Homework 3: Live Chat

At this point, you have built a dynamic web application where users can share images and comments with each other using only a TCP socket, your understanding of web protocols, and your programming skills. This is a great accomplishment, but let's take this further and allow users to interact in real time.

You will implement WebSockets to add a live chat feature to your app, and WebRTC to enable peer-to-peer video chat. Let's begin.

Simplifying Assumptions

You can assume the following throughout this assignment:

- FIN bit will always be 1
- The 3 reserved bits will always be 0
- You can ignore any frames with an opcode that is not bx0001 or bx1000
- Additional WebSocket headers are compatible with what we discussed in class (ie. You don't have to check the Sec-WebSocket-Version header)
- When establishing a WebRTC connection, there will be exactly 2 WebSocket connections (ie. If you receive a message from one user, you know the message is for the other user)

For this assignment, add the following HTML and JavaScript to the homepage of your app. As always, feel free to customize the front end to make it your own as long as you still fulfill the testing procedures of each objective.

```
<label for="chat-comment">Chat: </label>
<input id="chat-comment" type="text" name="comment">

<button onclick="sendMessage()">Send</button>

<div id="chat"></div>

<br/>
<button onclick="startVideo()">Start My Video</button>
<br/>
<button onclick="connectWebRTC()">Start Video Chat</button>
<br/>

<video id="myVideo" autoplay muted></video>
<br/>
<video id="otherVideo" autoplay></video>
```

JavaScript: https://cse312.com/static_files/new-javascript.js

This JavaScript assumes that welcome() is still being called when the page loads. Be sure that you still have this line in your HTML: "`<body onload="welcome();">`

Objective 1: WebSocket Handshake

Implement the handshake of the WebSocket protocol at the path “/websocket”.

```
const socket = new WebSocket('ws://' + window.location.host + '/websocket');
```

This line, which is in the provided JavaScript, will make a GET request to the path “/websocket” and attempt to upgrade the TCP socket to a WebSocket connection.

You may use libraries to compute the SHA1 hash and Base64 encoding.

Testing Procedure

1. Start your server using docker-compose up
2. Open a browser and navigate to http://localhost:8080/
3. Open the network tab of the browser console (refresh the page if necessary)
4. Verify that there is a successful WebSocket connection with a response code of 101
5. Open a second browser tab and repeat steps 2-4 to verify that the server supports multiple simultaneous WebSocket connections

Objective 2: Live Chat Over WebSockets

Enable users to chat using their WebSocket connections. The provided JavaScript and HTML will send WebSocket frames containing chat messages when a user submits text to the chat. The payload of each frame will be a JSON string in the format:

```
{
'messageType': 'chatMessage',
'comment': comment_submitted_by_user
}
```

Please note the messageType as your server will handle 4 different messageTypes by the end of this assignment. You should check the messageType and run different code for each different type.

For this objective, you will parse WebSocket frames that are received from any open WebSocket connection, parse the bits of the frame to read the payload, then send a WebSocket frame to all connected WebSocket clients, including the sender, containing the new message. The message sent by your server must be in the format:

```
{
'messageType': 'chatMessage',
'username': random_username,
'comment': html_escaped_comment_submitted_by_user
}
```

For the username, you should generate a random username for each user. These usernames must be consistent (ie. don't generate a new username for every message. User's should be able to have a conversation and know when the same person sent multiple messages). Don't overthink username generation and feel free to use this code {`username = "User" + str(random.randint(0, 1000))`}. It's ok to have a small chance of 2 users getting the same random username. Usernames should be generated and stored on the server, not the front end.

Security: Don't forget to escape the HTML in your users' comments.

Suggestion: There are a few separate steps involved in this objective. It will help your debugging process if you implement and test this functionality 1 step at a time. For example, make sure you can read and parse frames properly before trying to send frames and try sending a frame only to the user sending a message before broadcasting the messages to all users.

For this objective, it is ok if your server only handles frames with fewer than 126 bytes of payload.

Testing Procedure

1. Start your server using docker-compose up
2. Open a browser and navigate to http://localhost:8080/
3. Open at least 2 browser tabs and enter chat messages in each. Ensure that all messages have a payload length <126
4. Verify that each user can see both their own messages, and messages sent by other users in real time
5. Verify that multiple messages sent by the same user have the same username
6. Verify that the messages have been sent/received using a WebSocket connection
7. **Security**: Verify that the submitted HTML displays as text and is not rendered as HTML

Objective 3: WebSocket Features

Add the following features to your WebSocket connections:

- When a frame with an opcode of bx1000 (disconnect) is received, the connection should be severed and removed from any storage on your server. When new messages are received, any disconnected WebSocket connections should not be sent the new message. (ie. Disconnects must be handled gracefully)
- Handle messages of arbitrary size. This includes both messages where the 7-bit payload is set to 126 and 127. We will test with frames larger than 65536 bytes to ensure you handle all three cases (We will not test with payloads > 131000 since chrome sends messages over this size in multiple frames)
    - **Note**: This implies that you will need to buffer your WebSocket frames. You should read from the TCP connection once, parse only the headers of the frame, parse the payload length, then check if you've read that many bytes of payload. If not, you must buffer before de-masking and reading the payload.

