten können so unmittelbar von der Anwendung verwertet werden.

Longpolling

Eine weitere Technik, um einen Server-Push zu realisieren, ist Longpolling. Wieder wird ein HTTP-Request z.B. via Ajax oder iFrame gestartet, der nicht direkt beantwortet wird. Stehen dem Server neue Daten zur Verfügung, beantwortet er den offenen Request und beendet die Verbindung. Der Client verarbeitet die Daten und schickt sofort eine neue Anfrage, die dann wieder bis zur nächsten Antwort offen gehalten wird. [9]

4.1.2 Sockets über Flash

Eine weitere Möglichkeit um mit WebSockets bereits heute eine größere Reichweite zu erzielen, ist der Einsatz von WebSockets über den Adobe Flash Player. Dieser erlaubt seinerseits bereits seit Version 10 das Öffnen von Socket-Verbindungen zu einem Server. Dies erfordert allerdings eine individuelle Konfiguration des Servers, da z.B. ein zusätzlicher Port in der Firewall geöffnet werden muss. Technische Einschränkungen bestehen außerdem, wenn Proxys eingesetzt werden und der Flash-Player die Proxy-Einstellungen des eingesetzten Browsers nicht auslesen kann. [6]

4.2 Technologie-Fallback via Socket.IO

Wie in diesem Abschnitt gezeigt, existieren verschiedene Techniken, um bei Abwesenheit einer nativen WebSocket-Implementierung des Browsers eine bidirektionale Verbindung zu einem Server aufzubauen oder zumindest zu simulieren. Diese haben jedoch kein einheitliches Interface und können teilweise nur unter bestimmten Voraussetzungen, wie dem Vorhandensein des Flash Players, eingesetzt werden.

Aus diesem Grund wurde die JavaScript-Library Socket.IO[5] entwickelt, die ein einheitliches Interface für den Einsatz von Server-Push Funktionalität anbietet. Je nach Verfügbarkeit verschiedener Technologien des eingesetzten Webbrowsers werden native WebSockets, Flash-Sockets oder Comet-Technologien eingesetzt, um eine möglichst große Abdeckung unter den potentiellen Besuchern einer Web-Applikation zu erreichen.

[4] s. http://cometdaily.com/

[5] s. http://socket.io

5. ZUSAMMENFASSUNG UND AUSBLICK

Mit dem zunehmenden Wandel von statischen Hypertexten zu dynamischen Web-Applikationen erzeugt das HTTP-Protokoll einen Overhead an benötigten Netzwerkressourcen und unnötige zeitliche Verzögerungen bis zur Darstellung aktualisierter Webseiten-Inhalte.

Auf Basis des WebSocket-Protokolls steht Webentwicklern eine schlanke und gleichzeitig mächtige API zur Verfügung. Mit dieser kann der Overhead an verwendeten Ressourcen für Echtzeit-Applikationen minimiert und das Zeitverhalten gegenüber Technologien wie Polling verbessert werden. Durch die Konzeption des Protokolls als Upgrade des bestehenden HTTP-Protokolls können WebSocket-Verbindungen ohne großen technischen Mehraufwand in bestehende Internet-Infrastrukturen integriert werden.

Die Schwierigkeiten bei der Programmierung auf Basis des WebSocket-Protokolls liegen derzeit jedoch in der geringen Browserkompatibilität. Da der WebSocket-Internetstandard zum Zeitpunkt der Arbeit noch sehr neu ist und Webseiten meist Kompatibilitäten mit allen wichtigen Webbrowsern und deren verbreiteten Versionen erfordern, wird es noch einige Zeit dauern, bis native WebSockets für ein breites Publikum eingesetzt werden können. Dennoch lassen sich WebSockets schon heute für Web-Applikationen verwenden, indem die Software je nach Browser des Anwenders alternative Technologien wie Longpolling oder Flash-Sockets nutzt. Bibliotheken wie Socket.IO bieten dem Entwickler hierzu ein einheitliches Interface. Auf dieser Basis entwickelte Anwendungen können so schon heute ein verbessertes Zeitverhalten erreichen. Durch zunehmende Browserunterstützung profitieren so entwickelte Applikation mittelfristig automatisch von der Ressourceneffizienz nativer WebSockets.

6. LITERATUR

[1] HTML 5 - A vocabulary and associated APIs for HTML and XHTML. http://www.w3.org/TR/2008/WD-html5-20080610/comms.html. abgerufen am: 10.02.2012.

[2] WebSocket. http://en.wikipedia.org/wiki/WebSocket. abgerufen am: 10.02.2012.

[3] I. Fette and A. Melnikov. The WebSocket protocol. 2011.

[4] I. Hickson. The websocket api. 2011.

[5] L. Huang, E. Chen, A. Barth, E. Rescorla, and C. Jackson. Talking to yourself for fun and profit. *Proceedings of W2SP*, 2011.

[6] H. Ichikawa. HTML5 Web Socket implementation powered by Flash. http://github.com/gimite/web-socket-js, 2011. abgerufen am: 10.02.2012.

[7] J. Ihlenfeld. Websockets werden mit dem RFC 6455 zum Internetstandard. (Bild: IETF). http://www.golem.de/1112/88360.html, 12 2011. abgerufen am: 10.02.2012.

[8] P. Lubbers. Harnessing the Power of HTML5

WebSocket to create scalable Real-Time Applications.
[9] S. Platte. AJAX und Comet. 2010.

doi: 10.2313/NET-2012-04-1_8