Creating a Signaling Server

Creating a Signaling Server

- At some point when creating a WebRTC application, you will have to break away from developing for a client and build a server.
- Most WebRTC applications are not solely dependent on just being able to communicate through audio and video and typically need many other features to be interesting.
- In this session, we are going to dive into server programming using JavaScript and Node.js.
- We are going to create the basis for a basic signaling server that we can utilize through the rest of the course.

Creating a Signaling Server

In this session, we will cover the following topics:

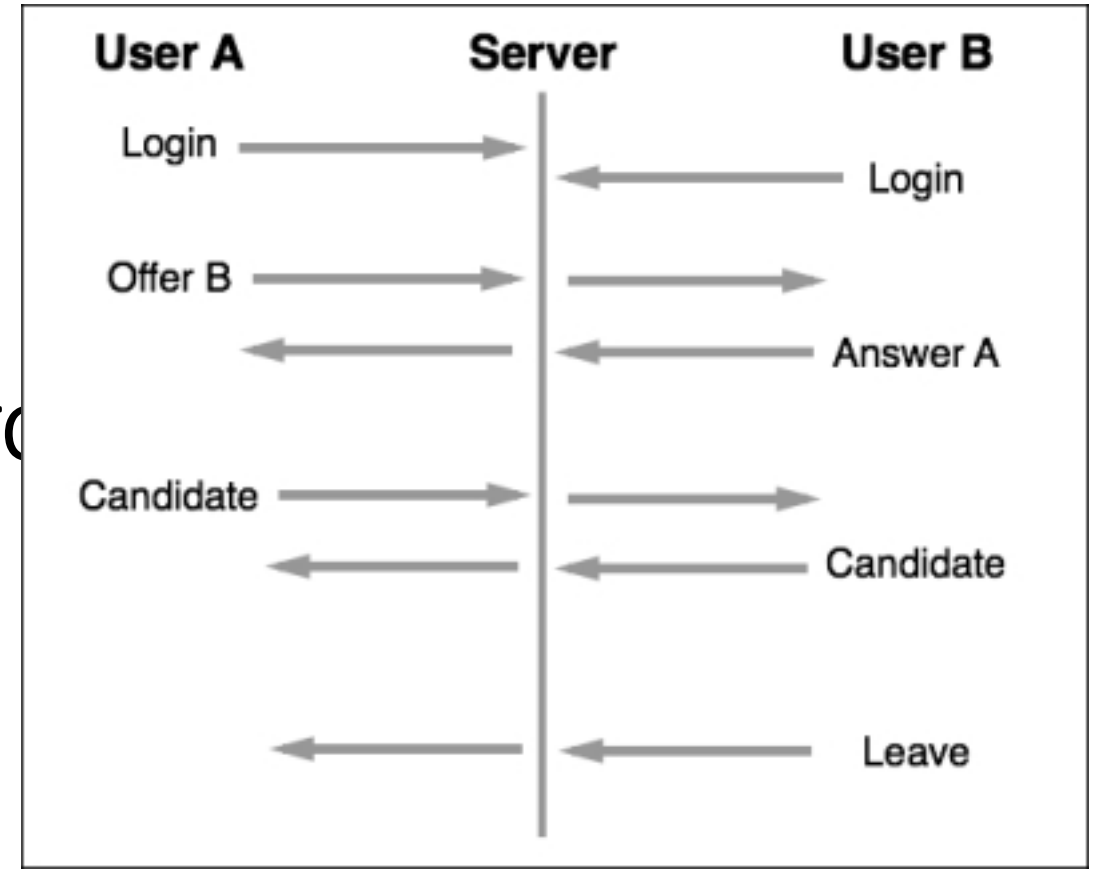
- Setting up our environment to develop in Node.js
- Connecting to the client using WebSockets
- Identifying users
- Initiating and answering a WebRTC call
- Handling ICE candidate transfers
- Hanging up a call

Building a signaling server

- The server we are going to build in this session will help us connect two users together who are not located on the same computer.
- The goal of the server is to replace the signaling mechanism with something that travels over a network.
- The server will be straightforward and simple, supporting only the most basic WebRTC connections.

Building a signaling server

- This will allow them to successfully setup a WebRTC connection.



Setting up our environment

- We are going to utilize the power of Node.js to build our server.
- If you have never programmed in Node.js before, do not worry! This technology utilizes a JavaScript engine to do all the work.
- This means that all of the programming will be in JavaScript so there will be no new language to learn.

