Project Report

Anton Kaminsky, Chedy Sankar, Alexander Greff

CSCD58: Computer Networks, Fall 2021

Table of Contents

---

0. Video Link

https://youtu.be/zsFLU8GWrQg

1. Team Members

Anton Kaminksy:

- Email: *anton.kaminsky@mail.utoronto.ca*
- Utorid: *kamins42*
- Student Number: *1004431992*

Chedy Sankar:

- Email: *chedy.sankar@mail.utoronto.ca*
- Utorid: *sankarch*
- Student Number: *1004174895*

Alexander Greff:

- Email: *alex.greff@mail.utoronto.ca*
- Utorid: *greffal1*
- Student Number: *1004254497*

---

2. Work Distribution

Anton Kaminksy:

- Setup HTTP server and architecture of the Websocket server
- Implemented the chat app using our implemented Websocket server
- Implemented the receiving side of the Websocket server (parsing and unmasking data)

Chedy Sankar:

- Added support for multiple concurrent clients to the Websocket server using threading
- Implemented sending side of the Websocket server (assembling the frame and sending out data)
- Wrote documentation for the Websocket server code

Alexander Greff:

- Setup the Electron desktop client app used for testing/demoing
- Implemented the JavaScript Websocket client library
- Implemented the reference chat Websocket server

---

3. Implementation and Documentation

To implement both the client and the server in compliance with the WebSocket protocol, we followed the official spec found here: https://datatracker.ietf.org/doc/html/rfc6455.

3.1 WebSocket Client

The Websocket client was implemented in JavaScript and Node.js. There are two major parts of the client, the *WebSocketClient* and *WebSocketFrame* classes (located in *websocket-client/WebSocketClient.ts* and *websocket-client/WebSocketFrame.ts*, respectively). The *WebSocketClient* class handles instantiating the Websocket connection via a HTTP upgrade. *WebSocketClient* extends the *EventEmitter* class provided by Node.js which allows for users of the class to easily register and listen to any Websocket related events. Additionally, *WebSocketClient* handles the Websocket connection, receiving and sending messages, receiving and responding to ping-pong messages, and handling connection close events. *WebSocketClient* utilizes the *WebSocketFrame* class for managing the creation and loading Websocket frames.

A *WebSocketClient* instance exposes the following functions which can be used by the programmer:

Register listener for a Websocket event:

```
client.on("open", listener: () => void)
client.on("message", listener: (data: Buffer,
isBinary: boolean) => void)
client.on("close", listener: () => void)
client.on("ping", listener: (data: Buffer) =>
void)
client.on("pong", listener: (data: Buffer) =>
void)
client.on("error", listener: (error: Error) =>
void)
```

Unregister listener for a Websocket event:

```
client.off("open", listener: () => void)
client.off("message", listener: (data: Buffer,
isBinary: boolean) => void)
client.off("close", listener: () => void)
client.off("ping", listener: (data: Buffer) =>
void)
client.off("pong", listener: (data: Buffer) =>
void)
client.off("error", listener: (error: Error)
=> void)
```

Send message to server:

```
client.send(data: any)
```

Send ping/pong to server:

```
client.ping(data?: any)
client.pong(data?: any)
```

Close connection to Websocket server

```
client.close(code?: number, data?: string |
Buffer)
```

3.2 WebSocket Server

The Websocket server is implemented in python 3. We make use of built-in libraries and classes, namely *BaseHTTPRequestHandler*, *ThreadingHTTPServer*, *hashlib*, *multiprocessing*, & *socket* (we use 0 external dependencies). Our intention was to use an existing base HTTPServer class to solely handle websocket upgrade requests, and implement the rest of the WebSocket functionality thereafter. (Implementation can be found in the file *websocket-server/WebSocketServer.py*).

Our implementation starts off with the creation of a custom *BaseHTTPRequestHandler* that only handles and responds to websocket upgrade requests. This custom handler also keeps the socket that was kept alive from the clients request for our WebSocketServer to make use of.

Our main WebSocketServer class runs this HTTPHandler on a separate thread (through

the *ThreadingHTTPServer*, to handle multiple clients), and provides a callback function to retrieve the client socket connections. The callback initializes a *WebSocketConnection* class, which is where most of our implementation details can be found.

The WebSocketConnection class is used to handle all data going to/from a connected client. Here, we find methods for sending data, sending pong replies, close connection signals, and a listen method for incoming data. To simplify the data management, we also implemented a simple WebSocket Frame class (*WSFrame*) that has getters and setters to manipulate the contents of WebSocket Frames.

