Möglichkeiten zur asynchronen Kommunikation zwischen Webbrowser und Server

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

Zusammenfassung

Bei Webseiten und Webapplikationen ist es häufig notwendig, dass der Server Daten an den Client (Webbrowser) sendet, ohne dass dieser eine Anfrage dafür gestellt hat. Dies ist gemäß der HTTP-Spezifikation nicht vorgesehen, weshalb eine Reihe von alternativen Methoden entwickelt wurden, um diese Funktionalität zu ermöglichen.

In diesem Artikel die Methoden *Polling*, *Long-Polling*, *Streaming* und *WebSockets* vorgestellt und verglichen. Dabei wird auf die Funktionsweise, die Implementierung und die Vor- und Nachteile eingegangen.

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.

Diese Arbeit beschäftigt sich mit verschiedenen Ansätzen zur asynchronen Kommunikation zwischen Client und Server. Sie soll dabei helfen, die Vor- und Nachteile der einzelnen Methoden zu verstehen und eine Entscheidungshilfe für die Wahl der geeigneten Methode liefern.

Dabei wird auf klassisches Polling, die Polling-Varianten *Long Polling* und *Streaming*, sowie auf die bidirektionale Kommunikation mittels *WebSockets* eingegangen. Es wird jeweils die Funktionsweise der Methoden erläutert, eine Beispielimplementierung vorgestellt und die Vor- und Nachteile der einzelnen Methoden aufgezeigt. Dabei werden Performance und Implementierungsaufwand berücksichtigt.

Zunächst wird die theoretische Funktionsweise der einzelnen Methoden erläutert. Es wird dafür auf die Spezifikationen der einzelnen Methoden sowie exemplarische Nachrichtensequenzen zurückgegriffen. Anschließend wird mit jeder Methode eine Chatanwendung implementiert, um die Funktionsweise zu demonstrieren. Anhand der Implementierung wird die Implementierungsaufwand und die Performance der einzelnen Methoden verglichen. Abschließend werden die Vor- und Nachteile der einzelnen Methoden aufgezeigt. Zudem werden gängige Frameworks kurz vorgestellt und mit den implementierten Methoden verglichen. Zuletzt werden die Ergebnisse zusammengefasst und eine Empfehlung für die Wahl der geeigneten Methode gegeben.

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.

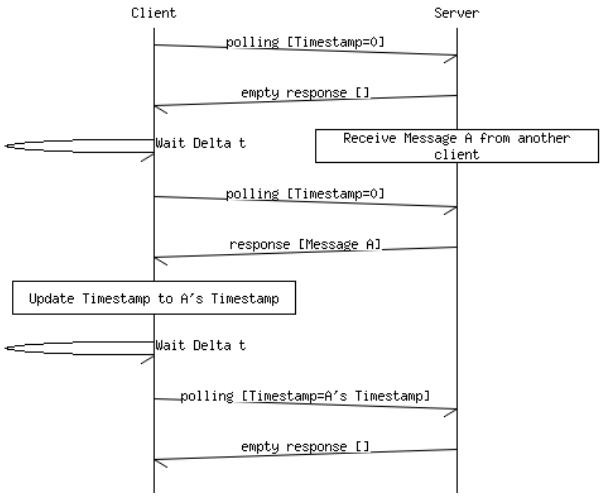

Abbildung 1. Beispiel von Nachrichtenempfang bei klassischem Polling. Der Client sendet regelmäßig eine Anfrage an den Server, welche entweder mit leerer Antwort oder mit neuen Inhalten beantwortet wird. Die Länge der Zeitspanne Δt zwischen zwei Polling-Anfragen ist implementierungsabhängig.

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.

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.

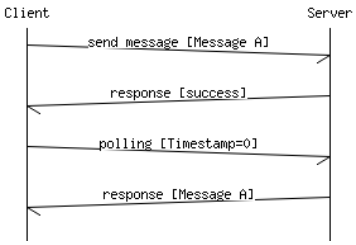

Abbildung 2. Beispiel von Nachrichtenversand bei klassischem Polling. Der Client sendet eine Nachricht an den Server, welche daraufhin mittels Polling abgerufen werden kann.

