Möglichkeiten zur asynchronen Kommunikation zwischen Webbrowser und Server

Hendrik Wagner
hendrik.wagner@mni.thm.de
Technische Hochschule Mittelhessen
Gießen, Hessen, Germany

Zusammenfassung

Die Kommunikation zwischen Webbrowser und Server wird klassischerweise durch den Client (in der Regel ein Webbrowser) initiiert. Dieser fragt den Inhalt einer Website an, und kann später weitere Inhalte laden oder Daten übermitteln. Gemäß der HTTP-Spezifikation kann der Server aber keine Daten nachträglich an den Client senden, wenn dieser diese nicht aktiv anfragt. In diesem Artikel werden verschiedene Möglichkeiten zur fortlaufenden Kommunikation zwischen Client und Server vorgestellt und analysiert. Dabei werden die Vor- und Nachteile der einzelnen Methoden aufgezeigt.

Keywords: web, http, WebSockets, polling, bidirectional communication, asynchronous communication

Inhaltsverzeichnis

1 Einleitung

Moderne Webapplikationen sind in der Regel auf eine bidirektionale Kommunikation zwischen Client und Server angewiesen, um zum Beispiel Benachrichtigungen oder Echtzeitdaten anzuzeigen. Webseiten basieren in der Regel auf dem HTTP-Protokoll, nach welchem alle Anfragen vom Client initiiert werden müssen. Um also eine asynchrone Kommunikation zwischen Client und Server zu ermöglichen, muss entschieden werden, wie mit dieser Einschränkung umgegangen werden soll. Dieser Artikel beschäftigt sich mit verschiedenen Ansätzen zur asynchronen Kommunikation zwischen Client und Server.

2 Vorstellung der Kommunikationsvarianten

Es gibt eine Reihe von Ansätzen, wie eine bidirektionale Kommunikation zwischen Client und Server im Web realisiert werden kann. In diesem Abschnitt werden die wichtigsten Ansätze chronologisch vorgestellt und kurz erläutert.

2.1 Klassisches Polling

Unter dem 1990 eingeführten [1, Abs. 1.2] HTTP-Protokoll gibt es eine Reihe von Anfragen, die der Client an den Server senden kann [2]. Das Protokoll definiert dabei explizit einen Client (in unserem Fall der Browser), welcher Anfragen an den Server (Bereitstellender der Webinhalte) versendet. Dieser wartet auf Anfragen und beantwortet diese [1, Abs. 1.3].

Für das Vorhaben sind GET-Requests besonders relevant, also klassische Abfragen von Inhalten mittels HTTP. Polling, in diesem Anwendungsfall wohl am besten übersetzt mit *zyklische Absuche* oder *Sendeaufruf*, beschreibt in der Webentwicklung das regelmäßige Abrufen (in einem Intervall Δt) von neuen Inhalten. Abhängig von der Implementierung gilt das Intervall entweder ab dem Zeitpunkt des letzten Abrufs oder ab dem Zeitpunkt der letzten Antwort des Servers. Gibt es Nachrichten, die der Server bereitstellen möchte, beantwortet dieser die Anfrage mit den neuen Inhalten – andernfalls wird der Antwortkörper leer sein.

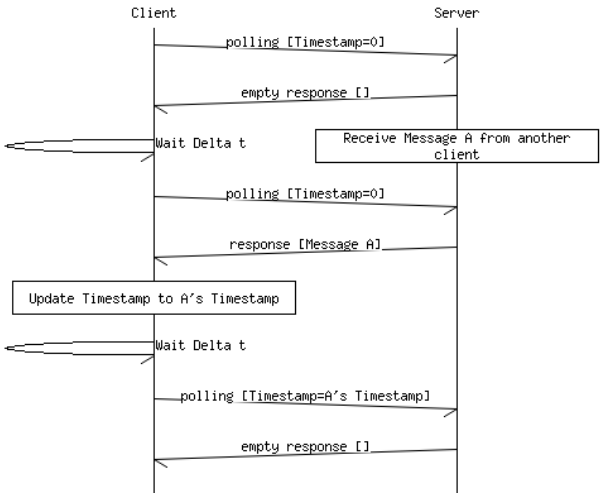

Abbildung 1. Beispiel von Nachrichtenempfang bei einer klassischen Polling-Kommunikation. Der Client sendet regelmäßig eine Anfrage an den Server, welche entweder mit leerer Antwort oder mit neuen Inhalten beantwortet wird. Die Länge der Zeitspanne Δt zwischen zwei Polling-Anfragen ist implementierungsabhängig.

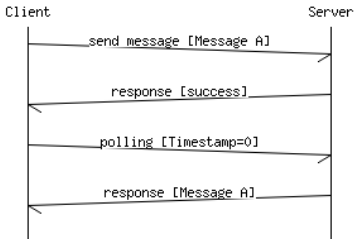

Abbildung 2. Beispiel von Nachrichtenversand bei einer klassischen Polling-Kommunikation. Der Client sendet eine Nachricht an den Server, welche daraufhin mittels polling abgerufen werden kann.

