Studienarbeit im Fach Aktuelle Technologien zur Entwicklung Verteiler Java Anwendungen

Thema: Server an Client - Push mit WebSockets

Verfasser: Christian Linha

Studiengang: Informatik Bachelor

Studiengruppe: IF6

Matrikelnummer: 43964513

Datum: 06.05.2016

Dozent: Michael Theis

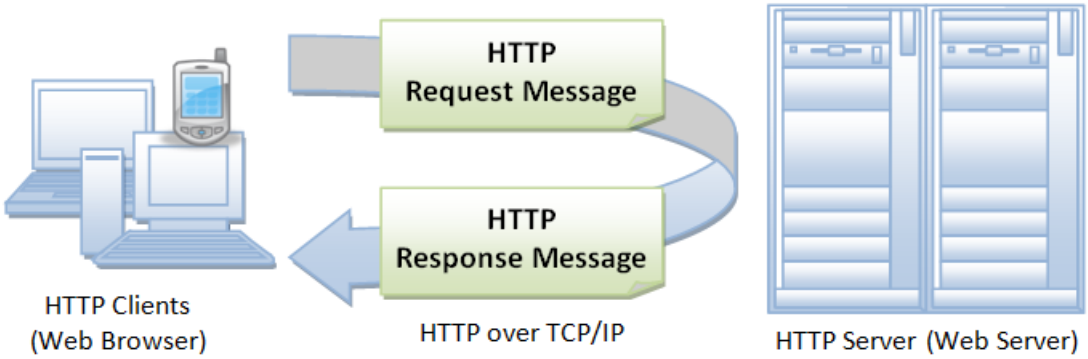
Inhaltsverzeichnis

1. Einleitung 3
1.1 Polling 4
1.2 Long Polling 5
2. Grundidee Websockets 6
3. Websocket Protokoll 7
3.1 Verbindungsaufbau 7
3. 2 Handshake 7
3.3 Datenübertragungen 9
3.3.1 Der Aufbau eines Frames 9
3.3.2 Client Payload-Data Maskierung 11
4. Websockets in der Praxis 12
5. Programmieren mit Websockets 13
6. Programmbeispiel: Serverseitige Uhr 15
6.1 Die clientseitige Javascript Anwendung: 15
6.2 Die serverseitige Java Anwendung: 18
6.3 Die fertige Anwendung 22
7. Fazit 23

1. Einleitung

Das Internet wurde zu Beginn an um das Request – Response Muster des HTTP Protokolls entwickelt. Ein Client, was in den meisten Fällen ein Browser ist, fordert Daten von einem Webserver an. Dies kann z.B eine Webseite in Form eines HTML Dokumentes sein. Die Daten werden in einem HTTP Requestes angefordert. Der Server antwortet daraufhin in einem HTTP Response und stellt dem Client darin die gewünschten Daten bereit.

Die Folgende Grafik veranschaulicht das Grundprinzip des HTTP Protokolles:



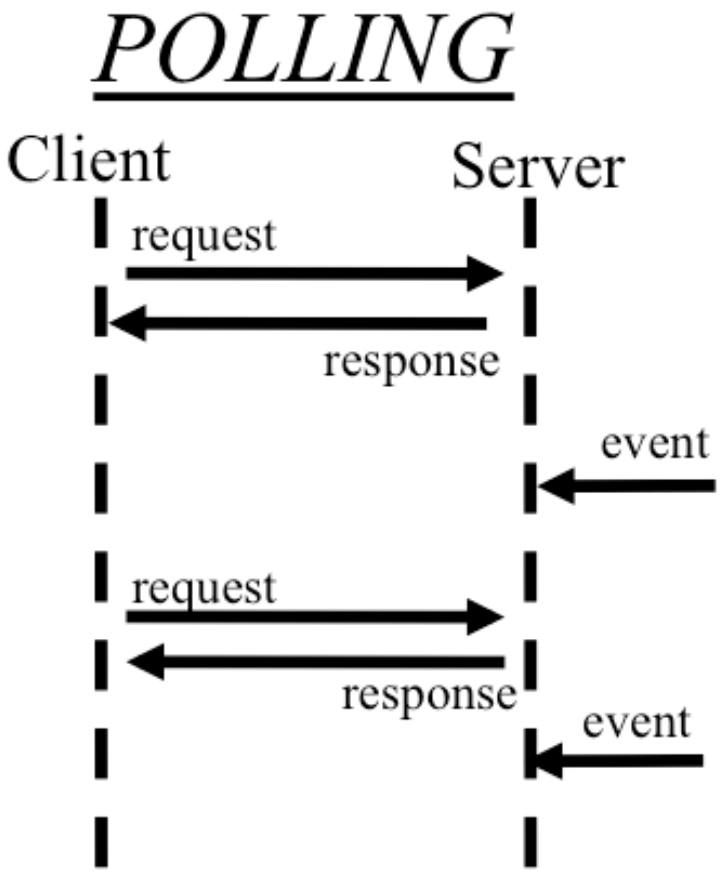
Da im Laufe der Zeit von den Webanwendungen immer mehr Dynamik abverlangt wurde, kam das HTTP Protokoll an seine Grenzen.

Bei dem HTTP Protokoll wird jeder Datenaustauch vom Client angetriggert. Das Problem an dem HTTP Protokoll ist nun der Fall, wenn der Server Daten bzw. ein Event an den Client versenden will. Diesen Vorgang nennt man Server-Push. In HTTP ist dies jedoch nur über Umwege möglich.

