COMP4021
Internet Computing

Using WebSocket

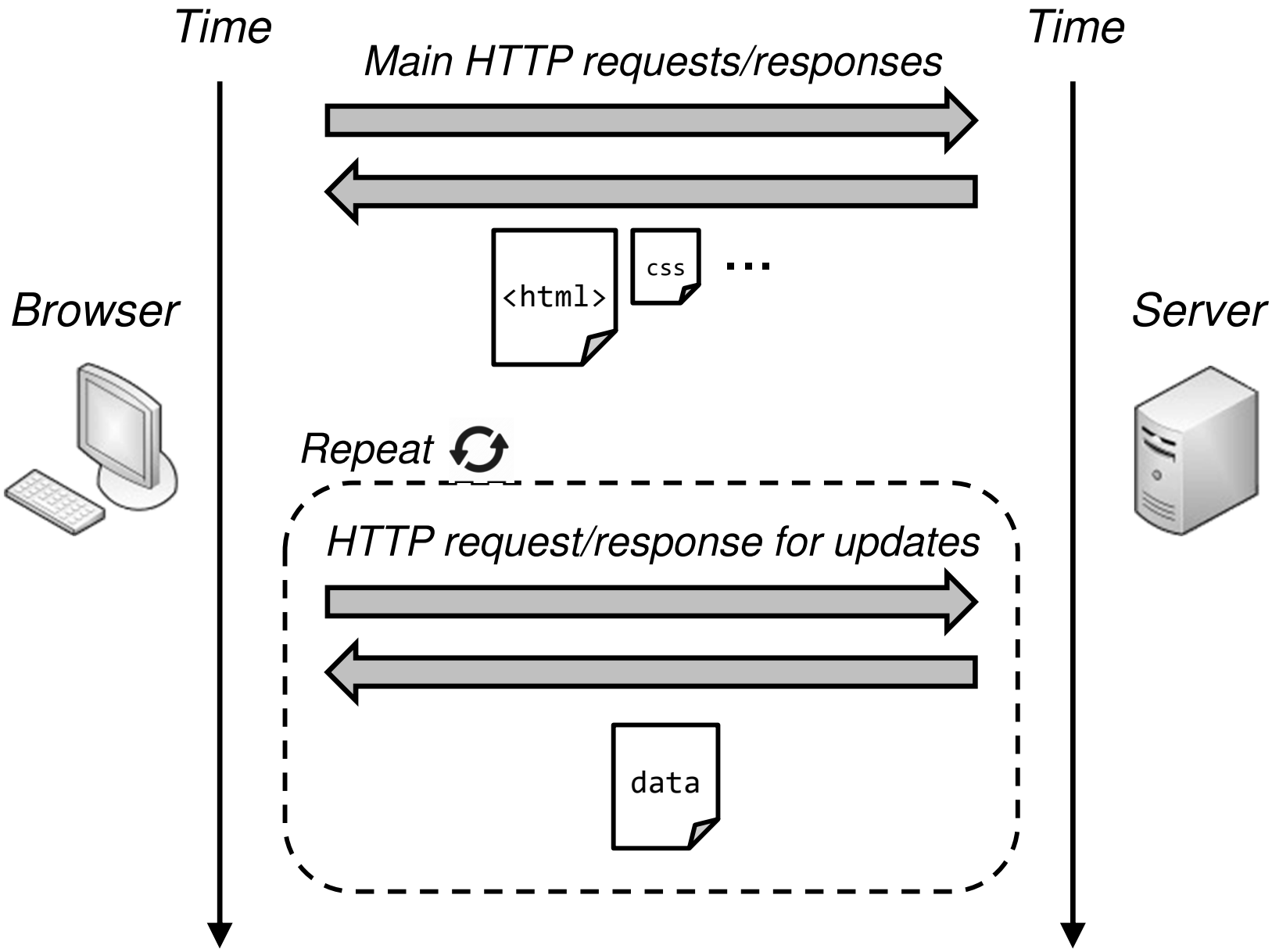
Gibson Lam

Updating a Web Page

- Let's imagine you need to make a web page to show some stock data in *real time*

- Based on what we have learned so far, you may use `fetch()` to continuously get the updated information from the server, e.g. using `setTimeout()`
- This is called *client pull* as the update is always initiated on the browser side