Das Hauptproblem mit Polling ist der entstehende Header Overhead, der für den gesamten Zeitraum in den regelmäßigen Abfragen besteht. Dadurch entsteht eine Netzwerkbelastung, welche keine tatsächlichen Informationen übermittelt. In der in späteren Abschnitten vorgestellten Chatanwendung handelt es sich so zum Beispiel um 156 Bytes, die sekündlich vom Server übermittelt werden, aber lediglich ein leeres JSON-Array enthalten.

2.1.1 Spezifikation. Polling ist in der Spezifikation von HTTP nicht definiert, es handelt sich hierbei um eine Implementierung des Clients und des Servers. Meist wird Polling durch JavaScript-Code realisiert, welcher in einem bestimmten Intervall (z. B. alle 5 Sekunden) eine Anfrage an den Server sendet. Diese Anfrage ist ein GET-Request, welcher die URL der Anwendung enthält. Wie der Server auf diese Anfrage antwortet, ist Implementierungsabhängig: So könnte er zum Beispiel den gesamten abgefragten Inhalt bei jeder Anfrage übermitteln, oder nur Änderungen, die dieser Client noch nicht übermittelt bekommen hat. Eine solche Optimierung setzt allerdings voraus, dass der Server in der Lage ist, zu identifizieren, welche Änderungen der Client bereits erhalten hat.

2.2 Long Polling

Eine Optimierung des Polling-Konzepts ist das sog. *Long Polling*. Dabei wird die Antwort auf eine HTTP-Anfrage zurückgehalten, bis der Server eine Nachricht versenden will [3].

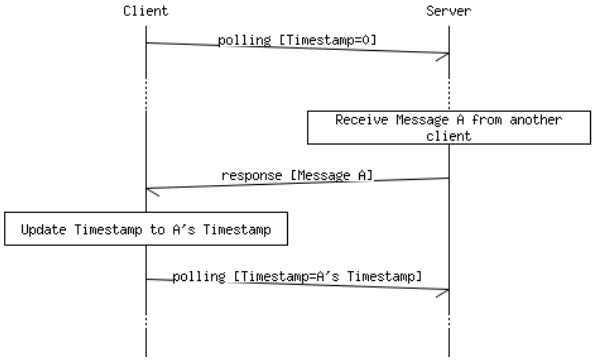

Abbildung 3. Beispiel von Nachrichtenempfang bei Long Polling. Der Server beantwortet die Anfrage des Clients erst, wenn er eine Nachricht hat, die er übermitteln möchte. Der Nachrichtenversand verläuft analog zu Polling.

Probleme von Long Polling sind unter anderem der (im Vergleich zu klassischem Polling reduzierter, aber weiterhin präsenter) Header Overhead, mögliche Timeouts und Ressourcen, die in Vorbereitung auf eine eingehende Nachricht vom Betriebssystem zu Verfügung gestellt werden. [4, Abs. 2.2].

2.3 Streaming

Eine weitere Optimierung des Polling-Konzepts ist das sog. *Streaming*, welches die HTTP Transferkodierung Chunking (vgl. [1, Abs. 7.1]) verwendet, also dem Aufspalten der Antwort in mehrere Packets, in welchen dann einzelne Nachrichten versendet werden. So kann die Anzahl an Anfragen ausgehend vom Client an den Server auf eine reduziert werden – der Server terminiert nie die Antwort auf die erste Anfrage [4, Abs. 3].

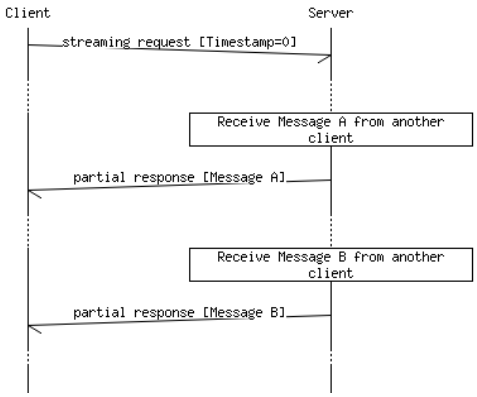

Abbildung 4. Beispiel von Nachrichtenempfang bei Streaming. Der Client initiiert den Streaming-Prozess durch eine einmalige Anfrage. Daraufhin sendet der Server Teilantworten an den Client, sobald neue Nachrichten verfügbar sind. Der Nachrichtenversand verläuft analog zu Polling.

Mit Streaming werden einige Nachteile von Polling beglichen, andere – insb. mögliche Timeouts und unnötig reservierte Ressourcen – bleiben bestehen. Zusätzlich besteht das Risiko, dass diese Form von Kommunikation nicht auf allen Systemen funktioniert – Proxys können Pakete bündeln und erst verzögert weiterleiten, wodurch Pakete ggf. nicht zeitgetreu, gebündelt oder aufgespalten ankommen [4, Abs. 3.2].

