Vorstellung der Studienarbeit zum Thema Websockets

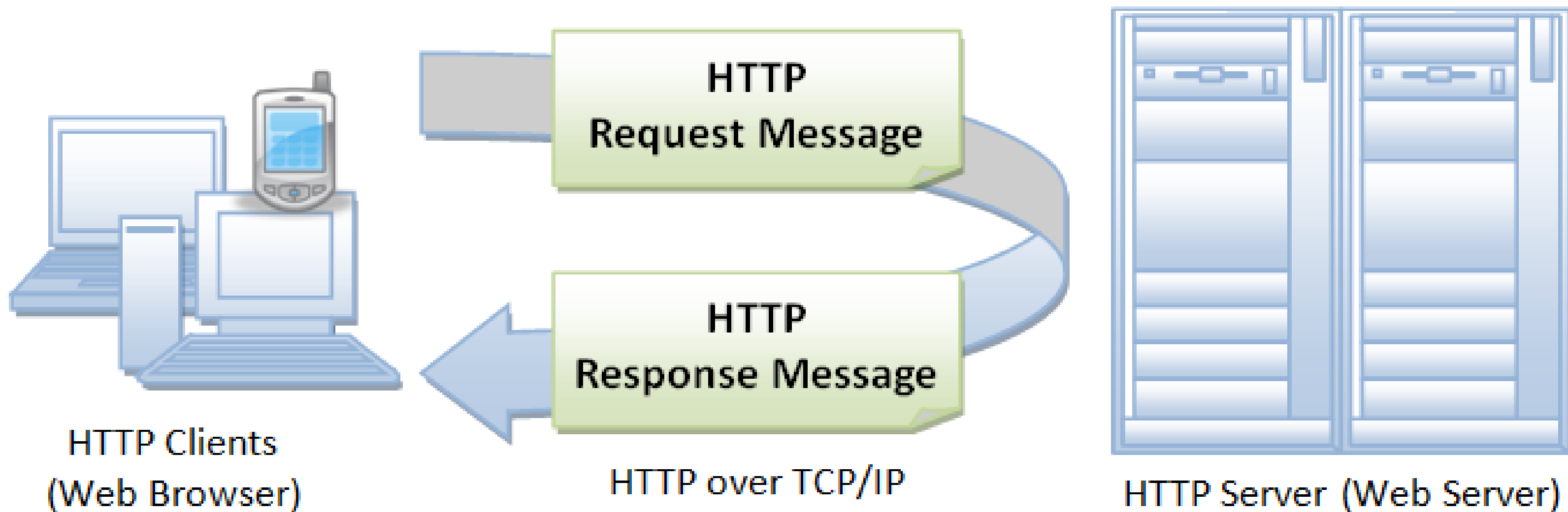
Christian Linha. IF6

HOCHSCHULE
FÜR ANGEWANDTE
WISSENSCHAFTEN
MÜNCHEN

Warum Websockets?

Das Internet wurde um das Request – Response Muster des HTTP Protokolls entwickelt:

- Client(Browser) sendet Anfrage an Server
- Anfrage in form eines HTTP Requests
- Server antwortet mit HTTP Response