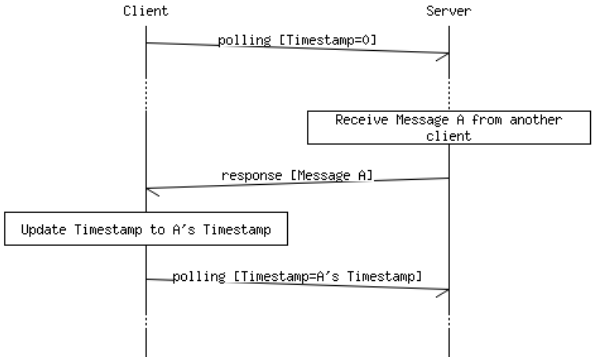

Abbildung 3. Beispiel von Nachrichtenempfang bei Long Polling. Der Server beantwortet die Anfrage des Clients erst, wenn er eine Nachricht hat, die er übermitteln möchte. Der Nachrichtenversand verläuft analog zu Polling.

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].

2.3 Streaming

Eine weitere Optimierung des Polling-Konzepts ist das sog. *Streaming*, manchmal auch *Comet* oder *Server Push* genannt. Es verwendet die HTTP Transferkodierung Chunking (vgl. [1, Abs. 7.1]), 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].

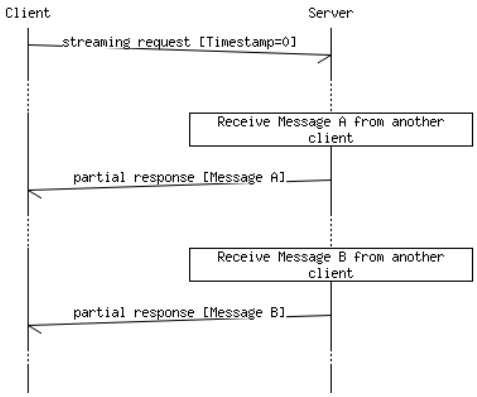

Abbildung 4. Beispiel von Nachrichtenempfang bei Streaming. Der Client initiiert den Streaming-Prozess durch eine einmalige Anfrage. Daraufhin sendet der Server Teilantworten an den Client, sobald neue Nachrichten verfügbar sind. Der Nachrichtenversand verläuft analog zu Polling.

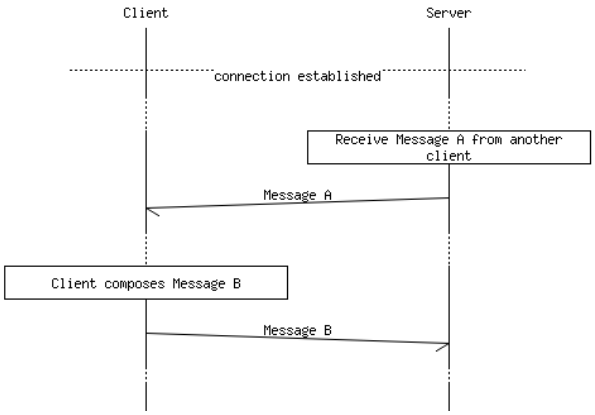

Abbildung 5. Beispiel von Nachrichtenaustausch bei WebSockets. Nach dem Handshake wird eine TCP-Verbindung zwischen Client und Server aufgebaut. Von nun an können Nachrichten in beide Richtungen ausgetauscht werden.

2.4 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 handelt sich bei WebSockets um ein vollwertiges Protokoll, welches auf TCP aufbaut. Es versucht, die Probleme von Polling, Long Polling und Streaming direkt zu adressieren (vgl. [5, Abs. 1.1]), indem es eine TCP-nahe Verbdingung 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.

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]).

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`].

[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.

3.1 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.