2.4 WebSockets

WebSockets sind ein vom IETF[1] entwickeltes Protokoll, welches 2011 als Standard veröffentlicht wurde [5]. Grund für dessen Entstehung ist unter anderem die Erkenntnis, dass der Versuch HTTP zu verwenden, um bidirektionale Kommunikation zu ermöglichen, vermeidbare Komplexität und Ineffizienz mit sich bringt [6, S. 137f]. Es ist anzumerken, dass die vorgestellten Polling-Varianten das HTTP-Protokoll effektiv missbrauchen, um serverseitige Kommunikation zu ermöglichen:

- Polling beinhaltet das Versenden von redundanten Anfragen, welche ohne Inhalt beantwortet werden,
- Long Polling simuliert eine Verbindung mit (sehr) hoher Latenz um eine Antwort herauszuzögern und
- Streaming verwendet HTTP Chunking, um innerhalb einer Response alleinstehende Nachrichten zu senden.

Diesen Missbrauch haben WebSockets nicht – es handelt sich hierbei um ein komplett neues Protokoll, welches diese Problemstellung direkt adressiert (vgl. [5, Abs. 1.1]), indem es eine TCP-Verbindung zwischen Client und Server aufbaut und unterstützt. Genauer baut WebSockets einen Tunnel zwischen TCP und IP auf, sodass darauffolgend TCP-Kommunikation Systemübergreifend erfolgen kann (vgl. [5, Abs. 1.5]). HTTP wird dann lediglich für die initialen Handshakes verwendet – danach nicht mehr.

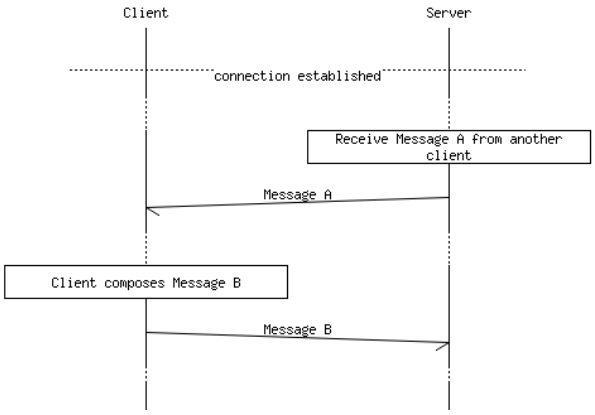

Abbildung 5. Beispiel von Nachrichtenaustausch bei WebSockets. Nach dem Handshake wird eine TCP-Verbindung zwischen Client und Server aufgebaut. Von nun an können Nachrichten in beide Richtungen ausgetauscht werden.

2.4.1 Spezifikation.

Um eine Verbindung über einen WebSocker aufzubauen, wird zunächst vom Client aus eine HTTP-Anfrage gesendet. Gemäß der Spezifikation muss es sich hierbei um eine GET-Request an den Pfad, auf dem der WebSocket hinterlegt ist, handeln.

Wäre die WebSocket URI `ws://example.org/chat`, so wäre die erste Zeile der HTTP-Anfrage `GET /chat HTTP/1.1` [5, Abs. 4.1]. Zusätzlich müssen in der Anfrage unter anderem der Host (hier `Host: example.org`) und die Intention (Felder `Connection: Upgrade` und `Upgrade: websocket`) als Headerfelder übermittelt werden.

Der Server antwortet auf diese Anfrage mit einer HTTP-Response, welche entweder den Statuscode 101 *Switching Protocols* oder einen Fehlercode enthält. Zudem meldet er bei erfolgreichem Protokollwechsel die vom Client angegebene Intention zurück.

Wurde die Verbindung vom Server akzeptiert, können nun Client und Server direkt miteinander kommunizieren. Für die Datenübermittlung sieht das WebSocket Protokoll eine auch in anderen Protokollen ähnlich vertretenes Frame-System vor, also dem Versenden von Daten in Form von Frames, welche jeweils einen Header und eine Payload enthalten. Header enthalten u. A. den Opcode[2] und die Payload-Länge (vgl. [5, Abs. 5.2]).

[1] Internet Engineering Task Force.

[2] Opcodes (*Operation Codes*, dt. *Befehlscode*) können bei WebSockets Frames mit Text- oder Binärdaten ankündigen, eine vorherige Frame fortsetzen, die Verbindung schließen, oder einen Ping/Pong markieren.

3 Beispielhafte Implementierungen

Um die Unterschiede zwischen den Kommunikationsvarianten ersichtlich zu machen, werden diese für eine Chat-Applikation implementiert. Dabei wird jeweils auf die Client- und Serverseitige Implementierung eingegangen.

Die Realisierung erfolgt in TypeScript, wobei die Serverseitige Implementierung auf Node.js basiert. Der Client benutzt das Vue.js-Framework mit Vuetify, um die Darstellung zu vereinfachen. Implementierungen in anderen Frontend-Frameworks erfolgen analog. Es wird nicht auf die Darstellung der Nachrichten oder des Chats eingegangen, da dies nicht relevant für die Implementierung ist. Entsprechende Codeabschnitte sind daher ausgelassen.

Die vollständige Implementierung der Frontendkomponenten ist im Repository im Verzeichnis [7, `code/client/src/components/`] zu finden. Der gesamte Code für alle Serverkomponenten befindet sich in [7, `code/server/chatserver.ts`].

3.1 Implementierung von Polling

Für alle Arten von Polling verwaltet der Server eine Liste aller bisher empfangenen Nachrichten. Neue von einem Client übermittelte Nachrichten werden mit Autor, Zeitstempel und Text in diese Liste eingefügt.

Im Falle des klassischen Polling wird der Client alle paar Sekunden eine Anfrage an den Server senden, um neue Nachrichten abzufragen. Dementsprechend muss der Server seine Nachrichtenliste nach für den Client neuen Nachrichten filtern und diese zurückgeben.

Listing 1. Beispielhafte Nachricht

```
{
  content: "Hello World!",
  sender: "Alice",
  timestamp: "2022-12-04T19:29:13.455Z"
}
```

3.1.1 Polling Server.

Der Server besteht aus einer Instanz von `http.Server`, welche die HTTP-Anfragen `/polling` und `/polling/send` verarbeitet. Zudem verwaltet er ein Array mit allen Nachrichten, die bisher gesendet wurden.

Bei einer Anfrage an `/polling` wird mit einem Zeitstempel gearbeitet, sodass der Server nur die Nachrichten zurückgibt, die nach diesem Zeitstempel eingegangen sind.

Listing 2. Polling Server: Nachrichten abfragen

```
31  if (req.url === "/polling") {
32    // we get a Buffer that contains our timestamp
33    let timestamp = "";
34    req.on("data", (chunk) => {
35      timestamp += chunk; // convert Buffer to string
36    });
37    req.on("end", () => {
38      res.setHeader("Content-Type", "application/json");
39      res.end(
40        JSON.stringify([...filterMessages(timestamp,
      lastPollingMessages)])
41      ); // return messages as json
42    });
```

Im Falle einer Anfrage an `/polling/send` wird die Nachricht in das Array eingefügt und der Client mit dem Statuscode 200 *OK* bestätigt.

Listing 3. Polling Server: Nachrichten abfragen

```
43  } else if (req.url === "/polling/send") {
44    let body = "";
45    req.on("data", (chunk) => {
46      body += chunk;
47    });
48    req.on("end", () => {
49      const message: Message = JSON.parse(body);
50      logMessage(message, "POLLING");
51      lastPollingMessages.push(message);
52      res.end(); // send 200 OK
53    });
```

3.1.2 Polling Client.

Der Client besteht aus der Komponente `PollingChat.vue`, welche die Funktionen zum Senden und Empfangen von Nachrichten enthält. Die Darstellung, sowie die Speicherung von Nachrichten und Nutzdaten erfolgt in der Komponente `ChatPage.vue` (vgl. jeweils [7, `code/client/src/components/`]).

Um eine Nachricht zu versenden, wird lediglich eine HTTP POST-Anfrage an die URL `/polling/send` gesendet. Die Nachricht wird dabei als JSON-Objekt in der Anfrage übermittelt.

Listing 4. Polling Client: Nachrichten abfragen

```
14 const sendMessage = async (msg: Message) => {
15   await fetch("http://localhost:8082/polling/send", {
16     method: "POST",
17     headers: {
18       "Content-Type": "application/json",
19     },
20     body: JSON.stringify(msg),
21   });
22 };
```

Mittels einer HTTP POST-Anfrage an die URL `/polling` werden die Nachrichten abgefragt. Die Anfrage enthält dabei den Zeitstempel der letzten Nachricht, die der Client erhalten hat, als String. Der Server antwortet mit einer Liste von Nachrichten, die nach diesem Zeitstempel eingegangen sind. Diese werden anschließend an die Komponente `ChatPage.vue` mittels des Events `appendMessages` übermittelt.

Um die Nachrichten regelmäßig abzufragen, wird ein Intervall gesetzt, welches alle zwei Sekunden die Funktion `fetchMessages` aufruft. Dieses Intervall wird bei der Initialisierung der Komponente gesetzt und bei der Entfernung der Komponente wieder gelöscht.

Listing 5. Polling Client: Nachrichten abfragen

```
24 const fetchMessages = async () => {
25   const response = await fetch("http://localhost:8082/
       polling", {
26     method: "POST",
27     headers: {
28       "Content-Type": "application/json",
29     },
30     body: lastMessageTimestamp, // send the last message
        timestamp to the server
31   });
32
33   if (response.ok) {
34     const newMessages = await response.json();
35     emit("appendMessages", newMessages); // append the new
        messages to the chat
36
37     // update the last message timestamp to the timestamp of
         the last message, if there are any
38     if (newMessages.length > 0)
39       lastMessageTimestamp = newMessages[newMessages.length
       - 1].timestamp;
40   }
41 };
42
43 // we fetch new messages every 2 seconds, regardless of how
       long the previous request took
44 setInterval(fetchMessages, 2000);
```

3.2 Implementierung von Long Polling

Die Implementierung von Long Polling ist sehr ähnlich zu der von Polling. Die einzige Änderung ist, dass der Server die Anfrage nicht sofort beendet, sondern die Verbindung offen hält, bis eine neue Nachricht eingegangen ist. Dazu wird sowohl der Client als auch der Server angepasst.