Das Problem

- Von Webanwendungen wird immer mehr Dynamik verlangt

- HTTP Protokoll kommt an die Grenzen

Der Grund

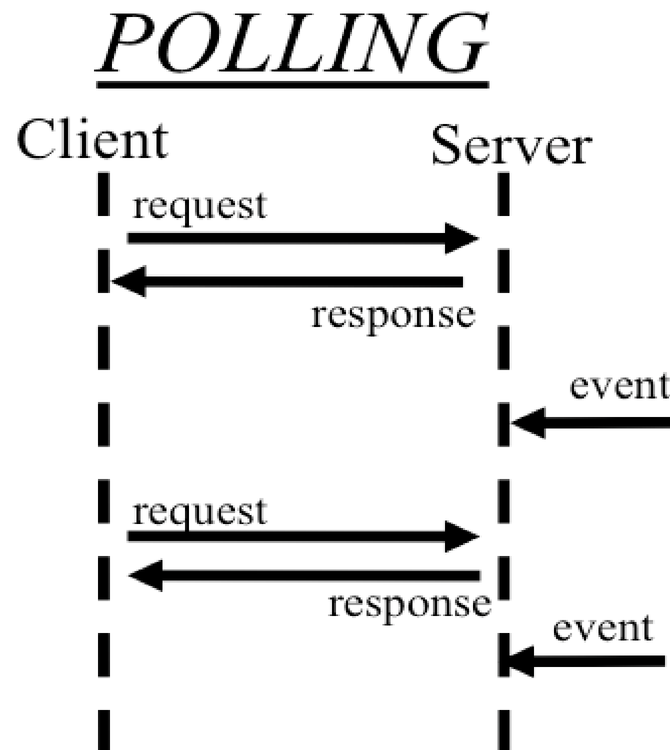
- Ein Datenaustausch wird immer von einem Client angetriggert
- Der Server kann Datenaustausch nur über Umwege auslösen
- Datenübertragung von Server zu Client wird Server-Push genannt

Folgende Möglichkeiten sehen dem Server zu Verfügung:

- Polling
- Long Polling

Polling

- Client sendet periodische Anfragen an Server
- Falls auf dem Server ein Event stattfindet werden Daten in dem Response gesendet
- Ansonsten wird ein leerer Response gesendet

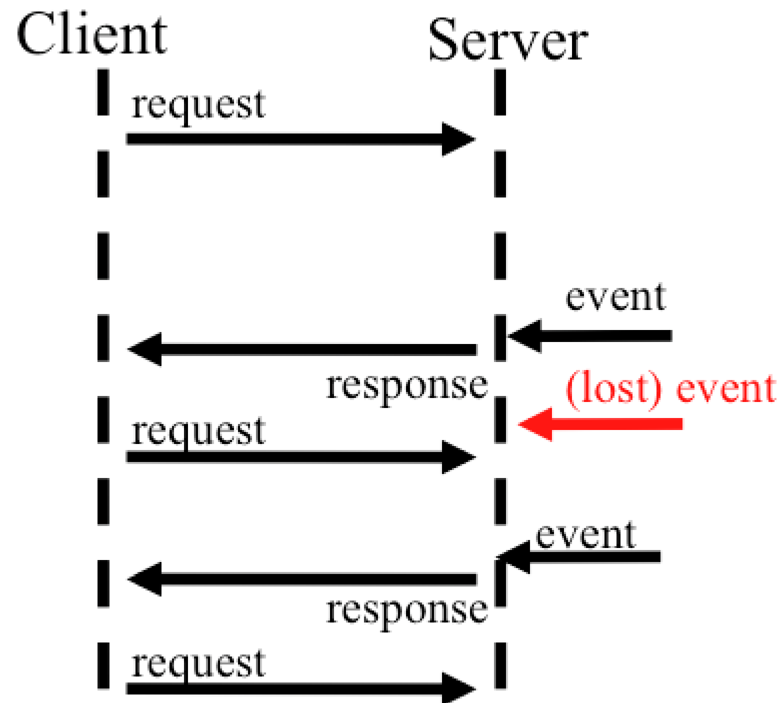


Nachteile des Pollings

- Client muss dauerhaft Anfragen senden
- Server muss dauerhaft Anfragen bearbeiten
- Verschwendet Bandbreite auf der Verbindung