1.1 Polling

Eine Möglichkeit wäre, das sogenannte Polling. Mit dieser Methode sendet der Client periodisch Requests an den Server. In dem Request fragt der Client den Server, ob neue Daten vorhanden sind. Der Nachteil dieser Methode ist, dass der Server und die Verbindung dauerhaft mit Anfragen von dem Client ausgelastet werden, auch wenn gar keinen neuen Daten bereit stehen. Somit erzeugt diese Methode einen großen Overhead und verschwendet nur die Kapazitäten der Verbindung bzw. des Servers.

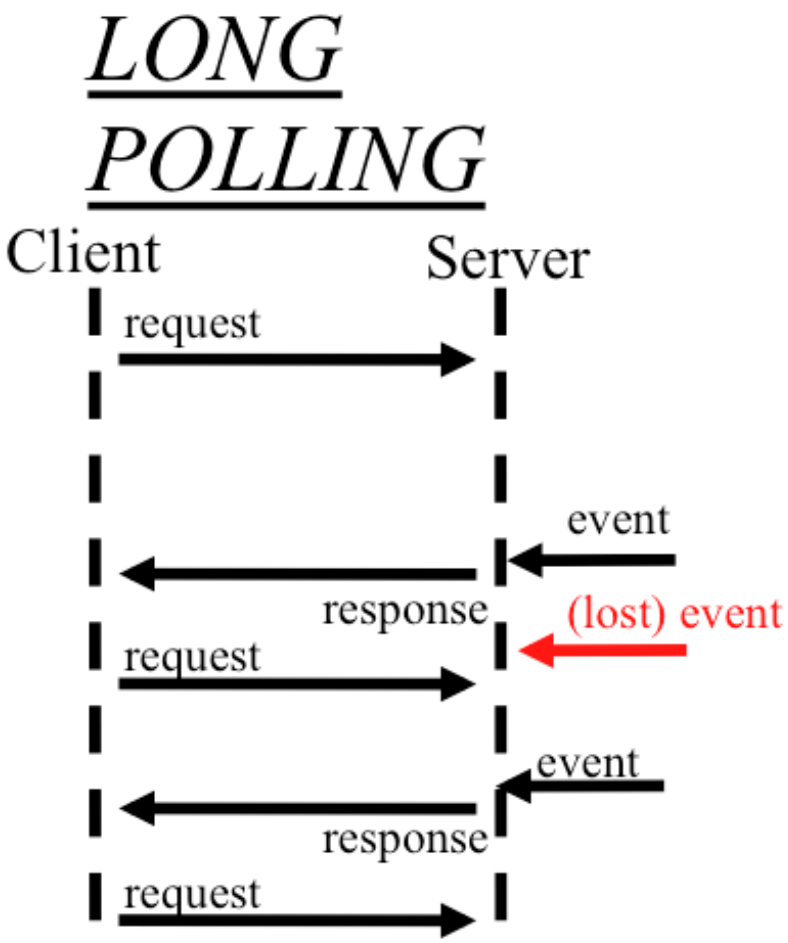
Die folgende Grafik soll den Ablauf des Pollings veranschaulichen:



1.2 Long Polling

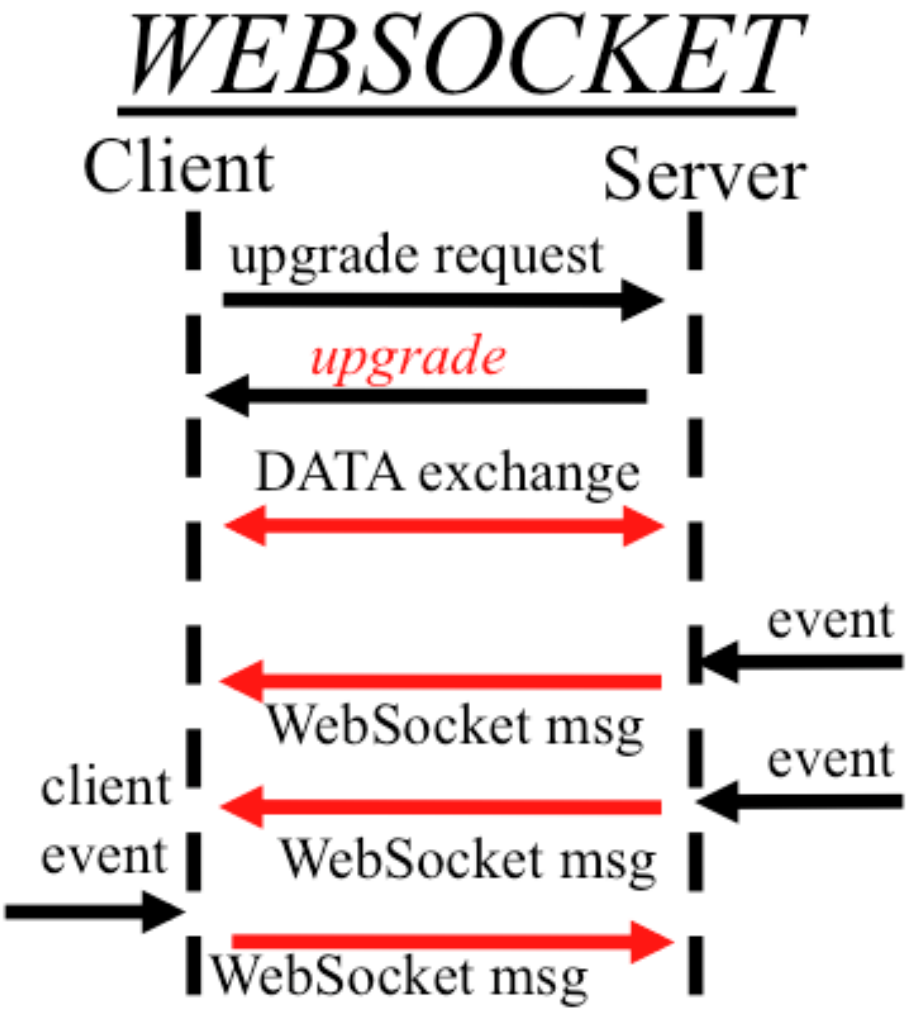
Eine alternative wäre das Long Polling. Bei dieser Methode sendet der Client einen HTTP Request an den Server. Der Server antwortet jedoch nicht direkt sondern wartet bis neue Daten für den Client verfügbar sind. Erst dann wird der Client mit dem HTTP Response darüber benachrichtigt. Die Gefahr bei dieser Methode ist jedoch, dass Events vom Server verloren gehen können. Dies ist der Fall wenn zwischen dem Response vom Server und einem erneuten Request des Clients ein Events stattfindet.