3.2.1 Long Polling Server. Der Server verwaltet nun eine Menge von offenen Verbindungen. Geht eine Anfrage ein, wird diese entweder mit neuen Nachrichten beantwortet, oder die Verbindung wird in die Menge der offenen Verbindungen aufgenommen.

Listing 6. Long Polling Server: Nachrichten abfragen

```
77       if (messages.length > 0) {
78         res.setHeader("Content-Type", "application/json");
79         res.end(JSON.stringify(messages));
80       } else {
81         longPollingClients.add(res); // register client
82       }
```

Geht eine neue Nachricht ein, wird diese an alle bis dahin erhaltenen offenen Verbindungen gesendet.

Listing 7. Long Polling Server: Nachrichten versenden

```
90       const message: Message = JSON.parse(body);
91       logMessage(message, "LONG POLLING");
92       lastLongPollingMessages.push(message);
93       res.end(); // send 200 OK
94
95       // send message to all clients
96       longPollingClients.forEach((client) => {
97         try {
98           // reserve client by removing it from the set
       immediately
99           longPollingClients.delete(client);
100
101          client.setHeader("Content-Type", "application/json
       ");
102          client.end(JSON.stringify([message]));
103        } catch (_) {
104          // ignore errors
105        }
106      });
```

3.2.2 Long Polling Client. Der Client ändert sich ebenfalls nur minimal. Das Versenden von Nachrichten bleibt unverändert, siehe Listing 4.

Listing 8. Long Polling Client: Nachrichten abfragen

```
25 const fetchMessages = async () => {
26   const response = await fetch("http://localhost:8083/long-
       polling", {
27     method: "POST",
28     headers: {
29       "Content-Type": "application/json",
30     },
31     body: lastMessageTimestamp,
32   });
33   if (response.ok) {
34     const messages = await response.json();
35     emit("appendMessages", messages);
36
37     // update last message timestamp to the timestamp of the
         last message
38     lastMessageTimestamp = messages[messages.length - 1].
       timestamp;
39   }
40   fetchMessages();
41 };
```

3.3 Implementierung von Streaming

Die Implementierung von Streaming hat wieder große Ähnlichkeiten mit den vorherigen Implementierungen.

3.3.1 Streaming Server. Wie bei Long Polling wird auch beim Streaming eine Menge von offenen Verbindungen verwaltet. Der Server schließt diese Verbindungen jedoch nicht mit der ersten eingehenden Nachricht, sondern lässt sie offen.

Listing 9. Streaming Server: Nachrichten abfragen

```
123   if (req.url === "/streaming") {
124     let timestamp = "";
125     req.on("data", (chunk) => {
126       timestamp += chunk; // convert Buffer to string
127     });
128     req.on("end", () => {
129       const messages = [...filterMessages(timestamp,
        lastLongPollingMessages)];
130
131       res.setHeader("Content-Type", "application/json");
132
```

```
133       // return all avaialble messages, but keep the
          connection open
134       if (messages.length > 0) {
135         res.write(JSON.stringify(messages));
136       }
137
138       // register client
139       streamingClients.add(res);
140     });
```

Geht eine neue Nachricht ein, wird diese an Verbindungen gesendet.

Listing 10. Streaming Server: Nachrichten versenden

```
141   } else if (req.url === "/streaming/send") {
142     let body = "";
143     req.on("data", (chunk) => {
144       body += chunk;
145     });
146     req.on("end", () => {
147       const message: Message = JSON.parse(body);
148       logMessage(message, "STREAMING");
149       lastStreamingMessages.push(message);
150       res.end(); // send 200 OK
151
152       // send message to all clients
153       streamingClients.forEach((client) => {
154         client.write(JSON.stringify([message]));
155       });
156     });
```

3.3.2 Streaming Client. Der Client ändert sich nur minimal. Das Versenden von Nachrichten bleibt unverändert, siehe Listing 4.

Listing 11. Streaming Client: Nachrichten abfragen

```
23  const fetchMessages = async () => {
24    const response = await fetch("http://localhost:8084/
          streaming", {
25      method: "POST",
26      headers: {
27        "Content-Type": "application/json",
28      },
29      body: new Date().toISOString(),
30    });
31    if(!response.ok || !response.body) {
32      console.log("Error fetching messages");
33      return;
34    }
35
36    const reader = response.body.getReader();
37
38    const read = async () => {
39      const { done, value } = await reader.read();
40      if (done)
41        return;
42
43      const messages = new TextDecoder("utf-8").decode(value);
44      emit("appendMessages", JSON.parse(messages));
45      read();
46    };
47
48    read();
49  };
```

3.4 Implementierung von WebSockets

Da WebSockets eine vollwertige bidirektionale Kommunikation zwischen Client und Server ermöglichen, ist die Implementierung sehr einfach.

3.4.1 WebSocket Server. Der Server wird durch die Bibliothek `ws` unterstützt. Diese stellt eine Klasse `WebSocket.Server` bereit, welche die Verwaltung von offenen Verbindungen übernimmt. In dieser Implementierung erhält auch der Client, welcher die Nachricht versendet, eine Kopie der Nachricht.

Listing 12. WebSocket Server

```
168 const websocketServer = http.createServer();
169 const wss = new WebSocket.Server({ server: websocketServer
        });
170
171 wss.on("connection", (ws: WebSocket) => {
172   ws.on("message", (message: Message) => {
173     wss.clients.forEach((client: WebSocket) => {
174       if (client.readyState === WebSocket.OPEN) client.send(
          message);
175     });
176   });
177 });
```

3.4.2 WebSocket Client. Der Client erzeugt eine `WebSocket` Instanz, welche die Verbindung zum Server herstellt. Die Nachrichten werden über dessen Methode `send` versendet. Im Falle einer neuen Nachricht wird die Callback-Funktion `onmessage` aufgerufen.

Listing 13. WebSocket Client

```
12  const ws = new WebSocket("ws://localhost:8081");
13
14  const sendMessage = async (msg: Message) => {
15    ws.send(
16      JSON.stringify({
17        type: "message",
18        data: msg,
19      })
20    );
21  };
22
23  ws.onmessage = async (event) => {
24    const reader = new FileReader();
25    reader.readAsText(event.data);
26    reader.onload = async () => {
27      const msg = JSON.parse(reader.result as string);
28      emit("appendMessages", [msg.data]);
29    };
30  };
```

4 Vergleiche zwischen Kommunikationsvarianten

4.1 Analyse der Implementierungen

Vergleichen wir zunächst die Implementierungen in Bezug auf Codezeilen. Dabei wird für den Client jeweils nur der Code innerhalb des `<script>` Tags betrachtet. Kommentar- und Leerzeilen werden nicht mitgezählt. Es ist zudem anzumerken, dass die Zeilenanzahl nicht ausschlaggebend für die Performance ist und lediglich einen groben Überblick über den Implementierungsaufwand geben soll.

Kommunikationsart	Server	Client	Gesamt
Polling	30	31	58
Long Polling	42	33	75
Streaming	35	37	72
WebSocket	11	23	34

Tabelle 1. Codezeilen pro Implementierung aus [7]

Es ist zu erkennen, dass WebSockets wesentlich weniger Code benötigen, um das gleiche Ziel zu erreichen. Auch ist ersichtlich, dass die Implementierung von Polling im Vergleich zu Long Polling und Streaming einfacher ist.

4.2 Analyse der Performance

Um die Performance der Implementierungen zu analysieren, werden zunächst Größen von Nachrichten, HTTP-Anfragen, Websocket-Verbindungen und Websocket-Nachrichten betrachtet. Dabei wird lediglich der HTTP-Overhead betrachtet, da der TCP/UDP-Overhead nicht von der Implementierung beeinflusst werden kann. In Tabelle 2 werden die übertragenen Bytemengen für die verschiedenen Implementierungen verglichen. Es folgt die Berechnung der dort angegebenen Werte.

4.2.1 Größe einer *Message m*. Die Größe einer Nachricht (wie in Listing 1 dargestellt) beträgt 65 (hier sind Timestamp und JSON-Syntax enthalten) + Name + Inhalt Bytes. Es wird in der Regel ein Array von Nachrichten M übertragen, wodurch sich die Größe um zwei Bytes (für die eckigen Klammern) erhöht.

4.2.2 Größe einer Nachrichtenanfrage. Es handelt sich bei der Nachrichtenanfrage um eine HTTP-Anfrage vom Typ `POST` mit einem Payload, welches den Timestamp der letzten empfangenen Nachricht enthält. Die Größe der Request Header der Anfrage ist stark abhängig von vielen Faktoren wie Host, Pfad, User-Agent, etc., weshalb hier nur mit einem ungefähren Wert von 600 Bytes gerechnet werden kann. [3] Die Request-Payload ist bei Polling-Varianten ein Timestamp, welcher 24 Bytes benötigt. Damit kommen wir auf eine Gesamtgröße von 624 Bytes für den Request-Teil der Anfrage.

Die Größe der Response ist abhängig von der Anzahl der Nachrichten, welche der Server an den Client sendet. Der Response Header ist in der Regel 145 Bytes groß. [4] Ist die Antwort leer, so ist die Größe der Response-Payload 2 Bytes (für ein leeres Array). Damit kommen wir auf eine Gesamtgröße von 147 Bytes für den Response-Teil der Anfrage.

Anfragen mit einer leeren Antwort haben eine Gesamtgröße von 771 Bytes. Anfragen mit einer Antwort, welche Nachrichten enthält, haben eine Gesamtgröße von 771 + M Bytes.

4.2.3 Größe einer Nachrichtenübermittlung. Der Request Header bleibt gleich, da die Nachrichtenübermittlung ebenfalls über eine POST-Request erfolgt. Die Größe der Request-Payload ist 65 Bytes + Name + Inhalt Bytes.

Die Response hat eine Größe von 114 Bytes – hier enthält der Response-Header nur den Status-Code und das Datum, da keine Payload übertragen wird.

Damit kommen wir auf eine Gesamtgröße von 714 + m Bytes für eine Nachrichtenübermittlung.

4.2.4 Größe eines Websocket-Verbindungsaufbaus. Für den Protokollwechsel versendet der Client eine HTTP-Anfrage vom Typ `GET` mit einem Request-Header, welcher den Wechsel zum WebSocket-Protokoll startet (vgl. hierzu Abschnitt 2.4.1). Auch hier gehen wir bei dem Request Header von einer durchschnittlichen Größe von 600 Bytes aus. Bei `GET` Requests ist die Request-Payload leer, sodass die Gesamtgröße des Request-Teils 600 Bytes beträgt.

Die Response hat eine Größe von 122 Bytes – hier enthält der Response Header lediglich den Status-Code und die Ankündigung, dass der Protokollwechsel erfolgreich war. Die Response-Payload ist ebenfalls leer, sodass die Gesamtgröße des Response-Teils 122 Bytes beträgt.

Der Aufbau einer Websocket-Verbindung hat somit eine Gesamtgröße von 722 Bytes.

4.2.5 Größe einer Websocket-Nachricht. Nachrichten, die über eine Websocket-Verbindung übertragen werden, haben keinen HTTP-Overhead. Die Größe einer Nachricht ist daher rein die Größe der *Message m* plus dem WebSocket-Overhead. Dieser gibt den Typ der WebSocket-Nachricht an und hat einen Overhead von 26 Bytes im Falle des Typs `message`. Dies gilt sowohl für gesendete als auch empfangene Nachrichten.

[3] Hier wurden die Zeichen von Request-Headern gezählt und gemittelt. Wird die Chat-Applikation auf localhost bereitgestellt und von einem Chrome-Browser abgerufen, beträgt der Request Header für eine Polling-Anfrage 588 Bytes.

[4] Dieser Wert ist weniger variabel als der Request-Header, da hier lediglich der Status-Code, Datum und Content-Type neben ein paar HTTP-Basisdaten enthalten sind.

Tabelle 2. Vergleich von übertragenen Datenmengen der Implementierungen aus [7]

Kommunikationsart	Initial	Regelmäßig	Overhead für Nachrichtenempfang	Nachrichtenversand
Polling	0 Bytes	771 Bytes	771 + M Bytes	714 + m Bytes
Long Polling	0 Bytes	0 Bytes	771 + M Bytes	714 + m Bytes
Streaming	771 Bytes[a]	0 Bytes	m Bytes	714 + m Bytes
WebSocket	722 Bytes	0 Bytes	26 + m Bytes	26 + m Bytes

[a] Tatsächlich nur der Request-Teil einer Nachrichtenanfrage. Der Response-Teil wird bei der ersten eingehenden Nachricht gesendet.

Eine WebSocket-Nachricht hat somit eine Gesamtgröße von 26 + m Bytes.

4.2.6 Schlussfolgerungen. Anhand der Tabelle 2 ist ersichtlich, dass die Menge an Datenverkehr abhängig von der Implementierung in unterschiedlichen Abschnitten der Kommunikation variiert. Es ist anzumerken, dass die Größe der Nachrichten großen Einfluss auf die Größe des Datenverkehrs darstellt. Polling ist die einzige Variante, die einen regelmäßigen Datenverkehr aufweist. Wie oft die Anfragen versendet werden, ist abhängig von der Implementierung bzw. dem gewählten Polling-Intervall Δt.

Es lässt sich argumentieren, dass die die klassische Pollingvariante attraktiver als Long Polling und Streaming ist, wenn über einen kleinen Zeitraum viele Nachrichten übertragen werden sollen, da hier dann weniger Datenverkehr entsteht. Bei einer längeren Zeit, in der keine Nachrichten übertragen werden, ist die klassische Pollingvariante jedoch weniger attraktiv, da hier redundanter Datenverkehr entsteht. Auch könnte dieser Vorteil bei Long Polling und Streaming genutzt werden, indem die Anfragen nur alle Δt Sekunden beantwortet werden.

Für eine Echtzeitkommunikation kommen alle Kommunikationsvarianten bis auf Polling in Frage, da in allen anderen Fällen die Nachrichten sofort übertragen werden. Kommen viele Nachrichten schnell aufeinander, kann es passieren, dass Long Polling mehrere Nachrichten in einem Request versendet, da es immer auf eine neue Anfrage warten muss. In diesem Fall wäre Streaming die bessere Wahl, da hier die Nachrichten sofort übertragen werden können.