Listing 1. Beispielhafte *Message*

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

Im Falle des klassischen Polling wird der Client in regelmäßigen Zeitabständen (hier $\Delta t = 2s$) 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.

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. Da die Abfragen nach Nachrichten in der Regel öfter als die Nachrichten selbst gesendet werden, wird das Array der Nachrichten oft leer sein.

Listing 2. Polling Server: Nachrichtenabfrage

```
37    if (req.url === "/polling") {
38      const timestamp = body;
39      res.setHeader("Content-Type", "application/json");
40      res.end(JSON.stringify([...filterMessages(timestamp,
        pollingMessages)])); // return messages as json
```

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: Nachrichtenempfang

```
41    } else if (req.url === "/polling/send") {
42      const message: Message = JSON.parse(body);
43      logMessage(message, "POLLING");
44      pollingMessages.push(message);
45      res.end(); // send 200 OK
```

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: Nachrichtenversand

```
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[3]. 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: Nachrichtenabfrage

```
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] Hier unterscheidet sich die Implementierung von der Spezifikation in Abschnitt 2.1.1. Der Zeitstempel hätte ebenfalls als Parameter übermittelt werden können.

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: Nachrichtenabfrage

```
67    if (req.url === "/long-polling") {
68      const timestamp = body;
69      const messages = [...filterMessages(timestamp,
        longPollingMessages)];
70
71      if (messages.length > 0) {
72        res.setHeader("Content-Type", "application/json");
73        res.end(JSON.stringify(messages));
74      } else {
75        longPollingClients.add(res); // register client
76      }
```

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

Aufgrund von asynchroner Bearbeitung der Anfragen kann es hier zu Race-Conditions kommen. Daher wird der Versuch, Nachrichten an gespeicherte Verbindungen zu versenden, in einem try-catch-Block abgefangen und Errors ignoriert.

Listing 7. Long Polling Server: Nachrichtenempfang

```
77    } else if (req.url === "/long-polling/send") {
78      const message: Message = JSON.parse(body);
79      logMessage(message, "LONG POLLING");
80      longPollingMessages.push(message);
81      res.end(); // send 200 OK
82
83      // send message to all clients
84      longPollingClients.forEach((client) => {
85        try {
86          // reserve client by removing it from the set
        immediately
87          longPollingClients.delete(client);
88
89          client.setHeader("Content-Type", "application/json
        ");
90          client.end(JSON.stringify([message]));
91        } catch (_) {
92          // ignore errors
93        }
94      });
```

3.2.2 Long Polling Client. Der Client ändert sich ebenfalls nur minimal. Das Versenden von Nachrichten bleibt unverändert, siehe Listing 4. Die Nachrichtenabfrage startet nun allerdings nicht mehr durch ein Intervall, sondern durch einen Selbstaufruf in der Funktion `fetchMessages`.

Listing 8. Long Polling Client: Nachrichtenabfrage

```
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();
```

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: Nachrichtenabfrage

```
116   if (req.url === "/streaming") {
117     const timestamp = body;
118     const messages = [...filterMessages(timestamp,
        longPollingMessages)];
119
120     res.setHeader("Content-Type", "application/json");
121
122     // return all avaialble messages, but keep the
        connection open
123     if (messages.length > 0) {
124       res.write(JSON.stringify(messages));
125     }
126
127     streamingClients.add(res); // register client
```

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

Listing 10. Streaming Server: Nachrichtenempfang

```
128   } else if (req.url === "/streaming/send") {
129     const message: Message = JSON.parse(body);
130     logMessage(message, "STREAMING");
131     streamingMessages.push(message);
132     res.end(); // send 200 OK
133
134     // send message to all clients
135     streamingClients.forEach((client) => {
136       client.write(JSON.stringify([message]));
137     });
```

3.3.2 Streaming Client. Der Client ändert sich nur minimal. Das Versenden von Nachrichten bleibt unverändert, siehe Listing 4. Die Nachrichtenabfrage läuft nun über einen

`Reader`, welcher die eingehenden Nachrichten Blockweise liest.