Setting up our environment

Now, let's perform the following steps to set up our Node.js environment:

- The first step to running a node.js server is to install node.js.

Setting up our environment

- Now you can open up your terminal application and use the node command to bring up the Node.js VM.
- Node.js is based on the V8 JavaScript engine that comes with Google Chrome.
- This means that it works extremely close to how the browser interprets JavaScript.
- Type in a few commands to get used to how it works:

Refer to the file 4_1.txt

Setting up our environment

- From here, we can start creating our server program.
- Luckily, Node.js runs JavaScript files as well as commands typed in the terminal.
- Create an helloWorld.js file with the following contents and run it using the node helloWorld.js command:

```
console.log("Hello World from Node.js!");
```

- When you run the node index.js command, you will see the following output in the Node.js terminal:

```
C:\nodejs>node Chapter04\helloWorld.js
Hello World from Node.js!
```

Getting a connection

- The steps required to create a WebRTC connection are required to be real-time.
- This means that clients will have to be able to transfer messages between each other in real time without using a WebRTC peer connection.
- This is where we will utilize another powerful feature of HTML5 called WebSockets.
- A WebSocket is exactly what it sounds like—an open bidirectional socket connection between two endpoints—a web browser and a web server.

Getting a connection

- The big difference between the WebSocket protocol and the WebRTC protocol is the use of the TCP stack.
- WebSockets has been designed to be client-to-server in nature and utilizes TCP transport for a reliable connection.
- This means it has many of the bottlenecks that WebRTC does not have, which we described in Understanding UDP transport and real-time transfer section previously, Creating a Basic WebRTC Application.
- This is also the reason that it works well as a signaling transport protocol.

Getting a connection

- To install the library, navigate to the directory of the server and run the following command:

npm install -g ws
npm install -g wscat

- You should see the following output:

```
dan:webrtc-book/ (master) $ npm install ws
npm http GET https://registry.npmjs.org/ws
npm http 304 https://registry.npmjs.org/ws
npm http GET https://registry.npmjs.org/tinycolor
npm http GET https://registry.npmjs.org/options
npm http GET https://registry.npmjs.org/commander
npm http GET https://registry.npmjs.org/nan
npm http 304 https://registry.npmjs.org/tinycolor
npm http 304 https://registry.npmjs.org/options
npm http 200 https://registry.npmjs.org/commander
npm http 200 https://registry.npmjs.org/nan

> ws@0.4.31 install /Users/dan/workspace/webrtc-book/node_modules/ws
> (node-gyp rebuild 2> builderror.log) || (exit 0)

  CXX(target) Release/obj.target/bufferutil/src/bufferutil.o
  SOLINK_MODULE(target) Release/bufferutil.node
  SOLINK_MODULE(target) Release/bufferutil.node: Finished
  CXX(target) Release/obj.target/validation/src/validation.o
  SOLINK_MODULE(target) Release/validation.node
  SOLINK_MODULE(target) Release/validation.node: Finished
ws@0.4.31 node_modules/ws
├── tinycolor@0.0.1
├── options@0.0.5
├── commander@0.6.1
└── nan@0.3.2
```

Getting a connection

- Now that we have installed the WebSocket library, we can start using it in our server.
- You can insert the following code in our index.js file:

Refer to the file 4_2.txt

- The first line requires the WebSocket library that we installed in our previous command.
- We then create the WebSocket server, telling it what port to connect to listen on. You can specify any port you would like if you need to change this setting.

Getting a connection

- We then listen to any messages that are being sent by the user.
- For now, we just log these messages to the console.
- Finally, we send a response to the client saying Hello World.
- This happens immediately when the server has completed the WebSocket connection with the client.