Client Pull

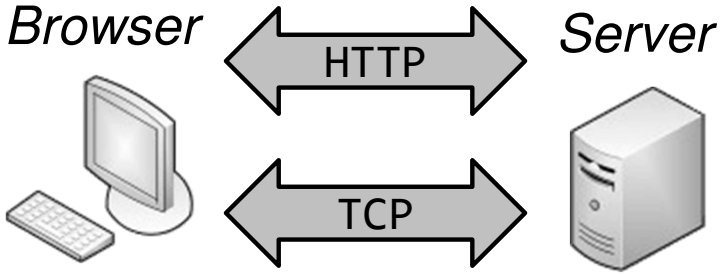

Time
Time
Main HTTP requests/responses
Browser
Server
<html>
css
...
Repeat
HTTP request/response for updates
data

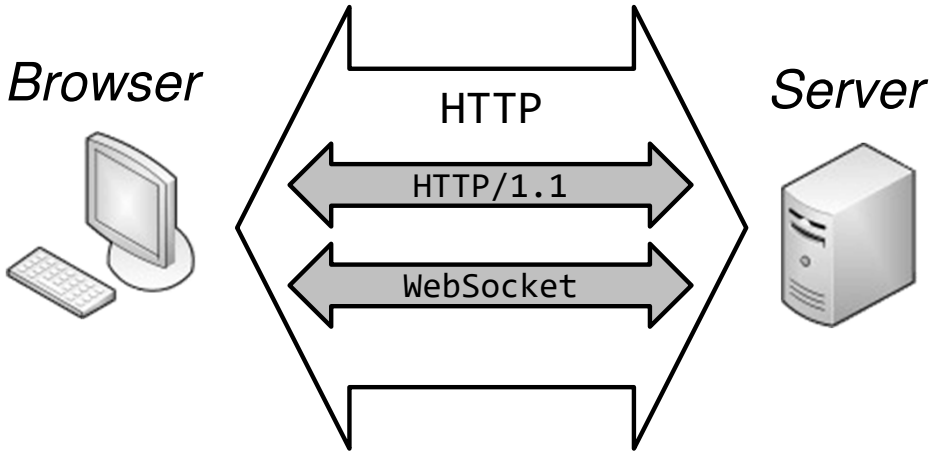
Issues With Client Pull

1. If the data changes frequently, the browser needs to pull data from the server with an even higher frequency to keep updated
   - Sometimes even if the data has not been changed on the server side!
2. The HTTP request/response contain significant overhead
3. The server cannot initiate anything, it can only wait for the browsers

Solution – Using WebSocket

- One way to solve those issues is by making a new network connection, e.g. on TCP, using a different port
- Or better, use *WebSocket*, which is essentially a new kind of connection but runs on HTTP





Initiating WebSocket Protocol

- WebSocket uses a protocol named `ws://`, or `wss://` (secure)
- At the start, it sends an ordinary HTTP request with additional HTTP headers:

```
...
Connection: Upgrade
Upgrade: websocket
...
```

- The headers request the server to ‘upgrade’ the connection to a WebSocket connection