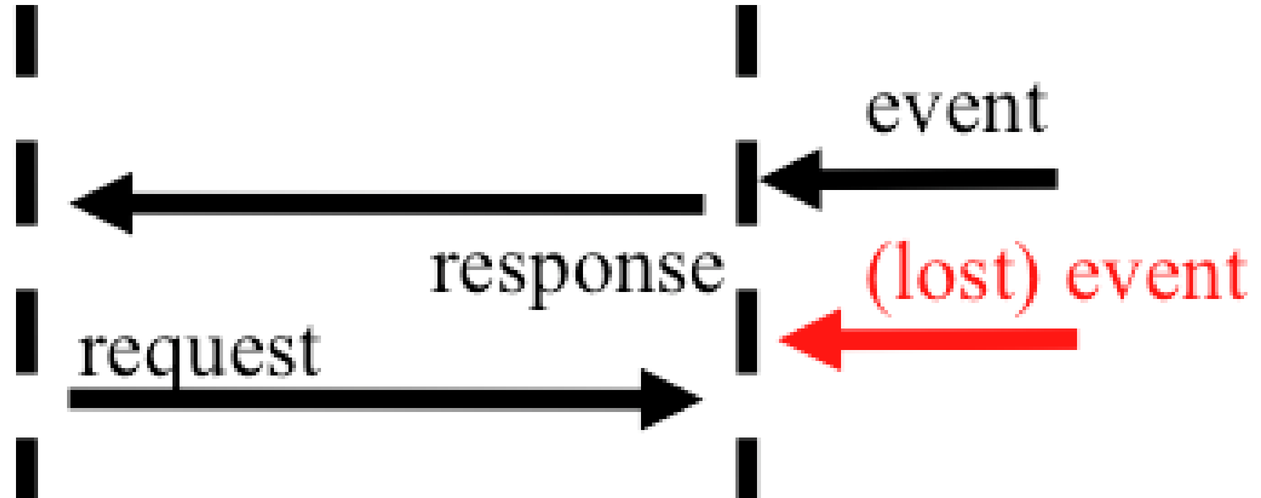
Long Polling

- Client sendet Request an Server
- Server antwortet erst wenn ein Event auf dem Server auftritt



Nachteile des Long Pollings

- Events auf dem Server können verloren gehen
- Kann passieren wenn ein Event zwischen dem Response des Servers und einem erneuten Request des Clients auftritt



Die Lösung

- Besser wäre eine bidirektionale Verbindung zwischen Server und Client

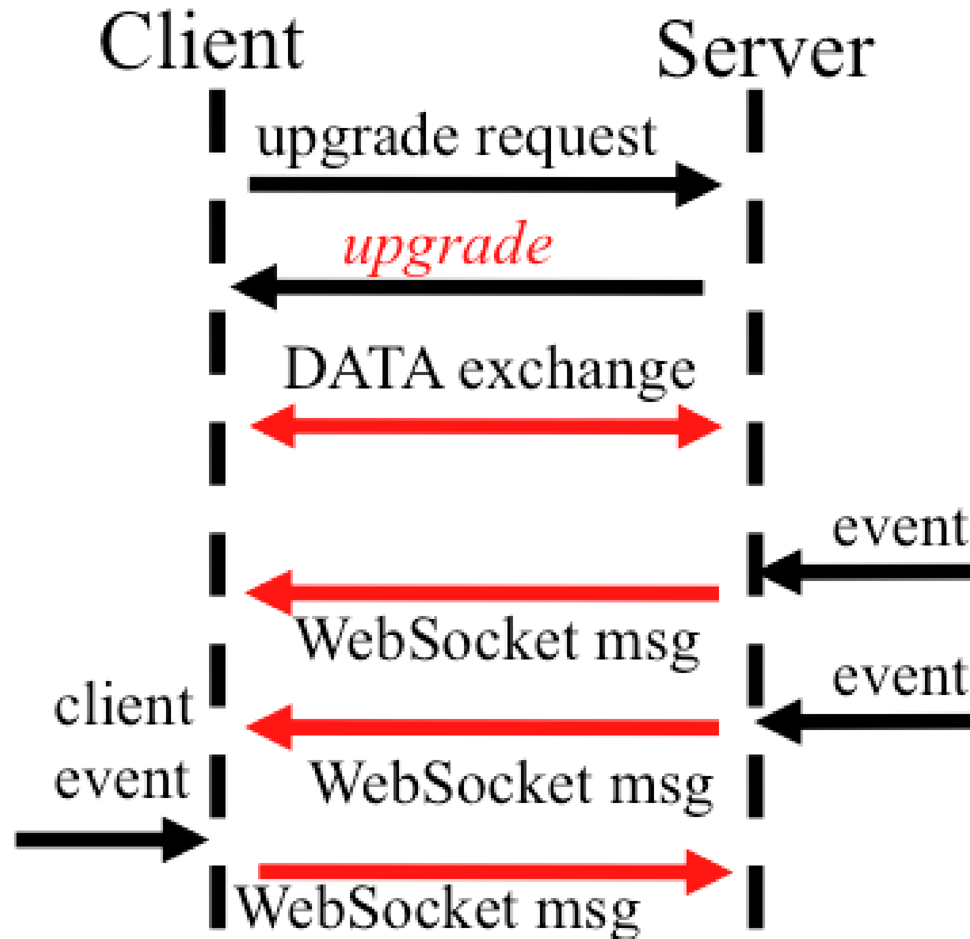
- Client und Server können Nachrichten versenden und gleichzeitig empfangen