Getting a connection

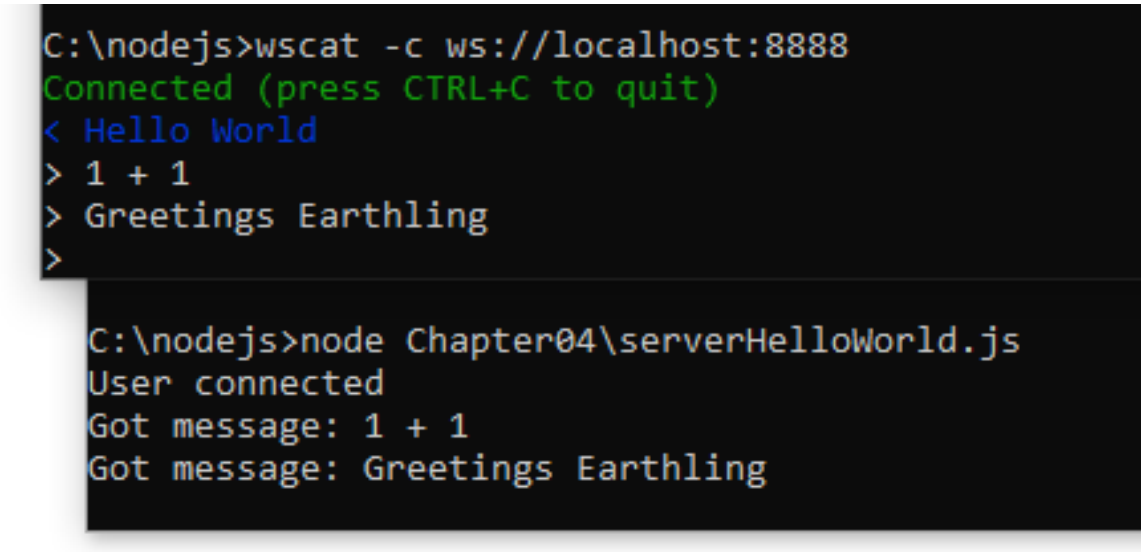
Testing our server

- To test whether our code is functioning properly, we can use the wscat command that comes with the ws library.
- The great thing about npm is that you cannot only install libraries to use in your application, but also install libraries globally to be used as command-line tools.
- The way to do this is by running npm install -g ws, although you might need to use administrator privileges when running this command.

Getting a connection

- You will notice ws://, which is the custom directive for the WebSocket protocol instead of HTTP.

- Your output should look similar to this:



Getting a connection

- Your server should also log the connection to its console:

Identifying users

- In a typical web application, the server will need a way to identify between connected clients.
- Most applications today use the one-identity rule and have each user login to a respective string-based identifier known as their username.
- We will also be using the same rule in our signaling application.

Identifying users

- To start, we are going to change our connection handler a bit, to look similar to this:

Refer to the file 4_3.txt

Identifying users

- We can change the top of our file to look similar to this:

```
var WebSocketServer = require(''ws'').Server,
    wss = new WebSocketServer({ port: 8888 }),
    users = {};
```

Identifying users

- This will allow our server to know what to do with the data that it is receiving.
- Firstly, we will define what to do when the user tries to login:

Refer to the file 4_4.txt

Identifying users

- I also added a helper function called sendTo in the code that handles sending a message to a connection.

- This can be added anywhere in the file:

```
function sendTo(conn, message) {
  conn.send(JSON.stringify(message));
}
```

Identifying users

- The last thing we have to do is provide a way to clean up client connections when they disconnect.
- Luckily, our library provides an event just when this happens. We can listen to this event and delete our user in this way:

```
connection.on(''close'', function () {
    if (connection.name) {
      delete users[connection.name];
    }
  });
```

Identifying users

- Once we connect, we can send the following message to our server:

{ ""type"": ""login"", ""name"": ""Foo"" }

- The output you receive should look similar to this:

```
dan:webrtc-book/ (masterX) $ wscat -c ws://localhost:8888
connected (press CTRL+C to quit)
> { "type": "login", "name": "Foo" }
< {"type":"login","success":true}
>
```

Initiating a call

- From here on, our code does not get any more complex than the login handler.
- We will create a set of handlers to pass our message correctly for each step of the way.
- One of the first calls that is made after logging in is the offer handler, which designates that one user would like to call another.
- It is a good idea not to get call initiations mixed up with the offer step of WebRTC.

Initiating a call

- We can now add the offer handler into this code:

Refer to the file 4_5.txt

- The first thing we do is get connection of the user we are trying to call.
- This is easy to do since the ID of the other user is always where our connection is stored in our user-lookup object.
- We then check if the other user exists and if so, send them the details of offer

Answering a call

- Answering the response is just as easy as offer.
- We follow a similar pattern and let the clients do most of the work.
- Our server will simply let any message pass through as answer to the other user.
- We can add this in after the offer handling case:

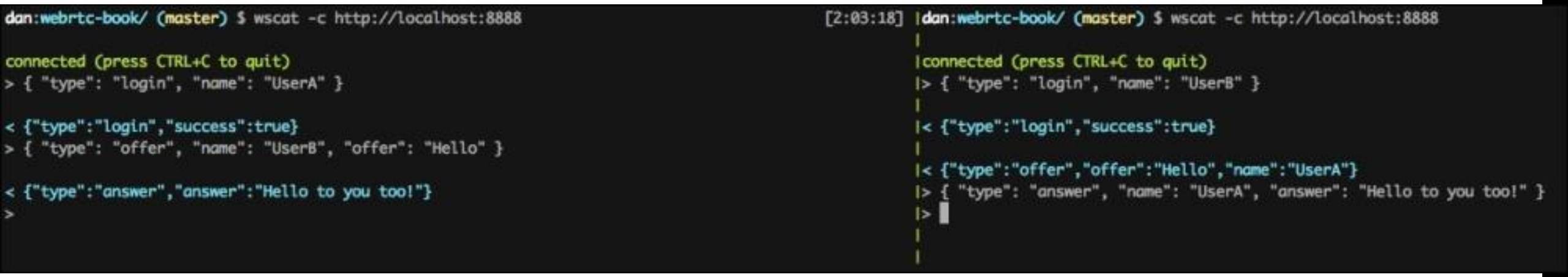
Refer to the file 4_6.txt

Answering a call

- We can even test our current implementation using the WebSocket client we used before.
- Connecting two clients at the same time allows us to send offer and response between the two.
- This should give you more insight into how this will work in the end.

Answering a call

- You can see the results from running two clients simultaneously in the terminal window, as shown in the following screenshot:



Handling ICE candidates

- The final piece of the WebRTC signaling puzzle is handling ICE candidates between users.
- Here, we use the same technique as before to pass messages between users.
- The difference in the candidate message is that it might happen multiple times per user and in any order between the two users.
- Thankfully, our server is designed in a way that can handle this easily.

Handling ICE candidates

- You can add this candidate handler code to your file:

Refer to the file 4_7.txt

- Since the call is already set up, we do not need to add the other user's name in this function either.

Hanging up a call

- Our last bit is not part of the WebRTC specification, but is still a good feature to have—hanging up.
- This will allow our users to disconnect from another user so they are available to call someone else.
- This will also notify our server to disconnect any user references we have in our code.
- You can add the ""leave"" handler as detailed in the following code:

Refer to the file 4_8.txt

Hanging up a call

- We can change the close handler we used before to look similar to this:

Refer to the file 4_9.txt

- This will now disconnect our users if they happen to terminate their connection unexpectedly from the server.

Complete signaling server

- Here is the entire code for our signaling server.
- This includes logging in and handling all response types.
- I also added a listening handler at the end to notify you when the server is ready to accept WebSocket connections:

Refer to the file 4_10.txt

Signaling in the real world

- It has taken us a lot of effort to get to a basic signaling server to connect two WebRTC users.
- At this point, you may be wondering how signaling servers are built in the real world for production applications.
- Since signaling is such an abstract concept that is not defined by the WebRTC specification, the answer is that anything goes.

Signaling in the real world

The woes of WebSockets

- The great thing about WebSockets is that it has brought bidirectional communication to browsers.
- Many consider WebSockets to be the answer to all their problems, enabling faster socket connections directly to servers.
- This being said, there are still a few wrinkles to iron out in the WebSocket space.

Signaling in the real world

Connecting with other services

- One of the most exciting parts of WebRTC is that it will not only work well as a standalone solution, but also pairs well with other technologies.

- There have been numerous peer connectivity applications before WebRTC came around and since its introduction, efforts have been put forward to make WebRTC backward compatible.

Signaling in the real world

XMPP

- XMPP is an instant messaging protocol that dates back to the 90's under the name Jabber.

- The protocol was aimed at defining a common way to implement instant messaging, user presence, and contact lists. It is an open standard which anyone can use and integrate into their application.

Signaling in the real world

Session Initiation Protocol

- The Session Initiation Protocol (SIP) is another standard dating back to the 90's.
- It is a signaling protocol that has been designed targeting cellular networks and phone systems.
- It is a well-defined and extremely well-supported protocol seen in use by major cell networks and network equipment providers.

Summary

- Over the course of this session, we covered each step of the signaling process.
- We walked through setting up a Node.js application, identifying users, and sending the entirety of offer/answer mechanisms between users.
- We also detailed disconnecting, leaving connections, and sending candidates between users.

Summary

- You should now have a firm understanding of how a signaling server works.
- The server that we built is built to be simple and straightforward from a pure learning standpoint.
- We could fill an entire course up with new features that we could add to our server, such as authentications, buddy lists, and more. If you are feeling adventurous, feel free to add as many functionalities as you would like to our implementation.