Server Response

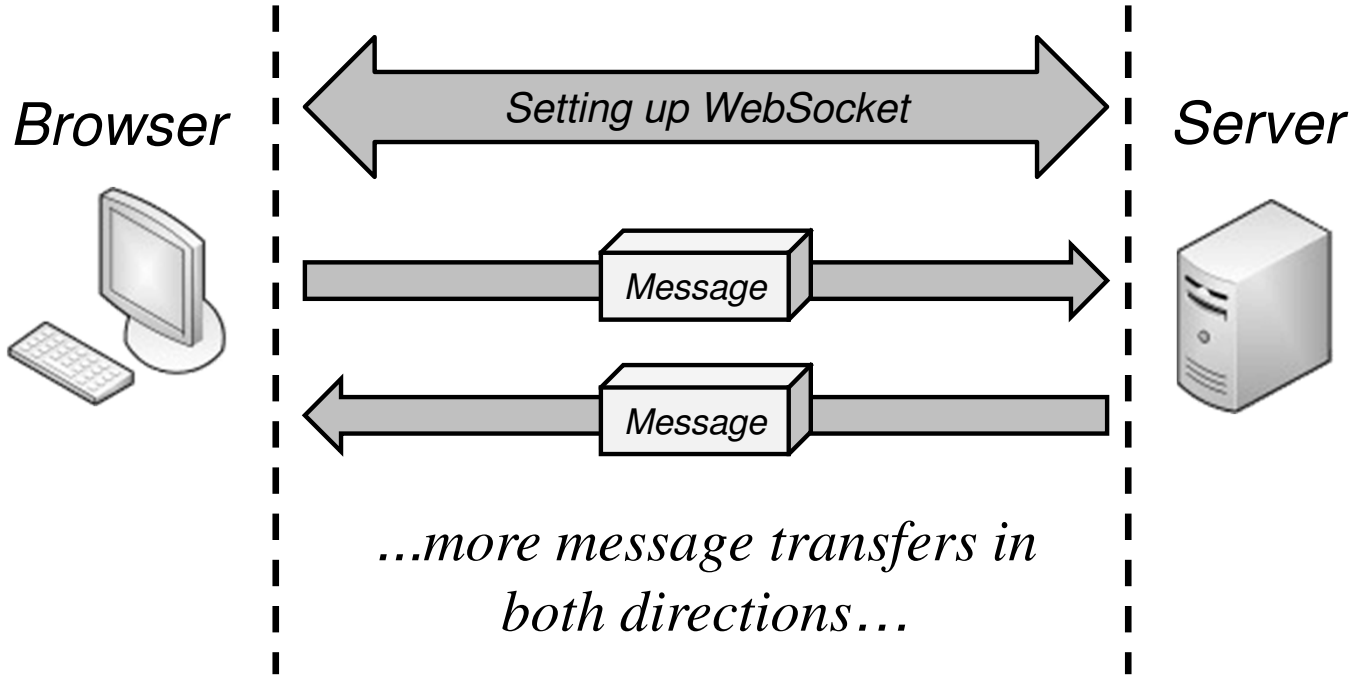
- If the server supports WebSocket, it will reply with a `101`, switching protocols:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
...
```

- Then, the connection will not close and become a WebSocket connection

Message Transfer

- Once the WebSocket connection is established, the browser/server can start to transfer messages bidirectionally



Using WebSocket in Node.js

- You can use the Socket.IO library to work with WebSocket
- Simply install the package using `npm`:

```
C:\Users\Gibson> npm install socket.io
```

- Programming in Socket.IO is easy as its code is mostly event-based
- However, it needs to modify the Express app, as shown on the next slide

Creating the Socket.IO Server

- Socket.IO can work with an Express server, but you need to add the following:

```
const { createServer } = require("http");
const { Server } = require("socket.io");
const httpServer = createServer( app );
const io = new Server(httpServer);
```

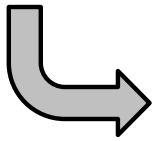
This is the Socket.IO server

This is an Express server app

Running the Web Server

- Since the web server now comes from the http module, you also need to adjust the code for starting the server:

~~`app.listen(8000);`~~ *Important!*