Listing 11. Streaming Client: Nachrichtenabfrage

```
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();
```

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. Es wird weiterhin der gleiche Typ *Message* verwendet.

Listing 12. WebSocket Server

```
150 const websocketServer = http.createServer();
151 const wss = new WebSocket.Server({ server: websocketServer
        });
152
153 wss.on("connection", (ws: WebSocket) => {
154   ws.on("message", (message: Message) => {
155     wss.clients.forEach((client: WebSocket) => {
156       if (client.readyState === WebSocket.OPEN) client.send(
        message);
157     });
158   });
159 });
160
161 websocketServer.listen(8081, () =>
162   console.log(`Websocket server started on port 8081`)
163 );
```

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 der Kommunikationsvarianten 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. Auch ist die Anzahl der Codezeilen stark abhängig von Programmiersprache, Formatierungen, verwendete Bibliotheken und wie viel der Implementierung in andere Funktionen ausgelagert wurde.

Kommunikationsart	Server	Client	Gesamt
Polling	23	31	54
Long Polling	38	33	71
Streaming	31	37	68
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 dann verglichen[4]. 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 vielen Fällen 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[5]. 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[6]. 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 769 + 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.

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

4.3 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, und die Nachrichtengröße bei allen Implementierungen gleich ist.

Es wird nun auf die Vor- und Nachteile der Kommunikationsvarianten eingegangen. Dabei werden Spezifikation, Implementierung und Analysen der Performance berücksichtigt.

4.3.1 Polling.

- **Vorteile**
  - Einfach zu implementieren: Es sind keine zusätzlichen Bibliotheken notwendig. Zudem hat klassisches Polling den geringsten Implementierungsaufwand unter den Polling-Varianten (siehe Tabelle 1).
  - Vollständige Akzeptanz bei Browsern: Polling basiert auf HTTP, welches von allen Browsern unterstützt wird.
  - Geringere Netzwerkbelastung im Falle von hohen Mengen an Nachrichten: Da bei Polling nur eine Anfrage pro Intervall Δt versendet wird, entsteht eine geringere Netzwerkbelastung bzw. Header-Overhead als bei anderen Polling-Varianten, wenn sehr viele

[4] Aufgrund des begrenzten Umfangs der Ausarbeitung können nicht auf Unterschiede in Latenz eingegangen werden. Oft ist jedoch ersichtlich, wie das Verhalten der Datenübermittlung die zeitliche Effizienz der Kommunikation beeinflusst. Siehe auch die Ausarbeitung von Pimentel und Nickerson, in welcher Latenzvergleiche zwischen WebSocket, Polling und Long Polling vorgenommen werden [8].

[5] 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.

[6] 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	Nachrichtenempfang	Nachrichtenversand
Polling	0 Bytes	771 Bytes[a]	769 + M Bytes	714 + m Bytes
Long Polling	0 Bytes	0 Bytes	769 + M Bytes	714 + m Bytes
Streaming	769 Bytes[b]	0 Bytes	m Bytes	714 + m Bytes
WebSocket	722 Bytes	0 Bytes	26 + m Bytes	26 + m Bytes

[a] 769 Bytes gemäß Abschnitt 4.2.2, plus 2 für ein leeres Array als Response-Payload.

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

Nachrichten in kleinem Zeitraum empfangen werden[7].

