CSE1500: Web - Lecture 8 - Real-time Applications With Web Sockets

Lecture Summary

The WDT Team

Ujwal Gadiraju, Sagar Chethan Kumar, Ciprian Ionescu

Contents

⚠️ These summaries are not extensive overviews of lecture content and **DO NOT** cover all possible exam material. Rather, these serve as a refresher and capture **main points** for a given lecture.

Learning Goals

You should be able to do the following after following the lecture:

- **Develop server-side code for authorization, cookies, sessions, and database access. (LO5)**

1 Real-time Applications With Web Sockets

1.1 Why WebSockets?

In traditional HTTP, the client sends a request to the server, and the server responds with the requested data. This request-response model requires continuous polling from the client to the server, which can result in increased latency and decreased efficiency. On the contrary, WebSockets provide bi-directional, full-duplex communication between the client and server. This is facilitated through single-socket connection. which is started as an HTTP request/response handshake. See the figure below for an overview:

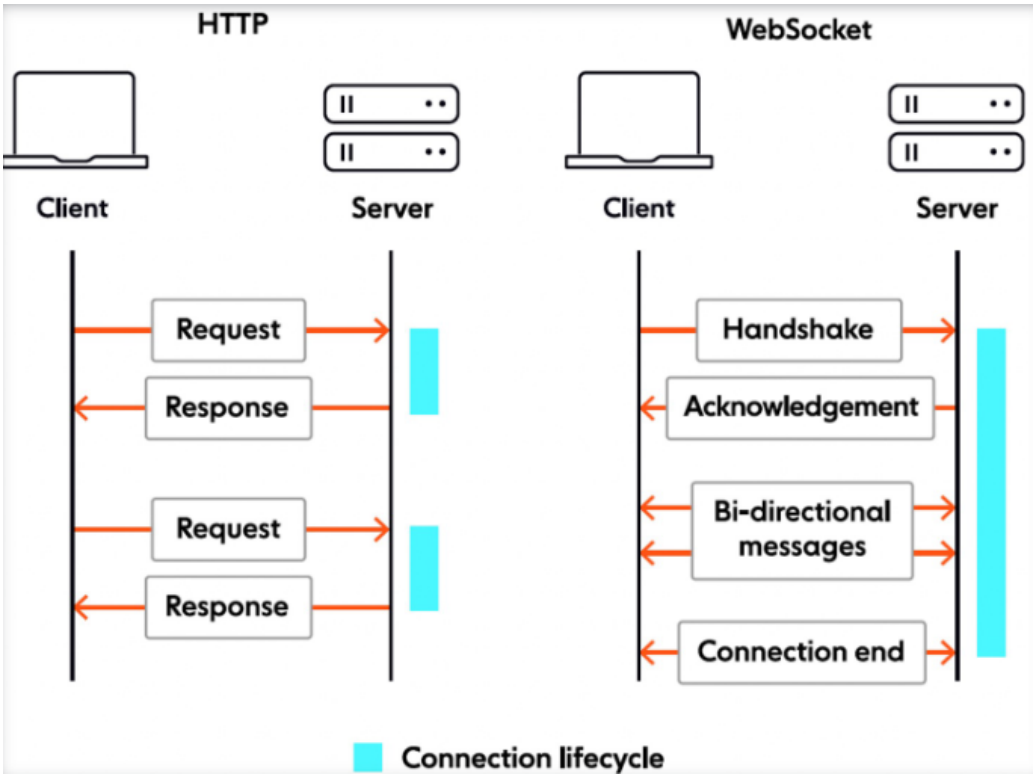

Figure 1: WebSockets vs. HTTP

For example, when a user sends a message in a chat application, through WebSockets the message can be instantly delivered to all other users without refreshing the page or making frequent HTTP requests. This results in a more seamless and efficient user experience.

1.2 The WebSocket Protocol

The WebSocket protocol uses the ws/wss scheme in a URI in order to connect to the WebSocket server. After connecting to the WebSocket server it sends a normal initial HTTP request in order to start the connection. This request is sent as follows:

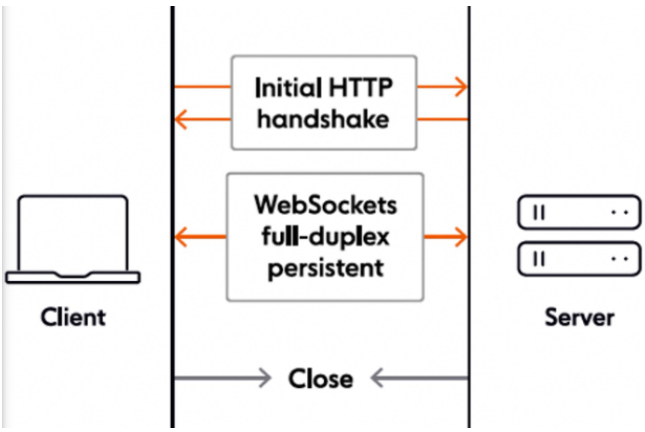

Figure 2: The WebSocket Protocol

Handshake Request

```
GET wss://example.com:8181/ HTTP/1.1
Host: localhost:8181
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: zy6Dy9mSAIM7GJZNf9rI1A==
```