```
httpServer.listen(8000);
```

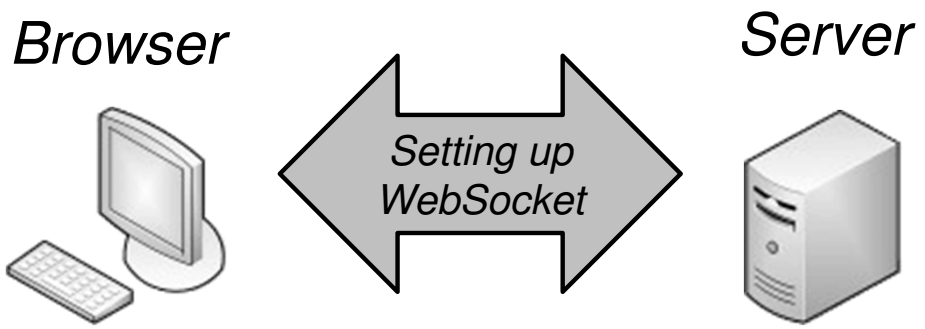
- This would finish the initialization of the Socket.io module and you are ready for new WebSocket connections

Using WebSocket on the Server

- Let's take a look at four things you can do on the server when using Socket.IO :
  - Waiting for new connection
  - Sending messages
  - Waiting for messages
  - Waiting for disconnection

Waiting for Connections

- You use the Socket.IO server to wait for new connections:

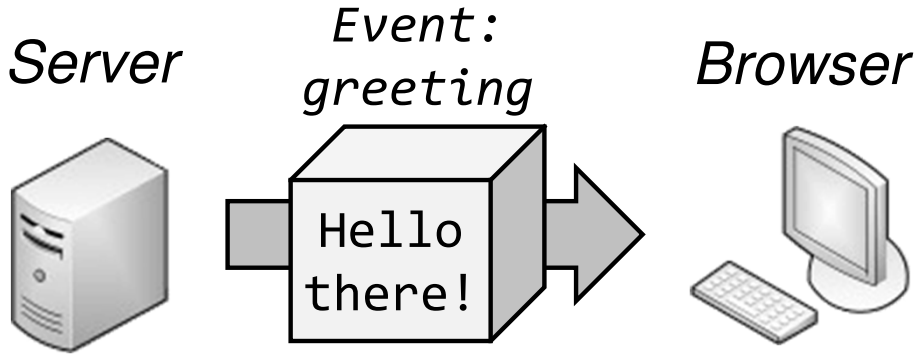



```
io.on("connection", (socket) => {
    ...Doing things for the
        connected browser...
}
```

You use this socket variable to work with the connected browser

Sending Messages

- Once a browser is connected, you can send messages to it using the socket
- Messages are sent as events, with any event name that you can think of
- Here is an example sending a message 'Hello World!' in an event called 'greeting':



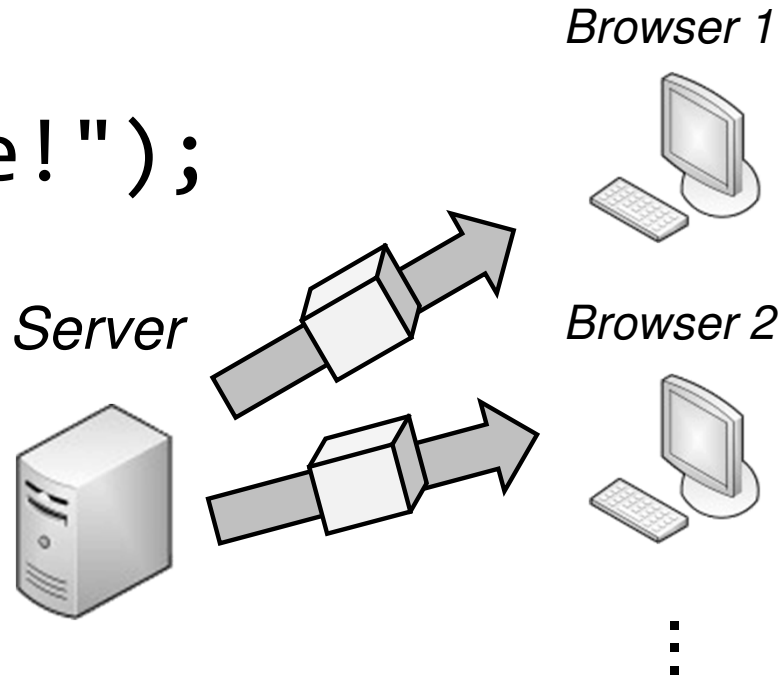

