Workshop: Real-Time Client and Server Communication

Introduction

TODO: Welcome Antematter's employees to the first workshop. Introduce the idea of workshops and discuss their purpose.

- The purpose is to improve interaction between engineers (especially fully-remote engineers who are often disconnected from the rest of the company), while simultaneously disseminating valuable information.
- It can be technical and non-technical. Technical workshops can be both theoretical or practical (or a combination of both).
- Clarify that this workshop is going to lean heavily on the theoretical side.

The Problem

The problem is quite simple: we need to establish real-time communication between a client and a server. This kind of functionality is essential for any feature

that requires instantaneous data exchange, such as:

- Live updates (for example, live weather updates or real-time analytics).
- Chatting and messaging.
- Real-time collaboration (for example, collaborative whiteboarding).

This functionality should therefore be able to cater for both unidirectional and bidirectional communication.

The Solution(s)

Instead of jumping to an ideal solution straight away, let's try to build it step by step from the ground up.

Long Polling

The web was built around a request-response model. The client initiates an HTTP request, and the server responds with the required data (and closes the connection). The distinction between the client and the server is significant: **the client initiates the request, not the server.**

Now, in an application where we want real-time data, the **most primitive** approach would simply be to abuse this request-response model. The client can keep sending HTTP requests to the server in the hopes of *pulling* any new data. This is commonly referred to as the traditional or 'short polling' technique. In JavaScript land, any modern libraries (such as Axios or the Fetch API) that are based on the `XMLHttpRequest` (XHR) object can use this technique.

The shortcomings of this technique are quite apparent. Not only is the server (and the client, for that matter) under increased load, but the network itself can become congested. Each request would incur HTTP overhead (in the form of DNS resolution, a TCP handshake, and a full set of HTTP headers which could often represent a large percentage of the actual data transmitted) which would ultimately lead to high latency and dropped connections. This kind of approach is not feasible — especially when an application starts to scale.

The first step towards a better solution is to **limit the number** of requests that are initiated from the client, but still be able to respond with updated data. As such,

one possible solution is the ‘long polling’ technique. The major difference here is that instead of responding right away, the server keeps the request open for a longer period — until new data is available (or a preconfigured timeout occurs, in which case it sends an empty response). Once the client receives the response, it immediately initiates a new long polling request.

A simple way to implement long polling using JavaScript would involve using the `setInterval` method in our API route to periodically check our data source for any changes, as follows:

```
const express = require('express');
const app = express();

// Dummy data to simulate server updates
let latestData = 'Initial data';

// Route handler for the long polling endpoint
app.get('/long-polling', (req, res) => {
  const sendUpdates = (data) => {
    res.json({ data }); // Send data to the client
  };

  // Check for updates every 5 seconds
  const intervalId = setInterval(() => {
    const newData = generateNewData(); // Generate new data
    if (newData !== latestData) {
      clearInterval(intervalId); // Stop the interval
      latestData = newData; // Update the latest data
      sendUpdates(newData); // Send updates to the client
    }
  }, 5000); // 5 seconds interval

  // Set timeout for long polling (e.g., 30 seconds)
  setTimeout(() => {
    clearInterval(intervalId); // Stop the interval
    res.status(204).end(); // No updates within the timeout
```

```
  }, 30000); // 30 seconds timeout
});

// Function to generate dummy data (replace with your actual dat
function generateNewData() {
  // Generate random data for demonstration purposes
  return Math.random().toString(36).substring(2, 15); // Random
}

// Start the Express server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is listening on port ${PORT}`);
});
```

Another simple way is to use an `EventEmitter` and biind it to our data source.

Although long polling greatly reduces the number of requests that are initiated from the client, we still haven't dealt with the problem of HTTP overhead. Additionally, we're still dealing with a large degree of latency since the server essentially has to *wait* for the next long polling request from the client (before it can respond with a long polling response). This is quite far from *real-time* communication, where the client has access to new data *as soon* as its available.

HTTP (Server) Streaming

Perhaps the next step towards a better solution still is to devise an approach in which requests from the client are **never** closed — even after a response has been delivered. This should, theoretically, help us reduce HTTP overhead (since we're not sending requests and responses back and forth) and minimise latency. Now, this is not something that the HTTP protocol was originally designed for, but we can work around that.

In the HTTP/1.1 specification (defined in RFC 2616), the concept of "chunked transfer encoding" was introduced, which basically allows the server to send a response back to the client in a series of "chunks" instead of all at once. This was introduced to cater for cases where the response's content length could not be

determined in advance. As such, the server will omit the `Content-Length` header from the response, and simply add the length of the current chunk at the beginning of each chunk.