Das Websocket Protokoll

- Implementiert eine Bidirektionale Kommunikation zwischen zwei Endpunkten

- Ist im HTML 5 Standard enthalten

Das Websocket Protokoll



Das Protokoll im Detail

Verbindungsaufbau

- Verbindung wird über HTTP Protokoll aufgebaut
- Port ist 80 bzw. 443 bei Verschlüsselung

Das Websocket Protokoll definiert 2 neue URI Schemas:

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

host: Domainname bzw. IP des Servers
port: Port über den die Verbindung aufgebaut wird
path: Den Pfad des Server Endpunktes
query: Optionale Parameter

Handshake

Bei Aufbau einer Verbindung findet zuerst ein Handshake statt:

1. Client sendet HTTP Anfrage an Server:

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

- Mit den Feldern ‚Connection' und ‚Upgrade' fordert der Client ein Upgrade auf das Websocket Protokoll
- Sec-WebSocket-Key ist ein zufällig generierter Wert
- Sec-WebSocket-Protocol gibt die Version des Protokolles an

Handshake

2. Der Server antwortet auf Client Anfrage:

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

- Mit den Feldern ‚Connection' und ‚Upgrade' bestätigt der Server den Wechsel

Handshake

Der Wert von ‚Sec-WebSocket-Accept' wird mit folgenden Schritten generiert:

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

Handshake

- Der Wert von ‚ Sec-WebSocket-Accept' ist zur Überprüfung für den Client

- Die Schritte 2-4 werden auch auf dem Client ausgeführt

- Falls der vom Client berechnete Wert mit dem vom Server empfangenen Wert übereinstimmt,
  weiß der Client, dass die Anfrage bearbeitet wurde

- Der Handshake ist bei Übereinstimmung abgeschlossen und die Verbindung läuft ab jetzt über das Websocket Protokoll

Datenübertragung

- Daten Werden in Form von Messages übertragen
- Message wird auf einen oder mehrere Frames aufgeteilt
- Frames werden sequentiell über darunter liegende TCP Verbindung übertragen
- Empfänger baut aus empfangenen Frames die Message zusammen

Aufbau eines Frames

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th><th>14</th><th>15</th></tr>
<tr><td>0</td><td>Fin</td><td>Rsv1</td><td>Rsv2</td><td>Rsv3</td><td colspan="4">Opcode</td><td>Mask</td><td colspan="7">Payload Length</td></tr>
<tr><td>1</td><td colspan="16">Extended Payload Length (if Payload Length == 126)</td></tr>
<tr><td>2</td><td colspan="16" rowspan="3">Extended Payload Length (if Payload Length == 127)</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td><td colspan="16" rowspan="2">Masking Key (if Mask == 1)</td></tr>
<tr><td>6</td></tr>
<tr><td>7</td><td colspan="16">Payload Data ....</td></tr>
</table>

Zu Beachten:
Die Nummerierung steht für die einzelnen Bits und nicht Bytes

Aufbau eines Frames

Bit 0: FIN

Signalisiert, ob es sich um den letzten Frame der Nachricht handelt.