```
socket.emit("greeting",
                    "Hello there!");
```

Broadcasting

- A very useful feature in Socket.IO is the ability to broadcast messages
- Instead of a socket, you use the Socket.IO server to send a message, e.g.:

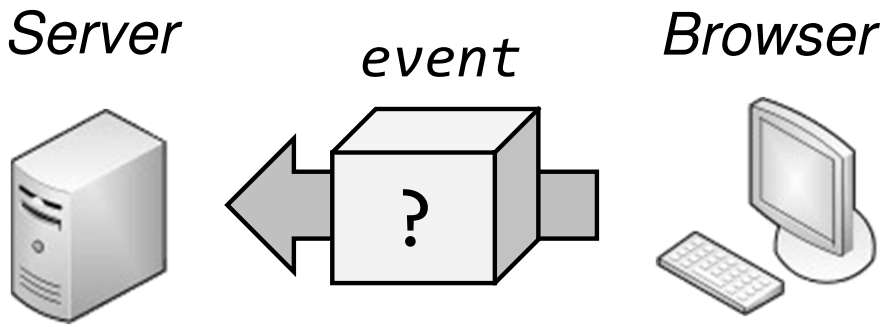
```
io.emit("greeting",
        "Hello there!");
```

- Then every browser that has connected to the server would receive the message



Receiving Messages

- You use a browser's socket to wait for messages received from that browser



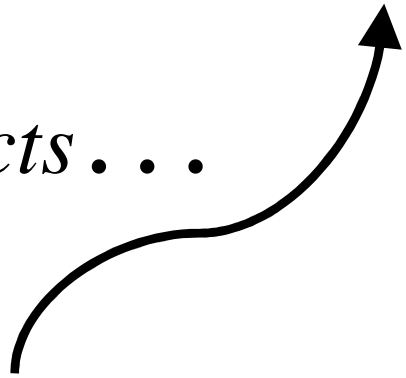
- Messages are identified by the event name that is used when they are sent
- For example, this code waits for a message from the browser in a 'greeting' event:

```
socket.on("greeting", (message) => {
    ...Doing things for the message...
}
```

Waiting for Disconnection

- Finally, the server can listen for the event when a browser disconnects from it:

```
socket.on("disconnect", (reason) => {
    ...Doing things when
        the socket disconnects...
}
```



- You can check the reason for the disconnection if you really want to

Socket.IO on the Browser

- Using Socket.IO on the browser is easy
- After enabling Socket.IO on the server, its JavaScript file is automatically available through this path:

```
/socket.io/socket.io.min.js
```

- You can then link to the JavaScript file using:

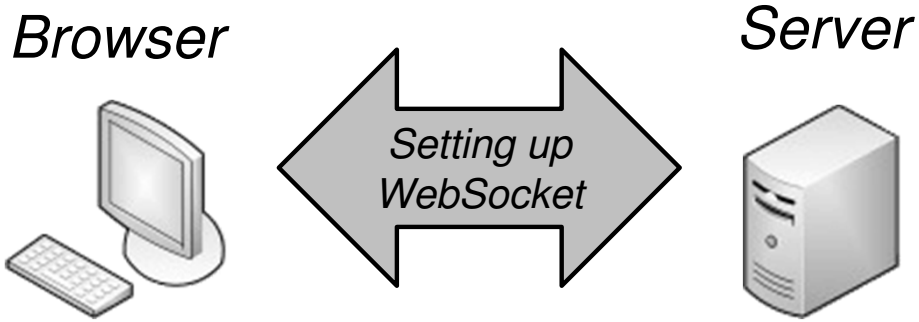
```
<script src="/socket.io/socket.io.min.js">
</script>
```

Using WebSocket on the Browser

- Similar to what we have done using WebSocket on the server
- Let's take a look at four things you can do on the browser this time:
  - Connecting to the server
  - Sending messages
  - Waiting for messages
  - Disconnecting from the server

Connecting to the Server

- You create Socket.IO and connect to the server at the same time using this code:




```
const socket = io();
```

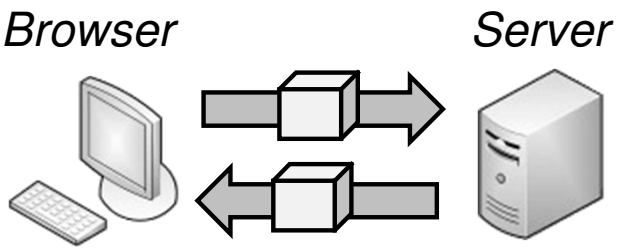
You use this socket variable to talk to the server

- The above code assumes you are connecting to the same domain of the web page containing the code, which we do most of the time

Sending/Receiving Messages

- The ways to send and receive messages are the same as what you have done before, i.e.:

```
socket.emit("greeting",
                    "Hello!");
```



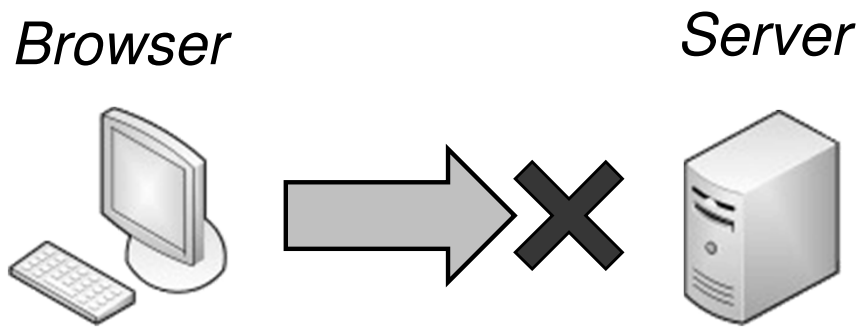
and:

```
socket.on("greeting", (message) => {
    ...Doing things for the message...
}
```

- But you cannot broadcast from the browser

Disconnecting from the Server

- You disconnect the browser from the server using `disconnect()`, as shown below:

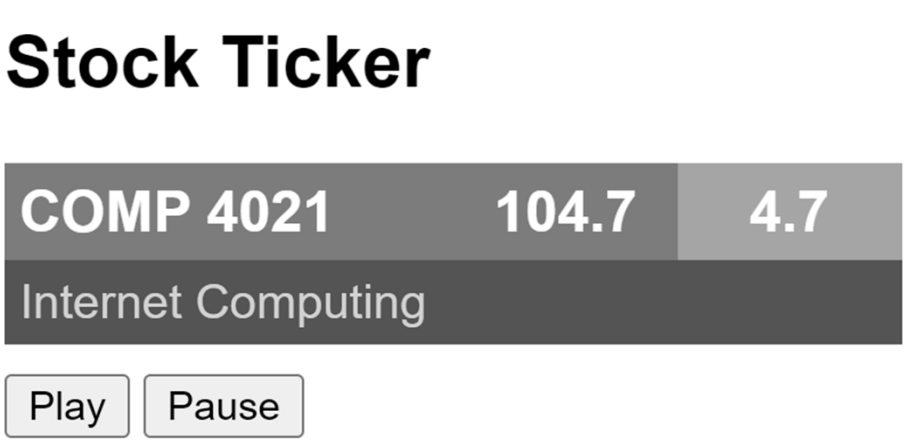