As such, we can exploit this concept. We can set the value of the `Transfer-Encoding` header to `chunked`, and keep any incoming requests indefinitely open (or until a preconfigured timeout occurs). New data can simply be delivered as a chunk. Let's mimic this by going back to our JavaScript example:

```
const express = require('express');
const app = express();

// Middleware to set headers for chunked encoding
app.use((req, res, next) => {
  // Set headers for HTTP chunked encoding
  res.setHeader('Content-Type', 'text/plain');
  res.setHeader('Transfer-Encoding', 'chunked');
  next();
});

// Route handler for the streaming endpoint
app.get('/stream', (req, res) => {
  // Function to send a chunk of data
  const sendChunk = () => {
    // Generate random data for each chunk
    const chunk = generateRandomData();
    // Send the chunk to the client
    res.write(chunk + '\n');
  };

  // Function to periodically check for new data
  const checkForNewData = () => {
    // Generate random data
    const newData = generateRandomData();
    if (newData !== latestData) {
      // If new data is available, send it immediately
```

```
      sendChunk();
      latestData = newData; // Update the latest data
    }
  };

  // Send the initial chunk immediately
  sendChunk();

  // Set interval to periodically check for new data
  const intervalId = setInterval(checkForNewData, 2000); // Chec

  // Timeout to stop streaming after 1 minute (configurable)
  setTimeout(() => {
    clearInterval(intervalId); // Stop interval
    res.end(); // End the response to stop streaming
  }, 60000); // 1 minute timeout
});

// Function to generate random data for each chunk
function generateRandomData() {
  // Generate random data (replace with your actual data source)
  return Math.random().toString(36).substring(2, 15); // Random
}

// Start the Express server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Server is listening on port ${PORT}`);
});
```

This technique is essentially called HTTP server streaming.

You would think that this solution has solved the problem at hand. Let's not celebrate just yet. There are still several problems with this approach:

- Compared to long polling, server streaming places less load on the server. However, we still need to dedicate resources for each open connection. Again,

the HTTP protocol was not originally designed with the idea of long, persistent connections. Scalability is still going to be a problem (although significantly less than with long polling). **Plus, the server now has the additional responsibility of handling client disconnects and stale connections gracefully.**

- Another problem that we're now introducing can be referred to as "back pressure". How do we handle cases in which our server is producing new data at a rate faster than clients can consume them? Do we add some sort of intermediary such as a buffering mechanism? Not all clients (browsers) will be able to handle this.
- This approach could work for unidirectional communication, but we're going to hit a fence as soon we require bidirectional communication.

Interestingly, the HTTP/2 specification (defined in RFC 7540) disallows the use of the `Transfer-Encoding` header altogether, since it introduced more efficient mechanisms for data streaming.

It seems that the next hypothetical step forward likely entails abandoning the HTTP protocol altogether.

However, before we take this next step, I'd like to take a step **sideways** first and cover some an additional technique that refines server streaming even further.

SSE (Server-Sent Events)

SSE is essentially a standardised **API** in the HTML5 specification that is built on top of HTTP server streaming. All major browsers (except Internet Explorer 🙂) currently support SSE.

SSE typically utilises UTF-8 encoding to send text-based "events" as they are produced (binary data cannot be transferred natively). In order to initiate an SSE connection, a client sends a `GET` request to the server with the following headers:

```
GET /api/v1/live-data
Accept: text/event-stream
Cache-Control: no-cache
Connection: keep-alive
```

The `Accept: text/event-stream` header is used to signal to the server that the client is expecting an event stream as the response. The `Connection: keep-alive` header is used to keep the connection open.

Once the server receives the request, it can respond as follows:

```
id: 1
event: weather
data: BERLIN 25.5
data: LONDON 24.3
data: PARIS 27.4
```

As soon as any updated data is available, the server can send another event:

```
id: 2
event: weather
data: BERLIN 25.5
data: LONDON 24.4
data: PARIS 27.4
```

As you can see, all events need to be accompanied with a unique `id` and an `event` field which is used to identify the data source. Additionally, the server can also specify a value for the `retry` field, which instructs the client to reconnect after the specified time if the connection is lost.

The main benefits that it offers over HTTP server streaming include:

- Automatic reconnection: If you recall, one of the problems with HTTP server streaming is the fact that the server has to manage client disconnects itself. SSE solves this by adding a reconnection mechanism.
- Standardised transmission format: Unlike HTTP streaming, which may often require custom implementations, SSE events have a structured format (with specific fields). Since all modern browsers widely support SSE out of the box, there's no fiddling involved.
- Efficiency: SSE operates on an event-driven architecture. With HTTP server streaming, the server has to continuously send data back to the client even if it is not changing (this could simply be an empty response too, but the point still

stands). SSE's event-driven nature means that processing can be done asynchronously.

- Simplicity:

However, SSE is far from perfect.

- As mentioned above, SSE typically utilises UTF-8 encoding to send text-based data. This means that binary data (such as an image or audio file) must be encoded into some form of textual format first (Base64 encoding is a common approach). The problem here is that this will result in an increase in data size (by around 33%-37%). This overhead can be a problem when handling multiple clients.
- Once again, we'll need some sort of a workaround when attempting to use SSE for bidirectional communication. This will introduce additional overheads.

