heise Developer

Matthias Weßendorf

WebSocket: Annäherung an Echtzeit im Web

Besser jetzt als gleich

Für die Realisierung von "Echtzeit" im Web kamen in der Vergangenheit Hacks zum Einsatz. Mit WebSocket haben Entwickler nun einen Standard in der Hand, der ihnen bidirektionale Kommunikation über eine TCP-Verbindung bietet.

In traditionellen Anwendungen (Client/Server) ist es nichts Ungewöhnliches, dass bei der Veränderung des Zustands eine sofortige Aktualisierung der GUI erfolgt. Unter Verwendung des Observer Patterns lässt sich etwa eine Liste von Einträgen in "Echtzeit" aktualisieren, sobald ein Objekt beispielsweise in der Datenbank gelöscht wurde. Die Aktualisierung erfolgt automatisch, ohne dass ein Benutzer (oder die Anwendung) erfragen muss, ob es einen neuen Zustand gibt.

Das Web als Plattform

Mittlerweile werden vermehrt Anwendungen ins Web gebracht. Die Anwendungen sind anschaulich, allerdings fehlt eine vernünftige Integration von "Echtzeit". In Webanwendungen wird "Echtzeit" häufig durch Polling oder Long Polling simuliert. Oft werden diese Techniken (oder Hacks) durch die Begriffe Comet oder Bayeux beschrieben. Im Fall von Polling wird der Server in einem festgelegten Intervall, beispielsweise alle zwei Sekunden, gefragt, ob er neue Informationen hat. Beim Long Polling wird eine separate HTTP-Verbindung zum Server erst geschlossen, wenn neue Daten verfügbar sind. Nachdem der Browser sie verarbeitet hat, sendet er einen neuen Request zum Server, um auf weitere Updates zu warten. HTTP ist von Natur aus "nur" halbduplex. Das bedeutet, dass für die bidirektionale Kommunikation zwischen Browser und Server ein separater HTTP Request für jede Richtung benötigt wird.

Das erzeugt natürlich einen Menge Overhead. Neben dem zusätzlichen I/O-Handling auf dem Server kommt noch der Netzverkehr hinzu: HTTP Request/Response Header können schnell ein paar Hundert Bytes **veranschlagen[1]**. Hinzu kommt ab und an die eigentlich wertlose Information, dass es keine Änderungen am Zustand des Servers gab.

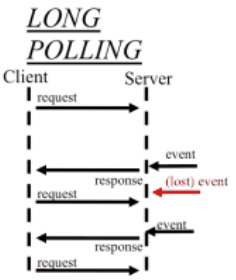

Abb. 2: Long Polling

Echtzeit im Web

Diese Tatsachen erschweren die effiziente Programmierung von Webanwendungen, die auf einen schnellen Datentransfer angewiesen sind. Beispiele dafür gibt es genug, etwa:

- kollaborative Websites
- Internet-Support
- Spiele (auf Basis von HTML5)
- Finanzanwendungen
- Server-Monitoring-Anwendungen im Web

Der Grundgedanke vom Web als Anwendungsplattform ist der, dass es einen vollständigen Ersatz für traditionelle Desktop-Anwendungen bieten soll. Doch beim Punkt "Echtzeit" stößt man schnell an die Effizienzgrenzen von HTTP. Hier kommt der neue Standard WebSocket ins Spiel.

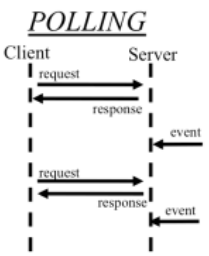

Abb. 1: Polling

Duplexe Kommunikation

WebSocket ist ein bidirektionaler und (voll-)duplexer Kommunikationsstandard und besteht aus einer Client-API und einem Netzwerkprotokoll.

Das W3C (World Wide Web Consortium) hat die JavaScript-API **definiert[2]**, die IETF (Internet Engineering Taskforce) **spezifizierte[3]** das WebSocket-Protokoll. Die Arbeiten zu beiden Spezifikationen begannen im Jahr 2009, eingeleitet durch die Firma Google. Weitere, wie Mozilla, Kaazing oder Opera, sind ebenfalls an der Diskussion beteiligt.