- **Nachteile**
  - Hohe Netzwerkbelastung: Da bei Polling regelmäßig Anfragen versendet werden, entsteht eine durchgehende Netzwerkbelastung, welche u. U. keine tatsächlichen Informationen übermittelt (Redundanz). Allein durch eine hohe Anzahl an verbundenen Clients kann die Netzwerkbelastung sehr hoch sein. So würden 1.000 Nutzer, die sekündlich Nachrichten anfragen, allein durch die Header etwa 6 Mbps an Netzwerkdurchsatz verursachen [9].
  - Hoher Header-Overhead für Nachrichtenempfang: Da bei Polling regelmäßig Anfragen versendet werden, entsteht ein hoher Header-Overhead, welcher für den gesamten Zeitraum der Kommunikation in den regelmäßigen Abfragen besteht.
  - Hohe Server- und Client-Last: Da bei Polling regelmäßig Anfragen versendet und beantwortet werden müssen, entsteht eine hohe Last auf Server- und Clientseite.
  - Keine Echtzeitübertragung: Es kann bis zu Δt dauern, bis eine Nachricht empfangen wird. Dies kann z. B. bei einer Chatanwendung zu einer unangenehmen Verzögerung führen.

4.3.2 Long Polling.

- **Vorteile**
  - Reduzierter Header-Overhead: Da bei Long Polling nur eine Anfrage pro eingehende Nachricht versendet wird, entsteht ein geringerer Header-Overhead als bei Polling.
  - Geringere Server- und Client-Last: Da bei Long Polling nur eine Anfrage versendet wird, entsteht eine geringere Last auf Server- und Clientseite als bei Polling.
  - Vollständige Akzeptanz bei Browsern: Long Polling basiert auf HTTP, welches von allen Browsern unterstützt wird.
- **Nachteile**
  - Timeouts: Da bei Long Polling eine Anfrage beliebig lang offen bleiben kann, kann es zu Timeouts kommen, also einem Abbruch der Verbindung.
  - Unnötig reservierte Ressourcen: Betriebssysteme reservieren oft in Antizipation auf eingehende Nachrichten Systemressourcen. Diese Ressourcen werden bei Long Polling unnötig lange reserviert. [4, Abs. 2.2]
  - Verlangsamung bei mehreren Nachrichten in kurzer Zeit: Werden zwei Nachrichten in kurzer Zeit nacheinander versendet, kann die zweite Nachricht erst empfangen werden, nachdem die erste beim Client eingegangen ist und dieser eine neue Anfrage versendet hat. Dies kann zu einer Verzögerung führen.

4.3.3 Streaming.

- **Vorteile**
  - Weiter reduzierter Header-Overhead: Streaming benötigt nur eine Anfrage für alle eingehenden Nachrichten, wodurch der Header-Overhead noch weiter reduziert wird.
  - Geringere Server- und Client-Last: Ähnlich wie bei Long Polling entsteht eine geringere Last auf Server- und Clientseite als bei Polling.
  - Vollständige Akzeptanz bei Browsern: Streaming basiert auf HTTP, welches von allen Browsern unterstützt wird.
  - Keine Verzögerung bei mehreren Nachrichten in kurzer Zeit: Da bei Streaming die Nachrichten sofort übertragen werden können, entsteht keine Verzögerung bei mehreren Nachrichten in kurzer Zeit.
- **Nachteile**
  - Timeouts: Da bei Streaming eine Anfrage permanent offen gehalten wird, kann es zu Timeouts kommen.
  - Unnötig reservierte Ressourcen: Bei Streaming werden für die permanent offene Anfrage Systemressourcen reserviert, selbst wenn sie nicht akut benötigt werden. [4, Abs. 2.2]
  - Möglicherweise nicht unterstützt: Streaming funktioniert nicht auf allen Systemen – Proxys können Pakete bündeln und erst verzögert weiterleiten, wodurch Pakete ggf. nicht zeitgetreu, gebündelt oder aufgespalten ankommen [4, Abs. 3.2]. Im schlimmsten Fall übermittelt die Proxy die Anfrage erst, wenn

[7] Dieser Vorteil wäre auch bei den folgenden Implementierungen gegeben, wenn die Nachrichten in einem Intervall Δt versendet werden, weshalb eine hohe Nachrichtenlast nicht zwingend ein Argument für klassisches Polling darstellt.

sie abgeschlossen ist, was bei Streaming nie der Fall ist.