WebSocket

If you've been paying attention, you should have noticed that all of the solutions we've discussed so far abuse HTTP for data transmission — and hence adopt the overheads that it comes with. Perhaps we need to abandon HTTP altogether.

This is where the WebSocket protocol comes in. It provides us with full-duplex (i.e., bidirectional) communication channels over a single, **persistent** TCP connection. The WebSocket protocol operates on top of Layer 4 of the OSI model. This means that it is a direct replacement of the HTTP protocol (which also operates on Layer 4). In fact, WebSocket was designed to operate over HTTP ports 80 and 443 so that it can be seamlessly integrated with existing HTTP infrastructure.

The process for setting up a WebSocket looks a little something like this:

- The client initiates a handshake by **sending an HTTP request** to the server, indicating its intention to upgrade to a WebSocket connection.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
```

```
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat

Origin: http://example.com
```

The `Sec-WebSocket-Key` is a random string of sixteen characters (encoded in Base64) which is used to ensure that clients don't accidentally request an upgrade to the WebSocket protocol. It is also used to ensure that the response is actually from a server, and not, say, a cache intermediary.

The `Sec-WebSocket-Protocol` field is optionally used by the client to indicate what subprotocols (application-level subprotocols built over the WebSocket protocol) are supported by it. The server selects a subprotocol and echos it back in the response.

- The server responds with an `HTTP 101 Switching Protocols` status code, signalling that the connection has been upgraded. We now have full-duplex communication over a single TCP connection.

  ```
  HTTP/1.1 101 Switching Protocols
  Upgrade: websocket
  Connection: Upgrade
  Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
  Sec-WebSocket-Protocol: chat
  ```

  The `Sec-WebSocket-Accept` field is generated by appending the `Sec-WebSocket-Key` with a global unique identifier (GUID) `258EAFA5-E914-47DA-95CA-`, taking that SHA-1 hash of the result, and then encoding it using Base64. The client can recompute this result once it receives a response. This serves a couple of purposes:

  - The client can ensure that the response is from a valid server that supports the WebSocket protocol (and not some caching intermediary that's merely returning a cached `HTTP 101 Switching Protocols` response.
  - The server can ensure that a client isn't mistakingly requesting a protocol upgrade. This is important since both the HTTP and

WebSocket protocols use the same ports.

- Data exchange is done in the form of frames. The header size for each frame is much smaller at around two bytes. Since we're not re-establishing a TCP connection each time, we're effectively lowering our overhead.
- WebSockets are stateful. This state can be maintained via the `Sec-WebSocket-Key` field, which is unique for each WebSocket session.

Bonus: WebRTC (Web Real-Time Communication)

WebRTC is a relatively newer protocol that came out in 2020. Unlike WebSocket, WebRTC utilises UPD under the hood. This allows for much greater throughput (but at the cost of reliability).

More importantly, WebRTC provides developers with a set of JavaScript APIs which allow for direct P2P communication between two web browsers. This reduces latency, which is key when transferring audio and video data.

In practice, however, two WebRTC clients do initially need a server in order to set up the connection.

The Future: HTTP/3?

Since we've been discussing the HTTP protocol, it makes sense to take a look at what the future will bring for it.

The next major upgrade to the HTTP/2 specification (which came out in 2015) promises to bring the following updates:

- It'll utilise the QUIC protocol instead of TCP. QUIC is a transport layer protocol (built by Google) that uses UDP underneath. It handles problems like congestion control and retransmission (which UDP does not cater for) itself.
- It'll be optimised for mobile devices, which switch between connections often (and might not have stable connections in the first place).

Given the context of this document, an important benefit of HTTP/3 includes the ability for clients to skip the handshaking process for servers that they've previously connected to. This *might* actually make techniques like long-polling more feasible in situations where they currently can't really be used.

Glossary

https://www.pubnub.com/guides/long-polling/

https://ably.com/topic/long-polling

https://www.rfc-editor.org/rfc/rfc6202#section-2.1

https://stackoverflow.com/questions/12555043/my-understanding-of-http-polling-long-polling-http-streaming-and-websockets

https://levelup.gitconnected.com/understand-and-implement-long-polling-and-short-polling-in-node-js-94334d2233f3

https://datatracker.ietf.org/doc/html/rfc2616

https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Transfer-Encoding

https://datatracker.ietf.org/doc/html/rfc7540

https://medium.com/deliveryherotechhub/what-is-server-sent-events-sse-and-how-to-implement-it-904938bffd73

https://en.wikipedia.org/wiki/Base64

https://datatracker.ietf.org/doc/html/rfc6455

https://getstream.io/glossary/webrtc-protocol/#:~:text=How Does WebRTC Work%3F,friendly and easy to implement.

https://www.cloudflare.com/learning/performance/what-is-http3/