Die folgende Grafik zeigt den Ablauf des Long Pollings und veranschaulicht auch die Gefahr, ein Update zu verlieren:



2. Grundidee Websockets

Eine bessere Alternative wäre eine bidirektionale Kommunikation zwischen Client und Server. Diese Idee wurde mit dem Websocket Protokoll aus dem HTML 5 Standard realisiert. In dem Websocket Protokoll können sowohl Client als auch Server Nachrichten an das andere Ende der Verbindung versenden. Gleichzeitig können auch beide parallel dazu Nachrichten empfangen und verarbeiten. Die folgende Grafik zeigt die Kommunikation zwischen Client und Server über das Websocket Protokoll:



Wie man aus der Grafik entnehmen kann, wird der Client sofort über ein Event auf dem Server benachrichtigt. Währenddessen kann der Client unabhängig von eingehenden Daten auch Nachrichten an den Server versenden.

3. Websocket Protokoll

3.1 Verbindungsaufbau

Eine Websocket Verbindung wird über das HTTP Protokoll aufgebaut. Der Port an dem Server angesprochen wird ist deshalb wie auch beim HTTP Protokoll Port 80 bzw. 443 falls eine Verschlüsselte Verbindung stattfinden soll. Der Vorteil daran ist, dass der Server dadurch sowohl Websocket als auch HTTP Verbindungen an einem Port verarbeiten kann.

Das Websocket Protokoll definiert 2 neue URI Schemas:

```
ws://<host>  [ :<port> ] path [ ?<query> ]
wss://<host> [ :<port> ] path [ ?<query> ]
```

Die beiden URIs sind denen einer HTTP Verbindung sehr ähnlich. Der einzige Unterschied ist das Protokoll Prefix. Das Prefix ‚`ws`' wird für eine unverschlüsselte und das ‚`wss`' Prefix wird für eine verschlüsselte Verbindung genutzt. Bei dem Host handelt es sich um den Domainnamen bzw. die IP des Servers. Optional kann der Port angegeben werden. Damit kann auch einen Websocket Verbindung über einen andren Port z.B. den alternativen HTTP Port 8080 aufgebaut werden. Der Pfad gibt die Ressource des Servers an, mit der die Verbindung aufgebaut werden soll.

3. 2 Handshake

Beim Aufbau einer Websocket Verbindung findet zu Beginn ein Handshake statt. Der Client sendet als erste Nachricht wie auch im HTTP Protokoll eine HTTP Anfrage. Der Header dieser Anfrage sieht folgend aus:

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

Die ersten zwei Zeilen gleichen dem HTTP Header und signalisieren, auf welche Ressource auf dem Server zugegriffen werden soll. Mit den Feldern ‚Connection' und ‚Upgrade' fordert der Client ein Upgrade auf das Websocket Protokoll. Der Wert des Feldes ‚Sec-WebSocket-Key' ist ein zufällig generierter Wert, welcher für den Handshake benötigt wird. Das Feld ‚Sec-Websocket-Protocol' gibt an, um welche Version des Websocket Protokoll es sich handelt.

Der Header der Antwort des Servers ist folgende:

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

In den ersten 3 Zeilen signalisiert der Server, dass das Protokoll nun auf das Websocket Protokoll geändert wird. Mit dem Feld ‚Sec-WebSocket-Accept' signalisiert der Server, dass die Anfrage des Clients gelesen und verarbeitet wurde. Der Wert des Feldes wird in folgenden Schritten erzeugt:

1. Als erstes wird der Wert des Feldes ‚Sec-WebSocket-Key' von dem empfangenen Request ausgelesen.
2. An diesen wird anschließend die Konstante ‚258EAFA5-E914-47DA-95CA-C5AB0DC85B11' angehängt
3. Aus dem zusammengesetzten String wird dann ein SHA1-Hash gebildet
4. Der Hash wird anschließend Base64 kodiert

Wenn der Client die Schritte 2 – 4 mit dem gesendeten Wert von ‚Sec-WebSocket-Key' auch durchführt, kann dieser überprüfen, ob das Ergebnis mit dem des empfangenen ‚Sec-WebSocket-Accept' Wertes übereinstimmt.

Falls die Werte übereinstimmen weiß der Client, dass der Server die Anfrage angenommen und Verarbeitet hat.