4.3.4 WebSockets.

- **Vorteile**
  - Kein Header-Overhead: WebSocket-Nachrichten verwenden keine HTTP-Header. Nachrichten werden allerdings durch das WebSocket-Protokoll verpackt und dadurch etwas größer (vgl. Abschnitt 4.2.5).
  - Wesentlich geringere Server- und Client-Last: Da bei WebSockets Kommunikation beinahe auf TCP-Ebene stattfindet [5, Abs. 1.5], entsteht eine geringere Last als bei Polling-Varianten (vgl. Tabelle 2).
  - Echtzeitübertragung: Nachrichten können in beide Richtungen sofort übertragen werden.
  - Vollständige Akzeptanz bei Browsern: WebSockets werden von allen Browsern unterstützt.
  - Geringste Latenz: WebSockets haben im Vergleich zu Polling-Varianten die geringste Latenz [8, Seite 52].
  - Kein Missbrauch von HTTP: Anders als Polling-Varianten[8] verwendet WebSockets das WebSocket-Protokoll, welches für die bidirektionale Übertragung von Nachrichten entwickelt wurde.
- **Nachteile**
  - Keine vollständige Akzeptanz bei Browsern: WebSockets sind ein relativ neues Feature und werden von älteren Browsern nicht unterstützt. Laut *Can I Use...* werden WebSockets von den meisten Browsern in Versionen ab etwa 2012 unterstützt. Nutzer mit WebSocket-fähigen Browsern werden auf etwa 98.3% geschätzt [10].

4.4 Auswahl einer Kommunikationsvariante für die Chatanwendung

Für den Anwendungsfall der Chatanwendung ist die Wahl der Kommunikationsvariante abhängig von der Anzahl und Art der Nutzer sowie der Anzahl der Nachrichten, die übertragen werden sollen.

Soll die Chatanwendung auf möglichst vielen Browsern laufen, ist es sinnvoll, eine Kommunikationsvariante zu wählen, die auch von älteren Browsern unterstützt wird. Gemäß Abschnitt 4.3.4 ist dies nur bei WebSockets eventuell nicht der Fall. Jedoch bildet der Anteil an Nutzern, die keine WebSocket-Verbindungen aufbauen können, eine eventuell vernachlässigbare Minderheit.

Die Wahl für die Chatanwendung fällt eindeutig auf WebSockets, da diese die geringste Latenz aufweisen und die Nachrichten mit minimalem Overhead übertragen werden können. Auch ist davon auszugehen, dass die meisten Nutzer der Chatanwendung einen Browser mit WebSocket-Fähigkeit verwenden werden.

[8] Polling beinhaltet das Versenden von redundanten Anfragen; Long Polling simuliert eine Verbindung mit (sehr) hoher Latenz um eine Antwort herauszuzögern; Streaming verwendet HTTP Chunking, um eine Response für mehrere Nachrichten zu verwenden.

4.4.1 Auswahl mit Priorität auf Verfügbarkeit.

Muss die Chatanwendung auf allen Browsern laufen, dürfen WebSockets (allein, siehe Abschnitt 5 für mögliche Alternativen) 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 infrage: Streaming, da hier die Nachrichten mit minimalem Overhead sofort übertragen werden können; und Polling, da Mengen von Nachrichten in regelmäßigen Abständen übertragen werden können.

Dann kann je nach Priorität – Effizienz bei großen Mengen von Nachrichten oder Echtzeit – entschieden werden, welche Variante verwendet werden soll. Werden in der Regel nur wenige Nachrichten 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. Sie präsentieren sich aufgrund ihrer Funktionalität und ihrer Implementierung als geeignete Alternativen zu den oben vorgestellten Kommunikationsvarianten.

5.1 Socket.IO

Socket.IO ist ein Framework, welches in vielen Programmiersprachen – sowohl Server- als auch Clientseitig – implementiert wurde. Es baut auf WebSockets auf und erweitert diese um Fallbacks für Browser, die WebSockets nicht unterstützen. Dabei greift es auf Long Polling zurück, wenn WebSockets nicht unterstützt werden oder nicht funktionieren. Auch baut es durch Fehler unterbrochene Verbindungen automatisch wieder auf [11]. Zusätzlich zu den Fallbacks bietet es auch einige neuen Features, wie zum Beispiel *Rooms*, die es ermöglichen, Nachrichten an Gruppen von Clients zu senden [12].