In dem Wort "vollduplex" steckt der große Vorteil von WebSocket. Die bidirektionale Kommunikation zwischen Server und Client läuft über genau einen Kommunikationskanal ab. Sobald die WebSocket-Verbindung aufgebaut ist, können Server und Client gleichzeitig miteinander kommunizieren. In einer frühen Version der Spezifikation wurde WebSocket auch als "TCP für das Web" bezeichnet. Es verwendet die Standard-HTTP-Ports 80 und 443 für TLS, sodass der Verbindungsaufbau auch durch Firewalls hinweg funktioniert. Proxy-Server stellen ebenfalls kein Problem dar, weil im Fall einer solchen Installation automatisch ein Tunnel geöffnet wird: Nutzt der Browser einen Proxy-Server, wird das clientseitig festgestellt und der Proxy-Server wird mit der *HTTP-CONNECT*-Methode angewiesen, über TCP/IP eine Verbindung zu einem bestimmten Host zu öffnen. Zudem lässt sich WebSocket mit bestehenden Authentifizierungs- und Autorisierungs-Frameworks, wie Kerberos, nutzten.

Der Verbindungsaufbau zwischen Client und Server erfolgt mit dem WebSocket Protocol Handshake:

Browser:

```
GET /chatService HTTP/1.1
```

```
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 8
```

Server:

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

Zunächst sendet der Browser eine normale *HTTP GET*-Anfrage an eine Ressource (*/chatService*). Zudem gibt er an, dass er ein Upgrade auf das WebSocket-Protokoll durchführen möchte. Zusätzlich werden beim Verbindungsaufbau verschiedene Key-Value-Pärchen zum Server geschickt. Der *Sec-WebSocket-Key* Header stellt eine Base64-enkodierte Zeichenkette dar, die der Server nutzt, um den Verbindungsaufbau zu akzeptieren. Optional kann der Browser weitere Header an den Server übermitteln. Beispielsweise kann er dem Server mitteilen, ob ein anwendungsspezifisches Protokoll über WebSocket genutzt werden soll, und von ihm unterstützte Protokollversion angeben.

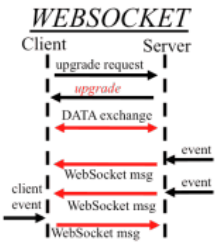

Abb. 3: Der WebSocket Handshake

Der WebSocket-Server antwortet mit dem HTTP-Statuscode 101 und signalisiert damit dem Client, dass er den Upgrade-Wunsch akzeptiert (*Sec-WebSocket-Accept*) und das Upgrade auf das WebSocket-Protokoll vornimmt. Zusätzlich gibt der Server an, dass er das "superchat"-Protokoll kennt. Das hat den Vorteil, dass die Browseranwendung direkt gegen dieses Protokoll beziehungsweise diese API geschrieben wird, statt gegen die WebSocket-API. Entwickler, die mit der Programmierschnittstelle beziehungsweise dem anwendungsspezifischen Protokoll vertraut sind, brauchen keine neue API erlernen, um WebSocket-Anwendungen zu erstellen. Die clientseitige Schnittstelle des "superchat"-Protokolls kapselt die eigentlich Kommunikation mit dem WebSocket-Server.

Mit der Hilfe von JavaScript

Neben dem Protokoll gibt es eine standardisierte JavaScript-API:

```
// Create new WebSocket
var mySocket = new WebSocket("ws://echo.websockets.org");
// Attach listeners
mySocket.onmessage = function(event) { alert("Der Server sagt: " + event.data); };
mySocket.onopen = function(event) {...};
mySocket.onclose = function(event) {...};
mySocket.onerror = function(event) {...};
```

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

- "onopen": Aufruf erfolgt nach dem Herstellen der Verbindung.
- "onclose": Aufruf erfolgt, sobald die Verbindung vom Client oder vom Server geschlossen wurde.
- "onerror": Aufruf erfolgt im Fehlerfall.
- "onmessage": Aufruf erfolgt, sobald Nachrichten vom Server im Client ankommen.