Bit 1 – 3: REV

Reserviert für zukünftiges Nutzen.

Bits 4 – 7: OPCODE

OP Code der Nachricht. Signalisiert, wie die Nachricht interpretiert werden soll.

Bit 8: MASK

Gibt an, ob die Payload Data der Nachricht maskiert wird.

Aufbau eines Frames

Bit 9 – 15: PAYLOAD LENGTH

Enthalten die Länge der Payload in Bytes.

16 - 79: EXTENDED PAYLOAD LENGTH

Zusätzliche Längenbits.

Bit 80 – 111: MASK KEY

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

Ab Bit 112: PAYLOAD

Nutzdaten der Message.

Client Payload-Data Maskierung

- Laut Spezifikation des Websocket Protokolles muss jede von Client gesendete Nachricht maskiert werden

- Maskierung ist Transformation der Payload mit dem Mask-Key

- Der Mask-Key wird vom Client zufällig generiert

Client Payload-Data Maskierung Algorithmus

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

1. Errechne den Wert j aus dem Index des aktuellen Bytes Modulo 4

2. Führe eine XOR Verknüpfung mit dem aktuellen Byte und dem Byte an der Stelle j im Mask-Key durch

3. Das Ergebnis ist das Byte an der Stelle mit demselben Index in den transformierten Daten

Client Payload-Data Maskierung Algorithmus in Java

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

// mask key
byte [] key;

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

//initialize payload and mask key
...

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

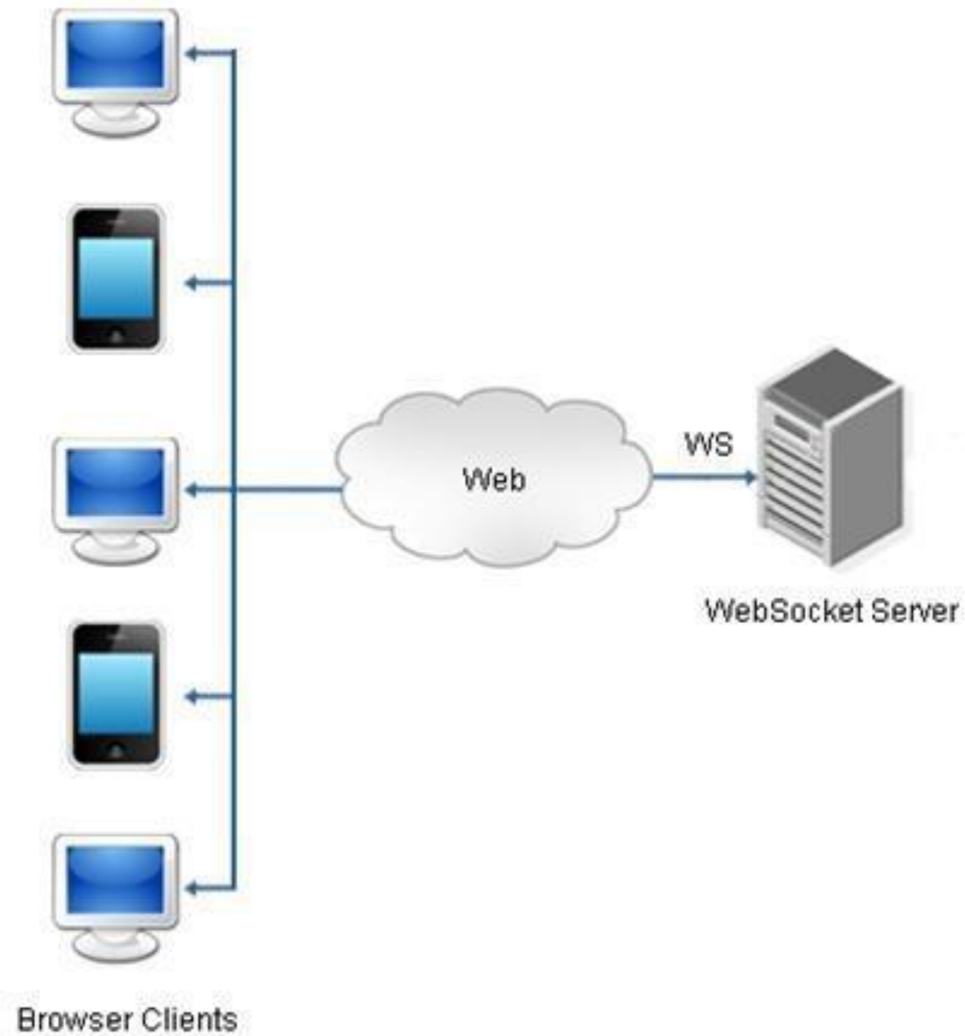
Websockets in der Praxis

Websockets in der Praxis

- Fokus auf Webanwendungen
- Client ist Webbrowser
- Serveranwendung läuft auf Webserver

- Client Implementierung in Javascript
- Server Implementierung in „richtiger“ Programmiersprache (Java, C#, etc.)

Websockets in der Praxis



Das Protokoll aus der Sicht eines Entwicklers

Programmieren mit Websockets

- Alle gängigen Sprachen haben Libraries, welche das Websocket Protokoll implementieren (Javascript, Java, C#, etc)

- Programmierkonzept ist unabhängig von der Sprache

Programmieren mit Websockets

Gewöhnliche HTTP Kommunikation über Sockets:

- Datenaustausch ist synchron

- Client sendet Anfrage, Server antwortet

- Client wird immer eine Antwort vom Server bekommen; vorausgesetzt der Server ist erreichbar

- Deswegen kann Client direkt nach Anfrage auf die Antwort warten

- Kommunikation lässt sich in einem Thread realisieren

Programmieren mit Websockets

Kommunikation über Websockets:

- Datenaustausch ist asynchron

- Beide Endpunkte können zu jedem Zeitpunkt eine Nachricht empfangen

- Ereignisverabreitung wird über Handlerfunktionen realisiert

- Wenn Ereignis auftritt wird der entsprechende Handler aufgerufen

- Beide Seiten implementieren Handler

Websocket Handlerfunktionen

In der Regel gibt es 4 verschiedene Handlerfunktionen:

onOpen:
Wird aufgerufen, wenn die Verbindung aufgebaut wurde.

onClose:
Wird aufgerufen, wenn die Verbindung geschlossen wird.

onError:
Wird aufgerufen, wenn ein Fehler in der Verbindung auftritt.

onMessage:
Wird aufgerufen, wenn eine Nachricht empfangen wird.

Veranschaulichung der Websocket Programmierung an einem Programmbeispiel

Programmbeispiel: Serverseitige Uhr

- Zeigt Uhrzeit des Servers im Client an
- Java Webanwendung auf Webserver
- Javascript auf Clientseite
- Java auf Serverseite
- Kommunikation über Websocket Protokoll

Der clientseitige Teil

- Verbindung wird über in HTML eingebettetes Javascript aufgebaut
- Als erstes muss der Pfad zu Serveranwendung definiert werden

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

- Anwendung läuft auf lokalem Rechner auf Port 8080
- Pfad der Anwendung ist ‚/WebsocketDemo/Clock'

Der clientseitige Teil

- Definition der Handler onOpen und onClose:

```
//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");
}
```

Der clientseitige Teil

- Definition der Handler onMessage und onError:

```
//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");
}
```

Der clientseitige Teil

- Eine Websocket Verbindung kann mit folgenden Befehlen aufgebaut werden

```
//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) };
```

- Dem Konstruktor des Websocket Objektes wird die URI zum Server übergeben
- Anschließend werden dem Objekt die Handler initialisiert

Der clientseitige Teil

- Mit der Methode ‚send(value)' des Websocket Objektes können Nachrichten an den Server versendet werden

- Versenden von Nachrichten ist in Methode ‚doSend' implementiert

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

Der clientseitige Teil

- Zur Anzeige der Zeit auf dem Bildschirm wurde eine weiter Methode definiert

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

Der clientseitige Teil

- Zusätzlich wurde eine Methode zur Logausgabe definiert

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

Der Serverseitige Teil

- Läuft als Java Webanwendung auf lokalem Glassfish Server
- Nutzt Java Websocket API (JSR 356)
- Stellt Websocket Server Endpoint für Client zur Verfügung
- Endpoint sendet Zeit des Servers an Client

Der Serverseitige Teil

- Als erstes wird eine Klasse für den Endpoint definiert

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

		Timer clockTimer;
}
```

- Mit der Annotation wird die Klasse als Server Endpoint auf dem Pfad ‚/Clock' deklariert

- Auf dem Endpoint läuft Timer der im Sekundentakt die Zeit des Servers an Client sendet

Der Serverseitige Teil

- Auch der Server muss die 4 Handler implementieren

- In Java werden Handler als Methoden in der Endpoint Klasse definiert und durch Annotationen als Handler festgelegt

Der Serverseitige Teil

Der Handler OnOpen:

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

          doSend("Connected!", session);

  }
```

- In der Endpoint instanz wird bei Verbindungsaufbau das Session Ojekt der Verbindung gespeichert

Der Serverseitige Teil

Der Handler OnMessage:

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

- Empfangene Textnachrichten werden auf der Logausgabe ausgegeben

Der Serverseitige Teil

Der Handler OnClose:

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

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

- Bei Abbau der Verbindung wird Timer abgebrochen

Der Serverseitige Teil

Der Handler OnError:

```
@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());
  }
```

- Bei Verbindungsfehler wird die Verbindung geschlossen

Der Serverseitige Teil

- Mit folgender Methode können Nachrichten an den Client gesendet werden:

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

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

Der Serverseitige Teil

- Für den Timer wird ein Timer Task definiert:

```
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);
		}
}
```

- Timer Task erzeugt String der aktuellen Zeit im Format HH:mm:ss

Der Serverseitige Teil

- Der Timer Task wird durch eine Timer im 1s Intervall aufgerufen
- Der Timer wird im Konstruktor des Endpoints initialisiert

```
public ClockServerEndpoint()
{

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

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

Die fertige Anwendung

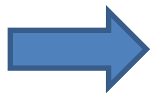
- Wird auf lokalem Glassfish Server Deployed
- Ist in Browser über folgender URI ansprechbar:

http://localhost:8080/WebsocketDemo

Fazit

Vorteile:

- Hochdynamische Kommunikation zwischen Client und Server
- Kaum Overhead
- Websocket Anwendungen leicht zu implementieren



Websocket Protokoll hat Potenzial für die Zukunft

Quellen

Protokoll Informationen:

- http://www.heise.de/developer/artikel/WebSocket-Annaeherung-an-Echtzeit-im-Web-1260189.html
- https://tools.ietf.org/html/rfc6455#section-1.5

API Informationen:

- https://docs.oracle.com/javaee/7/tutorial/we
- https://jcp.org/en/jsr/detail?id=356
- https://blog.idrsolutions.com/2013/12/websockets-an-introduction/

Bilder:

- http://blogs.microsoft.co.il/blogs/levi_moshe/FrameFormat_0D8DBC72.png
- https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/images/HTTP.png
- http://www.xoriant.com/blog/wp-content/uploads/2010/10/WebSocket-Architecture.jpg
- http://www.heise.de/developer/artikel/WebSocket-Annaeherung-an-Echtzeit-im-Web-1260189.html