Generell ist Streaming Long Polling vorzuziehen, da es den Overhead für Nachrichtenempfang reduziert indem es Redundanzen vermeidet.

WebSockets sind bei weitem die an Overhead geringste Variante, da hier keine HTTP-Requests und -Responses benötigt werden. Da hier eine TCP-Verbindung aufgebaut wird, ist die Nachrichtenübertragung wesentlich effizienter.

4.3 Auswahl einer Kommunikationsvariante für die Chat-Anwendung

Für den Anwendungsfall der Chat-Anwendung ist die Wahl der Kommunikationsvariante abhängig von der Anzahl und Art der Nutzer sowie der Anzahl der Nachrichten, die übertragen werden sollen. WebSockets sind ein relativ neues Feature und werden von älteren Browsern nicht unterstützt. Bei den meisten Browsern wurden WebSockets in Versionen von etwa 2012 implementiert [8]. Soll die Chat-Anwendung auf möglichst vielen Browsern laufen, ist es sinnvoll, eine Kommunikationsvariante zu wählen, die auch von älteren Browsern unterstützt wird. Dabei ist anzumerken, dass der geschätzte Anteil der Nutzer, die einen Browser mit WebSockets verwenden, auf etwa 98.3% geschätzt wird [8], was eventuell eine vernachlässigbare Minderheit darstellt.

Die Wahl fällt eindeutig auf WebSockets, da diese die geringste Latenz aufweisen [9] und die Nachrichten mit minimalem Overhead übertragen werden können.

4.3.1 Auswahl mit Priorität auf Verfügbarkeit. Muss die Chat-Anwendung auf fast allen Browsern laufen, darf WebSockets nicht verwendet werden. Nun muss entschieden werden, welche der anderen Kommunikationsvarianten sinnvoll ist.

Werden sehr viele Nachrichten von vielen Nutzern gleichzeitig versendet, kommen Streaming oder Polling in Frage: Streaming, da hier die Nachrichten mit minimalem Overhead sofort übertragen werden können; und Polling, da Mengen von Nachrichten in regelmäßigen Abständen übertragen werden können. Hierbei ist anzumerken, dass Polling eine hohe Netzwerkbelastung verursachen kann, wenn viele Nutzer die Nachrichten gleichzeitig abfragen. So würden 1.000 Nutzer, die sekündlich Nachrichten anfragen, allein durch die Header etwa 6 Mbps an Netzwerkdurchsatz verursachen [10].

Dann kann je nach Priorität – Effizienz bei großen Mengen von Nachrichten oder Echtzeit – entschieden werden, welche Variante verwendet werden soll. Werden nur wenige Nachrichten oder in großen Abständen versendet, ist Streaming die beste Wahl, da es den geringsten Overhead mit Echtzeitübermittlung hat.

5 Vorstellung von Frameworks

Es gibt Frameworks, die das Implementieren von asynchroner Kommunikation erleichtern.

5.1 Socket.IO

Was ist das? Wie ist es besser oder schlechter für die Implementierung als Chat-App? Performance? Auch: Attraktiv für Verfügbarkeit, da es auf allen Browsern läuft. Weicht auf Polling aus, wenn WebSockets nicht unterstützt werden.

5.2 Faye

6 Fazit

Literatur

[1] R. T. Fielding, M. Nottingham, and J. Reschke, "HTTP Semantics," Internet Engineering Task Force, Request for Comments RFC 9110, Jun. 2022, num Pages: 194. [Online]. Available: https://datatracker.ietf.org/doc/rfc9110

[2] "HTTP request methods." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

[3] "Long polling," Dec. 2021. [Online]. Available: https://javascript.info/long-polling

[4] P. Saint-Andre, S. Loreto, S. Salsano, and G. Wilkins, "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP," Internet Engineering Task Force, Request for Comments RFC 6202, Apr. 2011, num Pages: 19. [Online]. Available: https://datatracker.ietf.org/doc/rfc6202

[5] A. Melnikov and I. Fette, "The WebSocket Protocol," Internet Engineering Task Force, Request for Comments RFC 6455, Dec. 2011, num Pages: 71. [Online]. Available: https://datatracker.ietf.org/doc/rfc6455

[6] P. Lubbers, B. Albers, and F. Salim, *Pro HTML5 Programming*, 1st ed. Apress Berkeley, CA, 2010. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4302-2791-5

[7] H. Wagner, "Seminar2022," 2022. [Online]. Available: https://github.com/hwgn/seminar2022

[8] A. Deveria, "Web Sockets | Can I use..." [Online]. Available: https://caniuse.com/websockets

[9] V. Pimentel and B. G. Nickerson, "Communicating and Displaying Real-Time Data with WebSocket," *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, Jul. 2012, conference Name: IEEE Internet Computing. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6197172

[10] P. Lubbers and F. Greco, "HTML5 WebSocket - A Quantum Leap in Scalability for the Web." [Online]. Available: https://web.archive.org/web/20210422023846/http://websocket.org/quantum.html

Abbildungsverzeichnis

Tabellenverzeichnis