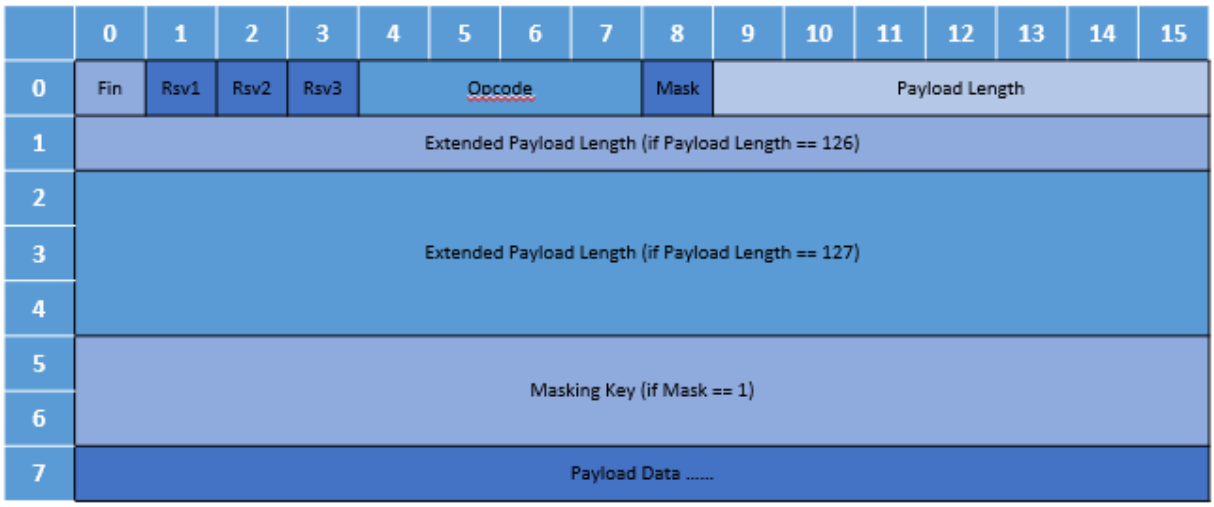
Der Handshake ist danach abgeschlossen und die Verbindung ist dann aufgebaut. Eine Übertragung kann nun bidirektional stattfinden.

3.3 Datenübertragungen

Die Daten werden beim Websocket Protokoll in Form von Messages und nicht als Bytestrom versendet. Zur Übertragung einer Nachricht wird diese auf eines oder mehrere Frames aufgeteilt. Die Frames werden dann sequentiell an den Empfänger versendet. Die Übertragung der Frames findet auf der zwischen dem Client und Server aufgebauten TCP Socket Verbindung statt. Der Empfänger kann aus den empfangenen Frames die Message wieder zusammen bauen.

3.3.1 Der Aufbau eines Frames

Die folgende Grafik zeigt den bitweisen Aufbau eines Frames:



Ich möchte nun die auf Inhalte eines Websocket Frames eingehen:

Bit 0: FIN

Dieses Bit signalisiert, ob es sich um den letzten Frame der Nachricht handelt.

Bit 1 – 3: REV

Reserviert für zukünftiges Nutzen.

Bits 4 – 7: OPCODE

Hier handelt es sich um den OP Code der Nachricht. Der Opcode signalisiert, wie die Nachricht interpretiert werden soll.

Bit 8: MASK

Dieses Bit gibt an, ob die Payload Data der Nachricht maskiert wird.

Bit 9 – 15: PAYLOAD LENGTH

Diese 7 Bits enthalten die Länge der Payload in Bytes.

15 - 79: EXTENDED PAYLOAD LENGTH

Mit den 7 Bits der Payload Length lassen sich ein maximaler Wert von 127 bilden. Falls mehr als 127 Bytes in dem Frame übertragen werden sollen, kann dies über diese Bits angegeben werden.

Bit 80 – 111: MASK KEY

Falls das Mask Bit gesetzt wurde, enthalten diese Bits den Wert des Mask-Keys.

Ab Bit 112:

Hier beginnen die die Nutzdaten der zu versendeten Message. Um wie viele Bytes es sich handelt, wird in den Payload Length Feldern angegeben. Dadurch weiß der Empfänger, wie viele Bytes vom Eingabestrom noch gelesen werden müssen, um das Frame komplett einzulesen.

3.3.2 Client Payload-Data Maskierung

In den Spezifikationen des Websocket Protokolles ist spezifiziert, dass jede von einem Client gesendete Nachricht maskiert werden muss. Bei der Maskierung handelt es sich um die Transformation der Payload mit den Mask-Key als Parameter. Im Folgenden wird der Algorithmus an einem Java Beispiel erklärt:

```
// payload data
byte [] payload;

// mask key
byte [] key;

//transformed payload data
byte [] transformed-payload;

…

for (int index = 0; index < payload.length(); i++)
{
        int j = index % 4;
        transformed-payload[i] = payload[i] XOR key[j]
}
```

Für jedes Byte in der Payload Data werden folgende Schritte angewendet:

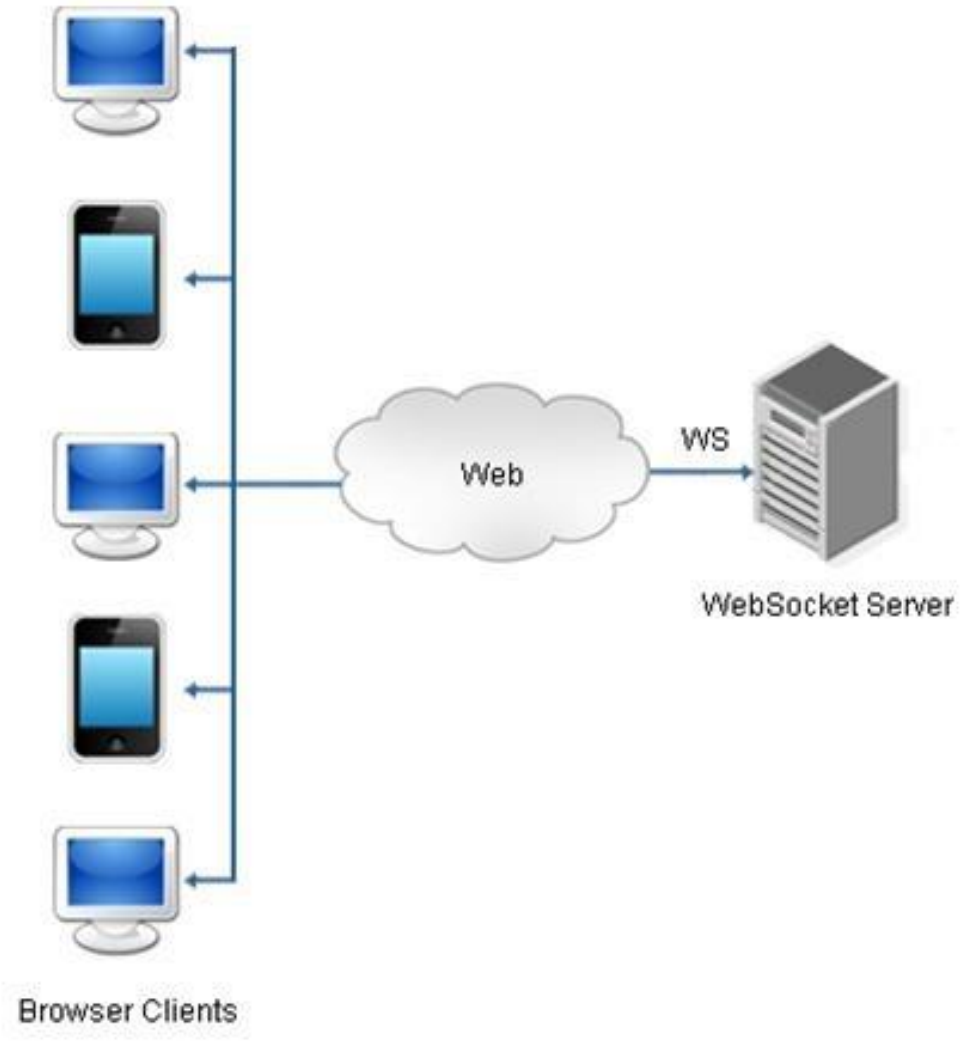
- Errechne den Wert j aus dem Index des aktuellen Bytes Modulo 4
- Führe eine XOR Verknüpfung mit dem aktuellen Byte und dem Byte an der Stelle j im Mask-Key durch
- Das Ergebnis ist das Byte an der Stelle mit demselben Index in den transformierten Daten

Die Länge der maskierten Daten bleibt gleich der Länge der Originaldaten, weshalb es sich bei dem Vorgang um eine Transformation handelt.

4. Websockets in der Praxis

Das Websocket Protokoll wurde im Fokus auf Webanwendungen entwickelt. Deswegen handelt es sich bei dem Client in der Regel um einen Webbrowser. Alle gängigen Webbrowser(Firefox, Chrome, etc) unterstützen das Websocket Protokoll. Der Verbindungsaufbau und das Nachrichtenhandling wird in der Regel in der Sprache Javascript implementiert.

Die auf dem Server laufende Anwendung ist dagegen meist in einer „richtigen“ Programmiersprache wie Java/C#/C++ implementiert.



Es ist jedoch auch möglich, dass z.B. zwei Java Anwendungen über das Websocket Protokoll kommunizieren.

5. Programmieren mit Websockets

Inzwischen gibt es in allen gängigen Sprachen Bibliotheken, welche das Websocket Protokoll implementieren. Auch wenn sich die Sprachen alle unterscheiden, so sind die Konzepte der Websocket Programmierung dieselben.

Bei einer normalen HTTP Kommunikation über Sockets ist der Datenaustausch in der Regel synchron. Der Client sendet eine Anfrage an den Server. Dieser verarbeitet die Anfrage daraufhin und sendet das Ergebnis an den Client. Ein Client muss also immer erst eine Anfrage senden muss, bevor er eine Nachricht vom Server bekommt. Außerdem kann ein Client kann nach einer Anfrage direkt auf die Antwort warten und wird diese garantiert erhalten – vorausgesetzt der Server ist erreichbar. Dadurch lässt sich eine synchrone Kommunikation in einem Thread realisieren.

Die Programmierung mit Websockets unterscheidet sich in einigen Punkten von der normalen Socketprogrammierung. Bei einer Websocket Verbindung kann der Client zu jedem Zeitpunkt Nachrichten vom Server erhalten. Dadurch muss das Programmierkonzept geändert werden. Die Kommunikation über das Websocket Protokoll ist dadurch nicht mehr synchron, sondern wird asynchron realisiert. Für das verarbeiten der Nachrichten werden dazu Handlerfunktionen auf beiden Seiten implementiert. In diesen wird die Behandlung verschiedener Events implementiert. Falls eines dieser Events auftritt wird die entsprechende Handlerfunktion parallel zum restlichen Programmablauf aufgerufen. In der Regel gibt es 4 Stück von diesen Funktion, welche in – egal welche Bibliothek/Sprache verwendet wird - implementiert werden müssen:

onOpen:

Diese Funktion wird aufgerufen, wenn die Verbindung aufgebaut wurde.

onClose:

Diese Funktion wird aufgerufen, wenn die Verbindung geschlossen wurde.

onError:

Diese Funktion wird aufgerufen, wenn ein Fehler in der Verbindung auftritt.

onMessage:

Diese Funktion wird aufgerufen, wenn eine Nachricht empfangen wird.