```
socket.disconnect();
```

- After running the above code, the server will then receive the ‘disconnect’ event

An Example Web Application

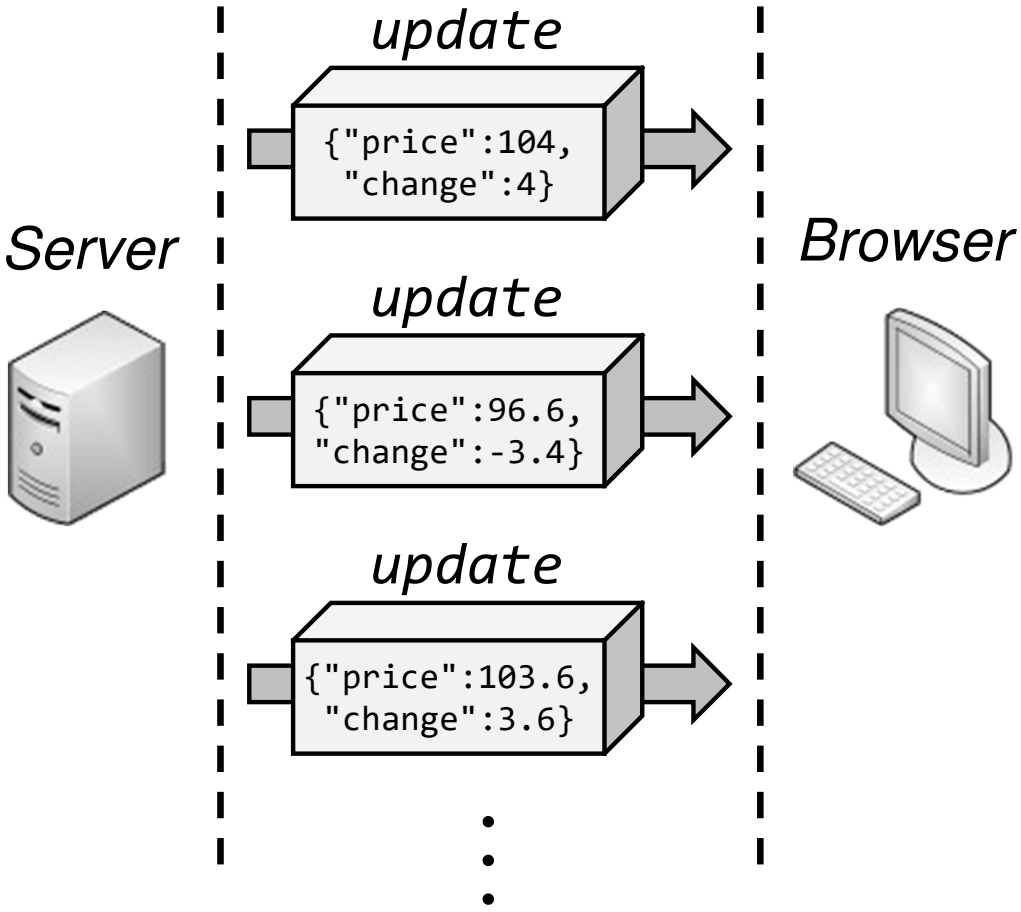
- A ‘stock ticker’ application has been created using WebSocket

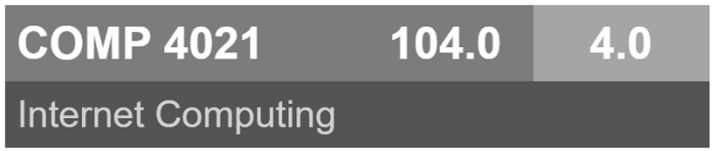


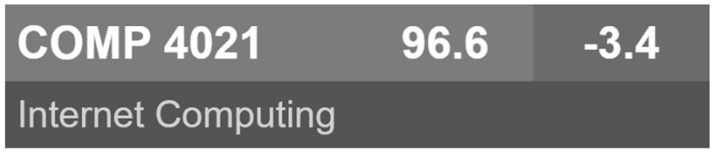
- The example continuously updates the stock price of the stock ‘COMP 4021’
- The updated price is sent from the server using WebSocket

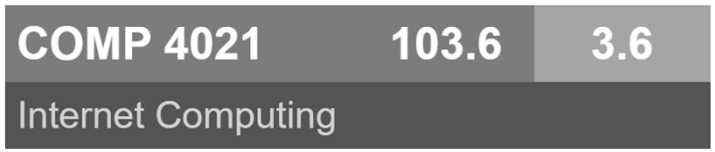
Updating the Price

- The price is updated every second from the server to the browser in a JSON message









Updating the Price from Server

- A function called `ticker()` keeps on updating the price by sending a message to the browser

```
const ticker = function() {
  if (playing && socket.connected) {
    ...
    const data = { price, change };
    socket.emit("update", JSON.stringify(data));

    timeout = setTimeout(ticker, 1000);
  }
};
```

Send the price as JSON to the browser

This function run every second

Updating the Price on Browser

- Then, the browser listens for the updated price and shows it in a table appropriately

```
socket.on("update", (data) => {
  const { price, change } = JSON.parse(data);

  $("#price").text(price.toFixed(1));
  $("#change").text(change.toFixed(1));

  ...
});
```

Display the number fixed with 1 digit after the decimal point

Parse the JSON data

Controlling the Update

- In addition, two buttons are used to control the price update
- It is achieved by sending a simple text message from the browser to the server


```
$("#play").on("click", () => {
  socket.emit("command", "play");
});

$("#pause").on("click", () => {
  socket.emit("command", "pause");
});
```

A simple 'command' is sent to the server

Over on the Server-Side

- After sending the command to the server, the server can process it easily using this code:

```
socket.on("command", (command) => {
    if (command == "play") {
        playing = true;
        clearTimeout(timeout);
        ticker();
    }
    else
        playing = false;
});
```

Start the update

Stop the update