Der *onmessage*-Listener nimmt die ankommenden Daten entgegen. Die liest er aus dem standardisiertem *data*-Feld des übergebenen *event*-Objekts. Die Kommunikation mit dem Server ist ebenfalls einfach:

```
// Send data...
mySocket.send("Hallo Server!");
// Close WebSocket
mySocket.close();
```

Nach dem erfolgreichem Aufbau der WebSocket-Verbindung kann der Client über den bidirektionalen Kommunikationskanal Nachrichten mit der JavaScript-Funktion *send()* an den Server senden. Sobald der Client die Verbindung beenden möchte, kann er die *close()*-Methode aufrufen. In dem Fall wird ebenfalls der *onclose*-Listener aufgerufen.

Der Overhead von WebSocket im Vergleich zu HTTP ist gering. Pro Nachricht beträgt er genau zwei Byte, statt mehrerer Hundert. Eine WebSocket-Nachricht fängt mit einem *0x00*-Byte an und endet mit einem *0xFF*-Byte. Zwischen den zwei Bytes befinden sich die Nutzdaten, kodiert in UTF-8:

```
\0x00Hello, WebSocket\0xff
```

Das vorgestellte Programm lässt sich in einem modernen Browser ausprobieren, da der Server **echo.websocket.org[4]** für Testzwecke im Internet bereitsteht.

Besonders **deutlich wird der minimale Overhead[5]** bei Anwendungen, die eine große Anzahl aktiver Clients haben.

Wie ist der Status quo?

Wie andere HTML5-Standards ist auch der für WebSocket noch nicht zu 100 Prozent verabschiedet, allerdings sind API und Protokoll nah an

einer Fertigstellung, und einem Einsatz von WebSocket in Webanwendungen steht nichts im Wege. Folgende Browser bieten bereits native Unterstützung an:

- Firefox 4/Firefox 5/Firefox 6*
- Google Chrome 6 und Chromium
- Safari 5 und "mobile Safari" (iOS 4)
- Opera 11

Im Firefox 4 und im Opera 11 ist WebSocket wegen einer Sicherheitslücke im Zusammenspiel mit alten Proxy-Servern deaktiviert. Die IETF hat sich des Fehlers in der Version 06 des Protokolls angenommen, sodass die Mozilla Foundation WebSocket mit dem Erscheinen von Firefox 6 (voraussichtlich im Herbst) wieder aktiviert. Auch für den Internet Explorer gibt es WebSocket-Unterstützung, allerdings ist dazu ein Plug-in von Microsoft nötig. In älteren Browser funktioniert WebSocket **nicht ohne zusätzliche Bibliotheken[6]**.

Auch bieten mehrere Server Unterstützung für WebSocket, etwa:

- Apache httpd (via "mod_pywebsocket")
- Jetty
- jWebSocket
- Kaazing WebSocket Gateway
- Oracle Glassfish 3.1
- Netty Project
- Node.js/Socket.io