Die Perfomance ist im Vergleich zu WebSockets schlechter, da es mehr Overhead hat. Bei 10000 verbundenen Clients benötigt eine wie in Abschnitt 3.4 vorgestellte Implementierung 40MB Heap-Speicher, während *Socket.IO* abhängig von der verwendeten Package 80 bis 170MB benötigt [13].

5.2 Faye

Faye ist ein Framework, das Konzept von Publish-Subscribe gemäß dem Bayeux-Protokoll implementiert [14]. Server und Client wurden für *Node.js* und *Ruby* implementiert. Ähnlich wie Rooms bei *Socket.IO* können Clients bei *Faye* Kanäle abonnieren und Nachrichten an diese Kanäle senden. Ebenfalls wie bei *Socket.IO* stellt *Faye* eine Reihe von Fallbacks zur Verfügung, um die Kompatibilität mit Browsern zu erhöhen, die WebSockets nicht unterstützen [14].

Im Vergleich zu *Socket.IO* ist *Faye* wesentlich unbekannter [15][9], sonst haben die beiden Frameworks ähnliche Eigenschaften.

6 Fazit

Die verschiedenen Kommunikationsvarianten haben unterschiedliche Vor- und Nachteile, weshalb sie in verschiedenen Kontexten sinnvoller erscheinen können als andere. So wurde ermittelt, dass WebSockets in der Performance am besten abschneiden, jedoch nicht von allen Browsern unterstützt werden. Auch haben die verschiedenen Polling-Varianten Aspekte, weshalb sie in bestimmten Situationen sinnvoller sind als andere. So hat zwar Streaming weniger Overhead als Long Polling, bringt aber das Risiko mit sich, aufgrund Proxys nicht richtig zu funktionieren.

Es ist zusätzlich zu diskutieren, ob die Verwendung von Frameworks für eine Anwendung möglicherweise sinnvoller ist. Diese sind zwar in der Regel nicht so performant wie eine selbst implementierte Lösung, können jedoch die Implementierung erheblich erleichtern und bieten zusätzliche Features.

Die Analysen der verschiedenen Varianten waren aufgrund der begrenzten Zeit nicht vollständig, weshalb es sich lohnt, diese weiter zu untersuchen. Interessant wäre es auch, die verschiedenen Varianten in einem größeren Kontext zu untersuchen, um die Vor- und Nachteile besser einschätzen zu können.

Zudem gibt es noch einige weitere Varianten, die nicht in dieser Arbeit betrachtet wurden. Für zukünftige Arbeiten in diesem Feld wäre es erwägenswert, einen größeren Vergleich zwischen verfügbaren Varianten zu erstellen, in welchem auch Latenz und TCP-Overhead analysiert werden.

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] 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

[9] 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

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

[11] "Introduction | Socket.IO." [Online]. Available: https://socket.io/docs/v4/

[12] "Rooms | Socket.IO." [Online]. Available: https://socket.io/docs/v4/rooms/

[13] "Memory usage | Socket.IO." [Online]. Available: https://socket.io/docs/v4/memory-usage/

[14] "Faye: Simple pub/sub messaging for the web." [Online]. Available: https://faye.jcoglan.com/architecture.html

[15] "Comparing faye-websocket vs. socket.io." [Online]. Available: https://npmcompare.com/compare/faye-websocket,socket.io

[9] Widersprüchlich zu dieser Aussage erscheinen die Downloads von *Faye* als wesentlich höher als die von *Socket.IO*. Die Ursache hierfür ist unklar – auf GitHub hat *Socket.IO* etwa zwei Größenordnungen mehr Sterne und Forks als *Faye*.