To use the server API, first create an instance of WebSocketServer, passing in the port number to run the server on, and a callback function which will be called with every new client connection. The function will get passed an instance of WebSocketConnection.

Simple WebSocketServer usage:

```
def handleMsg(text:str):
  print(text)
def connectionHandler(ws:WebSocketConnection):
  ws.onMessage(handleMsg)
port = 3051
wss = (port, connectionHandler)
```

Next, you need to provide callbacks for onClose and onMessage to the WebSocketConnection instance. You can do this inside the server handler function once you get a new instance of WebSocketConnection. Both of these callbacks will get a reference to the WebSocketConnection on which the messages originated, and the onMessage function will also get passed the data either as a string, or bytes depending on the opcode of the websocket frame.

And that's the whole API! Next you can use these functions in whatever way is appropriate for your specific application. Below, we show some snippets of a Chat Room WebSocketServer application using our implementation:

Chat Room WebSocketServer snippets:

```
class Room: # one chatroom, containing a list
of RoomMembers
   ...

class RoomMember: # one member represents a
websocket client connection
   ...

class Message: # a message sent by a
RoomMember, saved in a Room
   ...

def onMessage(text: Union[str, bytes], ws:
WebSocketConnection):
    """Called on every message sent to the
given websocket connection and data"""
    ...
    if (message["type"] == "room-connect"):
      ...
      ws.send(json.dumps(roomConnectedMsg))
    elif (message["type"] == "room-leave"):
      ...
    # rest of message sending logic is here
    # using ws.send() to send data

def onClose(ws):
    """Called when websocket connection is
about to be closed, cleanup"""
  ...

def connectionHandler(ws:
WebSocketConnection):
    """Called when new connection is
established just set up handlers"""
    ws.onMessage(onMessage)
    ws.onClose(onClose)


port = 3051
wss = WebSocketServer(port, connectionHandler)
```

The entire implementation can be found in *testing-app/server/chat-server.py*

4. Demo Instructions

4.1 Starting Demo

For this project, we set up a simple chat room Electron app to demonstrate the functionality of our Websocket client and server implementations. The app consists of two Websocket servers, one which uses our implemented Websocket server library while the other one uses the "ws" NPM package. These servers are connected to a client that uses Electron which gives a graphic user interface for interacting with the chat room.

To start up the demo, first start our Websocket chat server by running the following command in *testing-app/server*:

```
python3 chat_server.py
```

Next, in a new terminal with its current working directory at *testing-app/server-reference* start up the reference Websocket chat server with:

```
npm install
npm run start:dev
```

Now we must link our Websocket client NPM package with our Electron app so it can use it. First, in a new terminal in *websocket-client* run:

```
npm install
npm run create:link
```

Then in *testing-app/client* run:

```
npm install
npm run link:ws-client
```

Finally we can start the Electron client with:

```
npm run start:dev
```

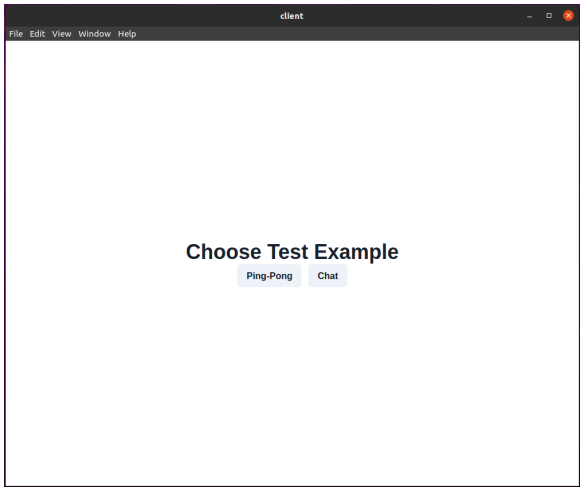
Note 1: you may need to run "npm run create:link" and "npm run link:ws-client" with sudo on Linux.