Die richtige Anwendung

Wie erwähnt, setzt WebSocket an, die komplexen Probleme des "Echtzeit-Web", wie Latenz oder Netzverkehr, anzugehen. WebSocket wird jedoch *nicht* als ein "besseres AJAX" entwickelt . Ebenfalls stellt WebSocket keinen 1:1-Ersatz für HTTP dar, sondern bietet vielmehr einen effizienten, bidirektionalen Kommunikationskanal an. Moderne Webanwendungen sollten ihn nutzen, um hochwertigere Protokolle beziehungsweise APIs zum Browser zu bringen. Traditionelle Client-Server-Anwendungen kommunizieren in der Regel nicht direkt über "rohes" TCP, sondern sie nutzen hochwertige Protokolle wie JDBC, XMPP oder JMS. Mit WebSocket ist es möglich, diese und andere Protokolle zum Browser zu zu überführen. JMS (Java Messaging Service) ist eine Java-API für das Versenden und Empfangen von Nachrichten im Kontext von "Message Oriented Middleware" (MOM). Ein Client kann so mit der Middleware kommunizieren oder Nachrichten von ihr empfangen (siehe **hier[7]**). Mit WebSocket lässt sich JMS zum Browser hin erweitern. Das folgende Beispiel stellt die JavaScript-API des **Kaazing WebSocket Gateway[8]** vor:

```
...
var stompConnectionFactory = new
StompConnectionFactory("ws://my.server.com:8000/jms")
var connectionFuture = stompConnectionFactory.createConnection(function () {
   if (!connectionFuture.exception) {
     connection = connectionFuture.getValue();
     connection.setExceptionListener(handleException);
     session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
     topic = session.createTopic("/topic/destination");
     // creating some consumers!
     var consumer1 = session.createConsumer(topic);
     consumer1.setMessageListener(handleMessageCallback);
     connection.start(someCallback);
   }
...
});
```

Der Verbindungsaufbau erfolgt mit einer *ConnectionFactory*, die die URL des WebSocket-Servers entgegennimmt. Tritt kein Fehler auf, werden eine JMS Session und einTopic angelegt. Das Topic wird genutzt, um einen sogenannten *Consumer* zu erzeugen. Damit er Nachrichten vom Topic empfangen kann, benötigt er einen JMS Message Listener. Die *handleMessageCallback()*-JavaScript-Funktion wird aufgerufen, sobald Nachrichten vom JMS-Broker im Browser eintreffen:

```
function handleMessageCallack(message) {
   // did Apache ActiveMQ send us a JMS TextMessage?
   if (message instanceof TextMessage) {
       var body = message.getText();
       // do more stuff...
   }
}
```

Fazit

Der JavaScript-Callback (*handleMessageCallback*) funktioniert ähnlich wie ein in Java geschriebener MessageListener: Er bekommt eine Nachricht und kann deren Nutzdaten (*message.getText()*) nutzen, um sie zu visualisieren.

Zusammenfassend lässt sich sagen, dass der WebSocket-Standard die Entwicklung einer neuen Generation von Anwendungen ermöglicht. Die Integration von "Echtzeit" innerhalb von Webanwendungen ist nicht mehr an Hacks und Workarounds gebunden, sondern erfolgt auf Basis eines standardisierten, effizienten und bidirektionalen Protokolls. Wichtig ist hierbei, dass man sämtliche TCP/UDP-Protokolle auf Basis von WebSocket zum Browser bringen kann. Der Abstraktionsgrad zukünftiger Webanwendungen steht damit den Desktop-Anwendungen in nichts nach. Statt gegen TCP/WebSocket zu programmieren, können Entwickler bereits vertraute Protokolle beziehungsweise APIs einsetzen. Besonders interessant ist das wiederum für Webspiele. Bei ihnen ist, im wahrsten Sinne des Wortes, Latenz tödlich. (rl)

*Matthias Weßendorf (**Blog[9]**)*
arbeitet für die Firma Kaazing, die einen kommerziellen WebSocket-Server anbietet. Dort beschäftigt er sich unter anderem mit AMQP, JMS, .NET, Flash und HTML5, um das "Next Generation Web" voranzutreiben.

URL dieses Artikels:
http://www.heise.de/developer/artikel/WebSocket-Annaeherung-an-Echtzeit-im-Web-1260189.html

Links in diesem Artikel:
[1] http://websocket.org/quantum.html
[2] http://dev.w3.org/html5/websockets/
[3] http://tools.ietf.org/html/draft-ietf-hybi-thewebsocketprotocol
[4] http://echo.websocket.org
[5] http://websocket.org/quantum.html
[6] http://tech.kaazing.com/documentation/html5/release-notes.html#browser_matrix
[7] http://java.sun.com/developer/technicalArticles/Ecommerce/jms/index.html
[8] http://kaazing.com/download.html
[9] http://matthiaswessendorf.wordpress.com