Testing Procedure

1. Start your server using docker-compose up
2. Open at least 3 browser tabs and navigate to http://localhost:8080/ in each
3. Enter chat messages of varying size. Ensure that at least one message has a payload length >=126 but <=65536 and at least one has payload length >65536
4. Verify that each user can see both their own messages and messages sent by other users in real time
5. Verify that the messages have been sent/received using a WebSocket connection
6. **Security**: Verify that the submitted HTML displays as text and is not rendered as HTML
7. Close one of the browser tabs
8. Send messages in the remaining tabs and verify that the chat is still functioning

Objective 4: Persistent Chat History

Use your database to store all of the chat history for your app. This will allow the chat history to persist after a server restart.

Add a path to you server of "GET chat-history" which returns every saves chat message and returns them as a JSON string in the format:

```
[
{'username': 'user596', 'comment': 'hello world'},
{'username': 'user1', 'comment': 'welcome to the chat!'}
]
```

The Content-Type of your response should be `'application/json; charset=utf-8'`.

Testing Procedure

1. Start your server using docker-compose up
2. Open a browser and navigate to http://localhost:8080/
3. Send several messages to the chat
4. Restart the server using docker-compose restart
5. Open a new browser tab and navigate to http://localhost:8080/
6. Verify that the messages sent in step 3 are visible on the page
7. Check the code the ensure that a database is being used via docker-compose (ie. File IO, SQLite, etc used in the same container as your app cannot be used for this objective)

Objective 5: Video Chat Over WebRTC

Add features to your server so it can function as a WebRTC signaling server for your users. This signaling will occur over your WebSocket connection. There are three different types of messages used to establish WebRTC connections:

```
{'messageType': 'webRTC-offer', 'offer': offer}
{'messageType': 'webRTC-answer', 'answer': answer}
{'messageType': 'webRTC-candidate', 'candidate': candidate}
```

Where offer, answer, and candidate are all generated by the WebRTC code built into your browser. Your task is to forward these messages to the other user via their WebSocket connection. To avoid overcomplicating this objective, you may assume that there are exactly 2 WebSocket connections when receiving WebRTC messages. This means that when a message is received on one WebSocket connection, you will send the message to the only other WebSocket connection.

Remember, your server is merely acting as a means for the 2 clients to communicate while they establish a peer-to-peer connection. The content of the messages you handle do not need to be parsed or processed. Your task is to extract the payload of these WebSocket messages, verify that they are not chat messages (and are WebRTC messages), then send the payload to the other WebSocket connection. The clients will do the rest through the front end.

Testing Procedure

1. Start your server using docker-compose up
2. Open exactly 2 browser tabs and navigate to http://localhost:8080/ on each
3. In each tab:
    a. Allow the camera/mic to be accessed
    b. If the local video has not started on the page load, click the "Start My Video" button
4. In one of the 2 tabs, click the "Start Video Chat" button
5. Verify that 2 videos are show in both of the tabs
    a. Both videos will be showing the same feed, but the remote video should be slightly behind the local video which is evidence that there is a video streaming connection between the two tabs. You'll also hear some horrible sounding audio with feedback. Don't worry, that means it's working :)
6. Shut down the server by pressing ctrl+c in the terminal running docker-compose (or stop the containers using Docker desktop
7. Verify that the video streams are not interrupted. After the signaling server is used to establish the connection, this is a true peer-to-peer stream

Submission

Submit all files for your server to AutoLab in a .**zip** file (A .rar or .tar file is not a .zip file!). Be sure to include:

- A docker-compose.yml file in the root directory that exposes your app on port 8080 and runs a separate container for your database
- A Dockerfile that is used by your docker-compose configuration
- All of the static files you need to serve (HTML/CSS/JavaScript/images)

It is **strongly** recommended that you download and test your submission after submitting. To do this, download your zip file into a new directory, unzip your zip file, enter the directory where the files were unzipped, run docker-compose up, navigate to localhost:8080 in your browser, and go through the testing procedures for all 5 objectives. This simulates exactly what the TAs will do during grading.

If you have any Docker or docker-compose issues during grading, your grade for each objective may be limited to a 1/3.

Grading

Each objective is equally weighted and will scored on a 0-3 scale as follows:

3	Clearly correct. Following the testing procedure results in all expected behavior
2	Mostly correct, but with some minor issues. Following the testing procedure does not give the exact expected results
1	Clearly incorrect, but an honest attempt was made to complete the objective. Following the testing procedure gives completely incorrect results or no results at all. This includes issues running Docker or docker-compose even if the code for the objective is correct
0	No attempt to complete the objective or violation of the assignment (Ex. Using an HTTP library) -or- a security risk was found while testing the objective

The following conversions will be used when translating grades on this 0-3 scale to percentage grades:

3	100%
2	80%
1	60%
0	0%