Handshake Response

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Sec-WebSocket-Accept: EDJa7WCAQQzMCYNJM42Syuo9SpQ==
Upgrade: websocket
```

Once this has been established, there are several events a client should listen for. These requests are **open**, **message**, **error** and **close**.

- The **open event** is fired when a connection as successfully started after the handshake is completed.
- The **message event** is fired whenever there's a new message that has been sent by the other party.
- The **error event** is fired when there's an unexpected error with the connection. An example of this could be that some data couldn't be sent.
- The **close event** is fired after the connection has been closed by the other party.

1.3 WebSocket + Node.js + Express

This is a short example of how to implement WebSockets in Node.js using the Express framework.

Server

```
const express = require('express');
const WebSocket = require('ws');

const app = express();

// Initialize WebSocket server
const wss = new WebSocket.Server({ port: 8080 });

// WebSocket event handling
wss.on('connection', (ws) => {
  console.log('A new client connected.');

  // Event listener for incoming messages
  ws.on('message', (message) => {
    console.log('Received message:', message.toString());

    // Broadcast the message to all connected clients
    wss.clients.forEach((client) => {
      if (client.readyState === WebSocket.OPEN) {
        client.send(message.toString());
      }
    });
  });

  // Event listener for client disconnection
  ws.on('close', () => {
    console.log('A client disconnected.');
  });
});

// Start the server
const port = 3000;
app.listen(port, () => {
  console.log(`Server started on http://localhost:${port}`);
});
```

1.4 Evaluating WebSockets

WebSockets offer a number of advantages over traditional HTTP requests, particularly in scenarios that demand real-time communication and efficient data transfer. Below, we elaborate on some of these benefits in detail:

(i) **Reduced Latency and Increased Performance:** Unlike HTTP requests, which follow a request-response model and often involve opening and closing connections repeatedly, WebSockets maintain a persistent, open connection between the client and server. This eliminates the need for repeated handshakes, resulting in minimal overhead per message sent. Consequently, latency is significantly reduced, making WebSockets particularly well-suited for real-time applications such as live sports updates, collaborative editing tools, and financial trading platforms where low latency is critical for a smooth user experience.

(ii) **Support for Subprotocols and Extensions:** The WebSocket protocol allows the definition of subprotocols that can be negotiated during the initial handshake. This capability enables applications to establish customized messaging patterns and adhere to specific data exchange protocols, such as those required for chat systems, multiplayer game synchronization, or even more complex domain-specific tasks. Additionally, WebSocket extensions offer features such as data compression and message multiplexing, providing developers with fine-grained control over the communication channel and enhancing flexibility for sophisticated use cases.

(iii) **Full-Duplex Communication for Real-Time Data Push:** WebSockets enable full-duplex communication, meaning both the client and server can send and receive data independently of each other. This contrasts with HTTP, where a client typically initiates requests and awaits responses. Full-duplex communication empowers the server to push data to the client as soon as new information becomes available, without any explicit polling from the client. This capability is ideal for applications such as instant messaging, live-streaming video, interactive dashboards, and online multiplayer games, as it ensures rapid updates and a responsive user experience.

Despite their numerous advantages, WebSockets come with certain drawbacks and complexities that developers must consider:

- **Resource Intensity and Server Strain:** Since WebSockets maintain a persistent connection for each client, they can place a higher load on the server compared to traditional HTTP request-response cycles. Each open connection consumes system resources, including memory and CPU processing power, which can become a bottleneck as the number of active connections increases. This can lead to scalability challenges, particularly for applications with a large and fluctuating user base.

- **Complex Load Balancing:** WebSockets introduce challenges for load balancing due to their stateful nature. Unlike stateless HTTP requests, where any request can be handled by any server instance, WebSocket connections need to maintain continuity with a specific server node, often requiring sticky sessions or connection routing techniques. This complexity can necessitate specialized infrastructure or application-layer solutions to ensure reliability and distribute connections evenly across servers.

- **Security Considerations:** The persistent connection offered by WebSockets exposes applications to potential security risks, such as denial-of-service (DoS) attacks, connection hijacking, or data interception if encryption is not properly applied. Developers

must ensure that appropriate security mechanisms, such as TLS encryption, authentication, and connection limits, are implemented to mitigate these risks and safeguard data transmitted over WebSockets.

By understanding these trade-offs, developers can make informed decisions on whether WebSockets are the right fit for their specific application needs, balancing performance gains with operational and security considerations.

2 Sample Question Types

🙃 Refer to the Weblab environment and the `.zip` file from the lecture.

3 References and Further Reading

Here are some useful resources for learning more about **Lecture 8**:

- WebSocket vs HTTP
- WebSockets Explained