Als Parameter enthalten die Funktionen in der Regel relevante Daten wie z.B. die Daten der Nachricht bei der Funktion onMessage.

6. Programmbeispiel: Serverseitige Uhr

Ich möchte nun die in der Praxis gängige Kommunikation zwischen Browser und Webserver über Websockets an einem Programmbeispiel aufzeigen. Das Beispiel ist eine Java Webanwendung, welche eine einfache Uhr realisiert. Es kommuniziert eine clientseitige Javascript Anwendung aus einem Browser mit einem in Java implementierten Websocket Server. Der Websocket Server sendet im Sekundentakt die aktuelle Zeit des Servers als Websocket Nachricht an den Client. Die Javascript Anwendung zeigt die aktuelle Zeit im Browser an. Damit kann eine Uhr realisiert werden, welche auf dem Client die Zeit des Servers anzeigt.

Als IDE wird Eclipse EE verwendet. Das Projekt wurde als dynamisches Web Projekt realisiert.

6.1 Die clientseitige Javascript Anwendung:

Die Clientseitige Anwendung wurde in einem HTML Dokument eingebettetem Javascript realisiert. Die Anwendung ist in der Lage, Websocket Nachrichten an den Server zu versenden. Parallel dazu empfängt die Anwendung die aktuelle Uhrzeit des Servers im Sekundentakt. Die empfangenen Zeitdaten werden anschließend auf dem Bildschirm ausgegeben. Ich möchte nun die relevanten Codestellen erklären.

Als erstes wird die URI des Websocket Servers festgelegt:

```
var wsUri = "ws://localhost:8080/WebsocketDemo/Clock";
```

Da das Beispiel auf einem lokalen Server läuft, wird als Server ‚`localhost`‘ angegeben. Die Kommunikation läuft über den alternativen HTTP Port 8080. Der Pfad der Serveranwendung ist ‚`/WebsocketDemo/Clock`‘

Mit folgenden Zeilen lässt sich nun eine Websocket Verbindung realisieren:

```
//initialize websocket connection
websocket = new WebSocket(wsUri);

//initialize Handler functions
websocket.onopen = function(evt) { onOpen(evt) };
websocket.onclose = function(evt) { onClose(evt) };
websocket.onmessage = function(evt) { onMessage(evt) };
websocket.onerror = function(evt) { onError(evt) };
```

Als erstes wird ein Websocket Connection Objekt mit dem Parameter ‚`wsUri`' als die Ziel URI des Websocket Servers erzeugt. Anschließend werden dem Objekt die 4 Handlermethoden ‚`onOpen`',‚`onClose`',‚`onMessage`',‚`onError`' zugewiesen. Das eingehende Event wird den Methoden dabei als Parameter übergeben. Die Websocket Verbindung wird automatisch geöffnet, sobald das Websocket Objekt erzeugt und mit den Handlermethoden initialisiert wurde.

Die Handlermethoden wurden folgenderweise implementiert:

```
//handle connection open event
function onOpen(evt)
{
	writeToScreen("CONNECTED to " + wsUri);
	setTime("Waiting for time update...");
}

//handle connection close event
function onClose(evt)
{
	writeToScreen("DISCONNECTED from " + wsUri);
	setTime("Connection closed");
}

//handle message event
function onMessage(evt)
{
	writeToScreen("RECEIVED MESSAGE: " + evt.data);
	setTime(evt.data);
}

//handle error event
function onError(evt)
{
	writeToScreen("ERROR: " + evt.data);
	setTime("Connection error");
}
```

Die Handler ‚onOpen', ‚onClose', ‚onError' wurden mit einer Simplen Statusausgabe implementiert. Die Handlermethode ‚onMessage' setzt die Zeitausgabe auf den empfangenen Zeitwert.

Außerdem wurde auch noch eine Methode implementiert, mit welcher das Versenden von Nachrichten an den Client ermöglicht wird.

```
//send a message
function doSend(message)
{
        writeToScreen("MESSAGE SENT: " + message);
        websocket.send(message);
}
```

Diese sehr simple Methode ruft die Methode ‚send' auf dem Websocket Objekt auf und übergibt die zu versendete Nachricht als Parameter.

Zum Schluss wurde noch zwei Methoden zur Bildschirmausgabe implementiert:

```
//write something to output screen
function writeToScreen(message)
{
        var pre = document.createElement("p");
        pre.style.wordWrap = "break-word";
        pre.innerHTML = message;
        output.appendChild(pre);
}

//write time to screen
function setTime(time)
{
        var t = document.getElementById("time");
        t.innerHTML = "Time: " + time;
}
```

Die Methode ‚writeToScreen' zeigt einen String auf dem Ausgabefeld der Anwendung an.

Die Methode ‚setTime' zeigt eine String auf der Ausgabe der Uhr an.

6.2 Die serverseitige Java Anwendung:

Der Websocket Server wurde als serverseitige Java Anwendung implementiert. Diese kann anschließend auf einem Java Anwendungsserver „deployed“ werden. In diesem Beispiel wurde Glassfish als Server genutzt. Für die Kommunikation über das Websocket Protokoll wird die Java Websocket API JSR 356 genutzt.

In der Anwendung wird ein Server Endpoint definiert, welcher in der Lage ist, eine Websocket Verbindung entgegen zu nehmen. Der Client kann anschließen mit der URI, auf welcher der Endpoint „deployed“ wurde, eine Websocket Verbindung aufbauen. Sobald eine Verbindung aufgebaut wurde, wird in der Endpoint Instanz ein Timer gestartet, welcher im Sekundentakt die aktuelle Zeit an den Client versendet. Ich werde nun wieder die relevanten Codeteile der Implementierung erklären.

Als erstes muss die Klasse für den Server Endpoint definiert werden:

```
@ServerEndpoint("/Clock")
public class ClockServerEndpoint
{
	Session session;

	Timer clockTimer;
}
```

Hier ist die Annotation ‚`@ServerEndpoint("/Clock")`‘ wichtig. Diese teilt dem Anwendungsserver mit, dass es sich bei der Klasse um ein Server Endpoint handelt. Der Parameter der Annotation gibt an, auf welchem Pfad der Endpoint erreichbar ist. Da der Pfad des Demoprojektes , `/WebsocketDemo`‘ und der Pfad des Endpoints ‚`/Clock`‘ ist, wird der Endpoint daraus folgend mit der URI ‚`ws://<server>/WebsocketDemo/Clock`‘ angesprochen.

Anschließend werden wie im Client auch hier die die 4 Handlermethoden ‚onOpen', ‚onClose', ‚onMessage', ‚onError' implementiert:

```
@OnOpen
public void onOpen(Session session)
{
	this.session = session;
	System.out.println("Created new Connection with Session ID: " + session.getId());

	doSend("Connected!", session);

}


@OnMessage
public void onMessage(String message, Session session)
{
	System.out.println("Message received from Session ID: " + session.getId() + ": " + message);
}


@OnClose
public void onClose(Session session)
{
	this.session = null;
	this.clockTimer.cancel();

	System.out.println("Connection closed on Session: " + session.getId());
}

@OnError
public void onError(Throwable t)
{
	try
	{
		session.close();
	}
	catch (IOException e)
	{
		e.printStackTrace();
	}
	this.session = null;
	this.clockTimer.cancel();

	System.out.println("Connection Error: " + t.getMessage());
}
```

Mit den Annotations ‚OnOpen' ‚OnClose' ‚OnMessage' ‚OnError' wird dem Anwendungsserver mitgeteilt, um welche Art von Handler es sich bei der Methode handelt.

Der Handler ‚onOpen' erhält als Argument ein ‚Session' Objekt welches die aufgebaute Websocket Verbindung repräsentiert. Dieses wird als Member in der aktuellen Endpoint instanz gespeicher. Dies ist notwendig, damit der parallel laufende Timer auf dem ‚Session' Objekt anschließend Nachrichten übertragen kann.

In der Handlermethode ‚onClose' wird der Member für Session auf ‚null' gesetzt, da die Verbindung geschlossen wird. Anschließend wird der Timer für das Versenden der Nachrichten beendet.

Der Handler ‚onError' wird aufgerufen, wenn ein Fehler in der Verbindung aufgetreten ist. In diesem Beispiel wird bei Auftreten eines Fehlers die Verbindung geschlossen, das Session Objekt auf ‚null' gesetzt und der Timer beendet.

Die Handlermethode ‚onMessage' enthält als Parameter die empfangene Message als String. In diesem Beispiel wir bei empfangen einer Nachricht diese nur als Textausgabe in der Serveranwendung ausgegeben.

Außerdem wurde eine Methode zum Versenden von Nachrichten implementiert:

```
public void doSend(String message, Session session)
{
	if(session == null || message == null)
		return;

	try
	{
		session.getBasicRemote().sendText(message);
	}
	catch (IOException ex)
	{
		ex.printStackTrace();
	}
}
```

In dieser Methode wird mit dem Aufruf ‚`session.getBasicRemote().sendText(message)`‘ eine Textnachricht auf dem in der Endpoint Instanz gespeicherten Session Objekt versendet.

Zuletzt muss noch ein TimerTask definiert werden, welcher für das versenden der aktuellen Zeit zuständig ist:

```
private class ClockTimerTask extends TimerTask
{

	@Override
	public void run()
	{
		if(session == null)
			return;

		String message = new SimpleDateFormat("HH:mm:ss").format(new Date());

		doSend(message, session);

	}
}
```

Dieser erzeugt einen String der aktuellen Zeit im Format HH:mm:ss und versendet diesen anschließend mit der implementierten ‚`doSend`‘ Methode.

Der TimerTask muss nun noch von einem Timer ausgeführt werden:

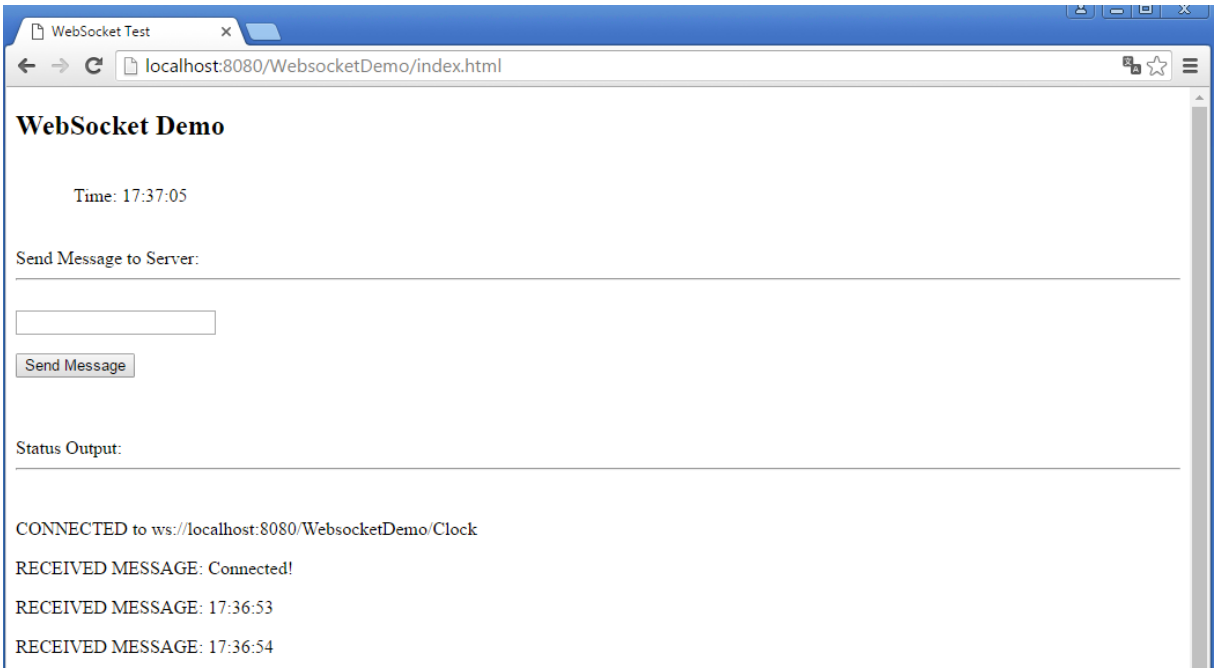
```
public ClockServerEndpoint()
{

	clockTimer = new Timer();
	clockTimer.schedule(new ClockTimerTask(), 1000, 1000);

	System.out.println("Server Endpoint Initialized!");
}
```

Im Konstruktor der Endpoint Klasse wird ein Timer gestartet. Dieser führt anschließend im Sekundenintervall den TimerTask aus.

6.3 Die fertige Anwendung

Die Anwendung wird nun auf einem lokalen Glassfish Server „deployed“. Anschließend kann von einem Browser aus über die URI ‚`ws://localhost:8080/WebsocketDemo`‘ auf die Anwendung Zugegriffen werden.

Das folgende Screenshot zeigt die Anwendung:



Im oberen Bereich wird die aktuelle Zeit des Servers angezeigt. Im mittleren Bereich kann eine Nachricht an den Server versendet werden. Der untere Bereich ist eine Logausgabe, welche verschiedene Events wie den Aufbau der Verbindung oder das Empfangen einer Nachricht anzeigt.

Anschließend ist hier noch ein Screenshot der Server Ausgabe:

```
GlassFish 4 at localhost [domain1]
2016-05-05T17:14:03.563+0200|Information: Connection closed on Session: 930b5a7e-fd78-4ce2-815a-eb85b2da4d6e
2016-05-05T17:36:52.872+0200|Information: Server Endpoint Initialized!
2016-05-05T17:36:52.872+0200|Information: Created new Connection with Session ID: 6c005905-3e8d-49e0-b11e-e3be8bb52c7b
2016-05-05T17:38:39.569+0200|Information: Connection closed on Session: 6c005905-3e8d-49e0-b11e-e3be8bb52c7b
2016-05-05T17:38:39.586+0200|Information: Server Endpoint Initialized!
2016-05-05T17:38:39.586+0200|Information: Created new Connection with Session ID: 53441bee-88df-4756-b139-1a4de2047a8f
2016-05-05T17:42:11.974+0200|Information: Message received from Session ID: 53441bee-88df-4756-b139-1a4de2047a8f: Hallo Welt
```

Hier sieht man die Serverseitigen Events, wie das Erzeugen einer neuen Endpoint Instanz oder das Empfangen einer Nachricht von einer Session.

7. Fazit

Das Websocket Protokoll ist ein sehr interessanter Ansatz zur Realisierung einer bidirektionalen Kommunikation zwischen Browser und Webserver. Im Vergleich zu den HTTP Methode Polling und Long Polling erzeugt es kaum Overhead auf der Verbindung. Außerdem werden die Clients sofort über ein serverseitiges Event benachrichtigt und es besteht keine Gefahr, dass wie beim Long Polling ein Event verloren gehen kann.

Durch die Verwendung einer IDE wie Eclipse EE ist auch die Entwicklung einer Java Webanwendung mit der JSR 356 API sehr einfach. Dadurch ist eine breitere Masse von Softwareentwicklern in der Lage, eine solche Anwendung zu entwickeln. Denn wenn die Entwicklung ein sehr komplizierter Prozess wäre, könnten nur wenige Entwickler mit sehr vielen Kenntnissen über das Thema solche Anwendungen entwickeln.

Aus diesen Gründen sehe ich ein großes Potenzial in dem Websocket Protokoll.

8. Quellen:

https://docs.oracle.com/javaee/7/tutorial/we

https://jcp.org/en/jsr/detail?id=356

https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/images/HTTP.png

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

https://tools.ietf.org/html/rfc6455#section-1.5

http://blogs.microsoft.co.il/blogs/levi_moshe/FrameFormat_0D8DBC72.png

http://www.xoriant.com/blog/wp-content/uploads/2010/10/WebSocket-Architecture.jpg

https://blog.idrsolutions.com/2013/12/websockets-an-introduction/