Note: multiple client windows can be opened up by instead running:

```
npm run start:dev-[num]
```

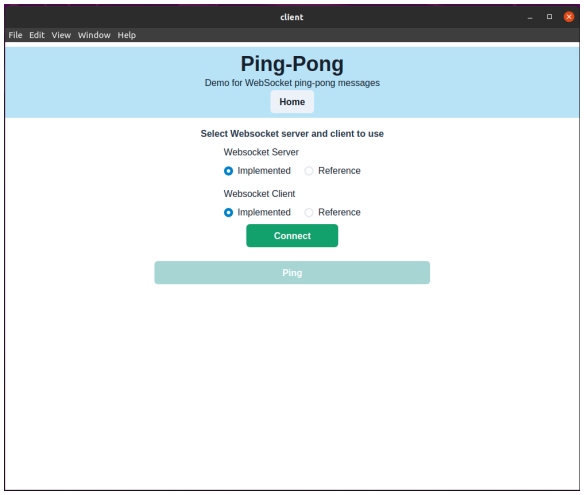
Where [num] is the number of windows to open between 1 and 4, inclusive.

The following window will be opened:



4.2 Ping-Pong Demo

From the main window opened in section 4.1, click on the "Ping-Pong" example button. This will open the following page:

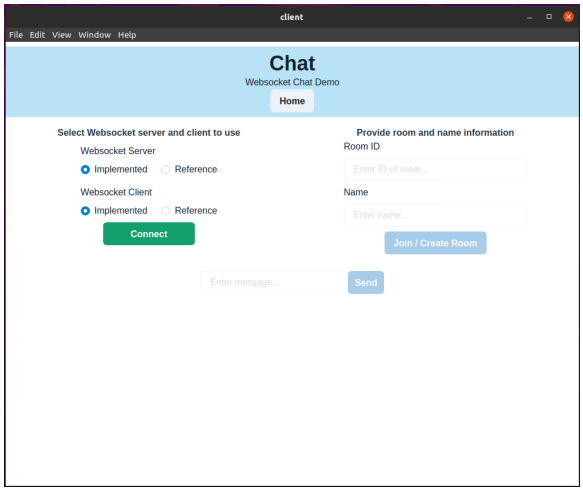


From here the Websocket server and client type can be selected and connected two. Once this is done, a ping can be sent to the server from the client by pressing the "Ping" button.

The corresponding pong received back from the server will be displayed as a toast on the page.

4.3 Chat Room Demo

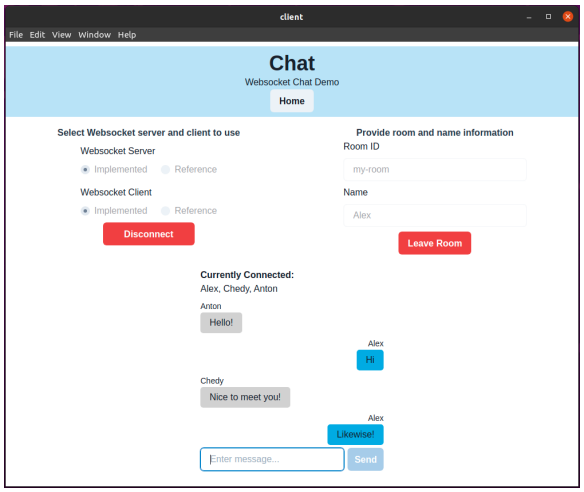
From the main window opened in section 4.1, click on the "Chat" example button. This will open up the following page:



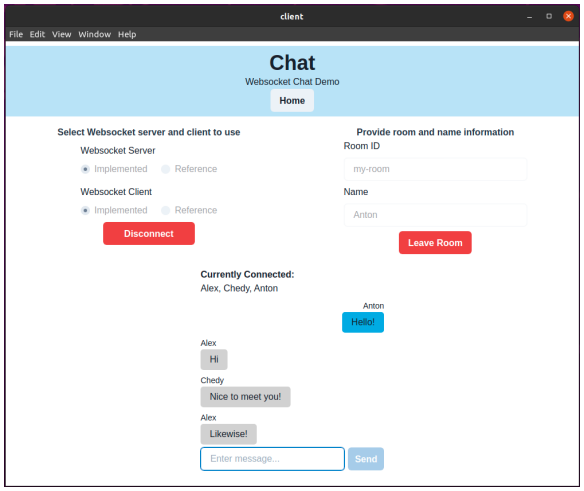
Like with the ping-pong example, connect to the Websocket client with the server. Then, you will be able to input a room ID to join as well as your name. After joining the room, you will be able to send and receive messages from all other clients in the room. To test the multiple client functionality, start up multiple windows (as described in section 4.1) and connect them all to the same room.

Below are images from an example chat log between users "Alex", "Anton" and "Chedy":

Alex's view:



Anton's view:



Chedy's view: