International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 07 Issue: 05 | May 2020 www.irjet.net p-ISSN: 2395-0072

Scaling WebSocket Connections using Shared Workers

Shivam Kumar[1]

[1]*Dept. of Information Technology, Maharaja Agrasen Institute of Technology, New Delhi, India*

Abstract - *WebSocket is a communication protocol which allows full-duplex communication between the two parties i.e. server and client. This full duplex communication is over a single TCP protocol. In order to facilitate this communication, each client is required to open a connection to the server and keep it alive till the client closes the browser tab/goes offline. In order to facilitate this persistent connection, a new connection is needed on every browser tab of a single client which introduces scalability issues. This paper aims to tackle the issue using Shared Workers and reduce the load of multiple connections on a client.*

Key Words: **Web sockets, Real-time communication, Duplex communication, Web Workers, Shared Workers, Broadcast channels, Scalability.**

1. INTRODUCTION

WebSocket is a communication protocol which allows real time communication to be established between the server and the client.

A WebSocket connection is initialized from a HTTP request, but the connection itself is not based on HTTP. It is layered over the TCP. The connection is established by an opening handshake initiated by the client that is a simple HTTP request. It includes the special header "Upgrade: websocket" indicating that the client would like to upgrade the connection to the WebSocket protocol. [4]

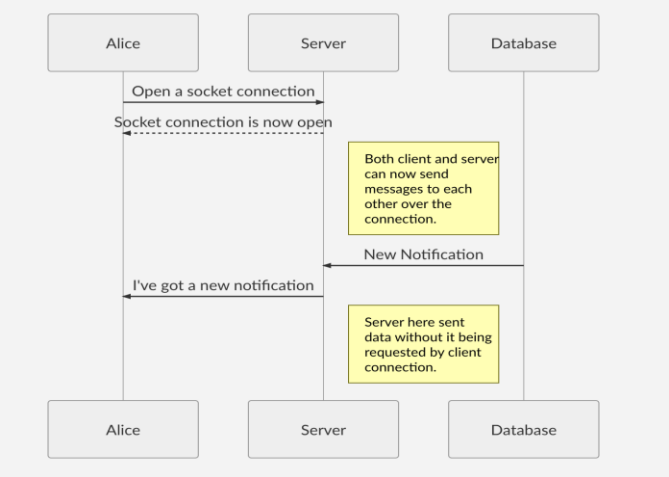


Fig -1: Realtime communication between server and client

The client first opens a dedicated socket connection over TCP protocol. Once the server responds, the connection is persisted throughout the online activity. Since the connection is bi-lateral, both the client and server have the ability to send new messages to each other.

This connection is disconnected only when the client goes offline or closes the browser tab.

2. ISSUES WITH SCALABILITY OF WEBSOCKETS

In order to facilitate the full duplex communication, each client needs to open a connection with the server and keep it alive till the time client goes offline/ closes the tab. This persisted connection makes the interaction stateful, leading both the sides of communication to maintain some amount of data in memory.

The issue of scalability arises in scenarios where a client opens multiple tabs of the same web application which is maintaining a WebSocket connection. So, if a client has 15 tabs open, they will have 15 open connections to the server.

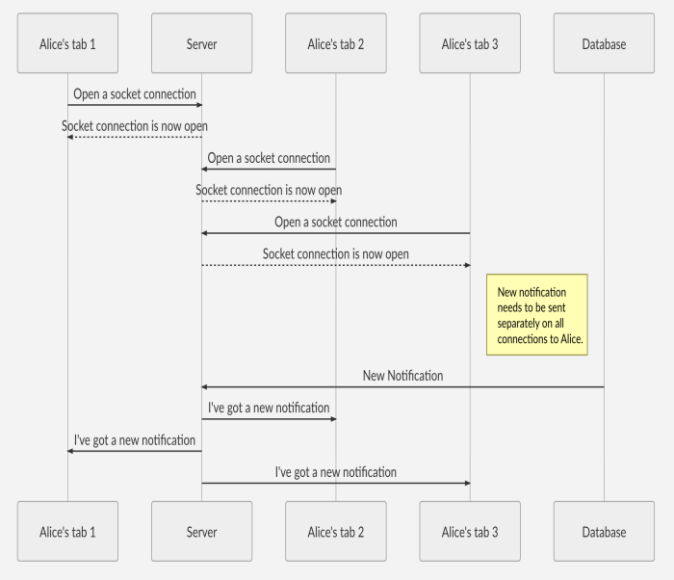


Fig -2: Socket communication with multiple contexts.

3. OVERVIEW OF WEB WORKERS, SHARED WORKERS, BROADCAST CHANNELS

3.1 WEB WORKERS

They are a simple means for web content to run scripts in background threads. The worker thread can perform tasks without interfering with the user interface. Once created, a worker can send messages to the JavaScript code that created

it by posting messages to an event handler specified by that code (and vice versa). [1]

3.2 SHARED WORKERS

They are a type of web workers that can be accessed from several browsing contexts, such as several windows, iframes or even workers. [2]

3.3 BROADCAST CHANNELS

They are used to allow simple communication between browsing contexts (i.e. windows, tabs, frames or iframes) with the same origin. [3]

4. REDUCE SERVER LOADING USING WORKERS

The Shared Workers can be used for solving the issue of scalability i.e. the problem of a single client having multiple connections open from the same browser. The idea is to use a Shared Worker to open the socket connection with the server instead of every browser tab/window. We can utilize the broadcast channel API to broadcast state change of web socket to all the contexts (tab/window).

Thus, allowing us to communicate with and receive messages from the server from any of the contexts. This shared connection would be open until all the tabs of the same origin/website are closed.

5. IMPLEMENTATION OF SERVER AND SHAREDWORKERS

5.1 IMPLEMENTATION OF WEBSOCKET SERVER

The implementation on server side is same as a typical web socket supporting server. Below is the code for a standard express and WebSocket implementation of a server:

```
const express = require("express");
const path = require("path");
const WebSocket = require("ws");
const app = express();

// for static file requests
app.use(express.static("public"));

// Start our WS server at 5000
const wss = new WebSocket.Server({
    port: 5000
});
wss.on("connection", ws => {
  console.log('Client connected!');
  ws.on("message", data => {
    console.log(`Message: ${data}`);

    // Modify the input
    // and return the same.
    const parsed = JSON.parse(data);
    ws.send(
      JSON.stringify({
        ...parsed.data,
        messageFromServer: `Hello
         tab id: ${parsed.data.from}`
      })
    );
  });
  ws.on("close", () => {
    console.log("Client disconnected");
  });
});
// Listen at 3000
app.listen(3000,() => {
  console.log("Running");
});
```

Fig -3: Code for express and WebSocket server.

5.2 IMPLEMENTATION OF SHAREDWORKER

According to create any type of Worker in JavaScript, a separate file needs to be created that houses the logic which would be executed by the worker. Within this worker file, we can define whatever logic we want to execute when this worker gets initialized. After that until the last tab connecting to this worker to this worker is not closed/ends connection with this worker, this code cannot be re-run.

We can use the "on connect" event handler to handle each tab connecting to this Shared Worker. Our worker.js file will look something like this:

```
// Open a common connection.
const ws =
   new
WebSocket("ws://localhost:2001");

// Create a broadcast channel
// to notify about state changes
const bChannel =
  new
BroadcastChannel("WebSocketChannel");

// Mapping to keep track of ports.
// Consists of uuid assigned to
context.
// This is needed because Port API
// does not have any identifier we can
// use to identify messages coming
from it.
const  idToPortMap  = {};

// Connected contexts know state
changes.
ws.onopen = () =>
  bChannel.postMessage({
   type: "WSState",
   state: ws.readyState
});
ws.onclose = () =>
  bChannel.postMessage({
   type: "WSState",
   state: ws.readyState
});

// When we receive data from the
server.
ws.onmessage  = ({ data }) => {
  console.log(data);
  // Construct object to be passed
  const parsedData = {
   data:  JSON.parse(data),
   type:  "message"
  }
  if (!parsedData.data.from) {
    // Broadcast to all contexts(tabs)
    bChannel.postMessage(parsedData);
  } else {
    // Get the port to post to using
    // uuid, ie send to expected tab.
    idToPortMap[parsedData.data.from]
     .postMessage(parsedData);
  }
};
```

```
// Event handler called when a tab
// tries to connect to this worker.
onconnect = e => {
  // Get the MessagePort from the
event.
  // This will be the
  // communication channel between
  // SharedWorker and the Tab
  const  port  =  e.ports[0];
  port.onmessage  =  msg  => {
    // Collect port information in the
map
    idToPortMap[msg.data.from] =
port;

    // Forward this message to
    //  ws connection.
    ws.send(JSON.stringify({
     data:  msg.data
    }));
  };

  // We need this to notify the
  // newly connected context to know
  // the current state of WS
connection.
  port.postMessage({
   state: ws.readyState,
   type: "WSState"
  });
};
```

Fig -4: Code for worker.js file

There are few different things we have done here. The following points try to clarify what we have done:

- We used broadcast channel API to broadcast the state changes of the socket.
- We used "post Message" to the port on connection to set the initial state of the context(tab).
- We used the "from" field from the context themselves to identify where the redirect the messages.
- If we do not have a "from" field set from the message coming from the server, we broadcasted it to every context(tab).

5.3 CONSUMING THE SHAREDWORKER

In order to consume our SharedWorker we create a "main.js" file. This JS file holds the logic for consuming the SharedWorker which houses the WebSocket connection. We import this JS file in our client-side html file.

Our "main.js" file will look something like this:

```
// Create a SharedWorker Instance
using our worker.js file.
const worker =
  new SharedWorker("worker.js");

// Create a unique identifier.
const id = uuid.v4();

// Set initial web socket state
let  webSocketState =
   WebSocket.CONNECTING;

console.log(`Initialize worker for:
    ${id}
`);

// Connect to the shared worker
worker.port.start();

// Set an event listener
worker.port.onmessage = event => {
  switch (event.data.type) {
    case "WSState":
      webSocketState =
       event.data.state;
      break;
    case "message":

handleMessageFromPort(event.data);
      break;
  }
};
// Set up the broadcast channel to
listen to web socket events.
const bChannel =
  new
BroadcastChannel("WebSocketChannel");

bChannel.addEventListener("message",
 event => {
  switch (event.data.type) {
    case  "WSState":
      webSocketState  =
        event.data.state;
      break;
    case  "message":
      handleBroadcast(event.data);
      break;
  }
});
```

```
// Listen to broadcasts from server
function handleBroadcast(data) {
  console.log("Message for everyone!");
  console.log(data);
}

// Handle event only meant for this tab
function handleMessageFromPort(data) {
  console.log(`only for user: ${id}`);
  console.log(data);
}

// Use this method to send data
// to the server.
function postMsgToWS(input) {
  if (webSocketState ===
       WebSocket.CONNECTING) {
    console.log("Still connecting!");
  } else if (
    webSocketState ===
WebSocket.CLOSING ||
    webSocketState  ===
WebSocket.CLOSED
  ) {
    console.log("Connection Closed!");
  } else {
    worker.port.postMessage({
      // Include the sender information
      // as a uuid to get back the
response
      from:  id,
      data:  input
    });
  }
}

// Sent a message to server after
// approx 2.5 sec. This will give
enough
// time to web socket connection.
setTimeout(() =>
    postMsgToWS("Initial message"),
2500);```
```

Fig -5: Code for client side main.js using SharedWorkers

6. SENDING MESSAGES TO SHARED WORKERS

As we saw above, we can send messages to this SharedWorker using "worker.port.postMessage()" method.

A good practice here can be passing an object that contains the context from which the message is coming so that worker can act accordingly. So, for example, if we have a chat app where one of the tabs wants to send a message, we can use something a JS object something like this:

```
// Define type, from and value
properties
{
  type: 'message',
  from: 'Context1'
  value: {
    text: 'Hello World',
    createdAt: new Date()
  }
}
```

Fig -6: JS Object representing a message

7. LISTENING TO MESSAGES FROM THE WORKER

We had set up a map in the beginning to keep track of Message Ports of different tabs. We then setup a worker.port.onmessage event handler to handle events coming from the Shared Worker directly to the tab.

In cases where the server doesn't set a from field, we just broadcast the message to all tabs using a broadcast channel. All tabs will have a message listener for the WebSocket Channel which will handle all message broadcasts.

This type of a set up can be used in following 2 scenarios:

- Let's say you're playing a game from a tab. You only want the messages to come to this tab. Other tabs won't be needing this information. This is where you can use the first case.
- Now, if you were playing this game on Facebook, and got a text message. This information should be broadcasted across all tabs as the notification count in the title would need to be updated.

9. SHORTCOMINGS

Apart from all the advantages which we get by utilizing SharedWorkers along with standard web socket connection, there are still some rough edges around the implementation.

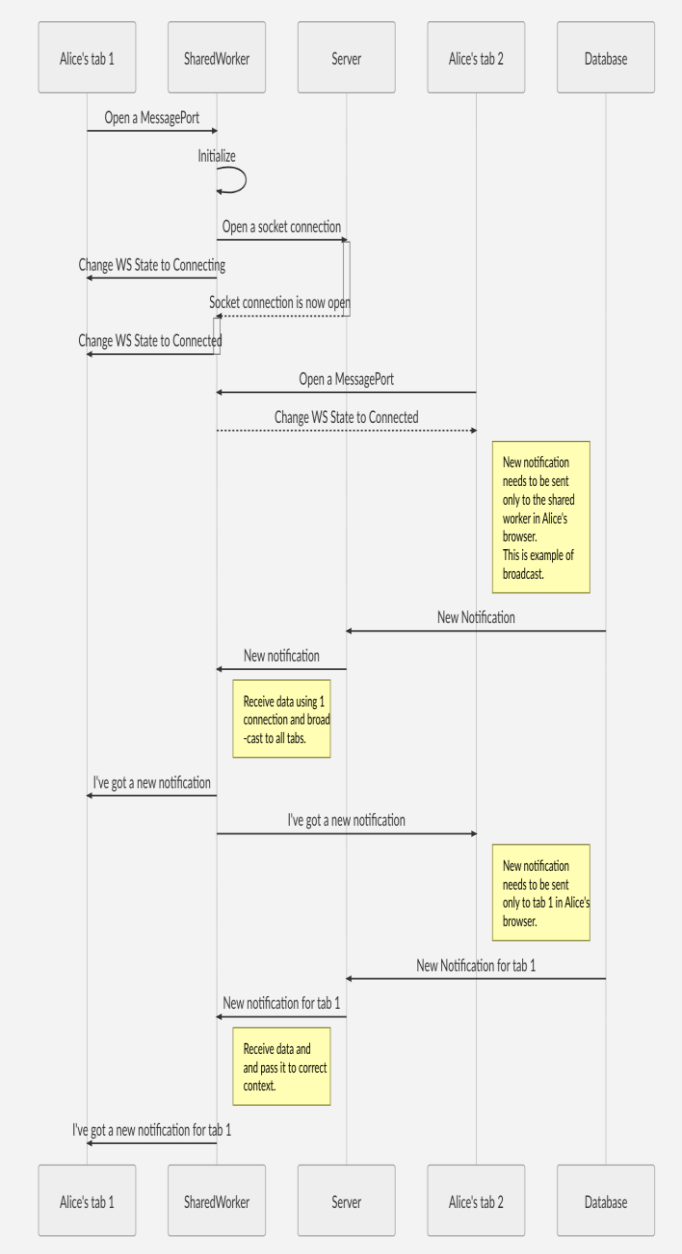
One of limitation of this approach is the message which is sent between the contexts. The messages sent are not references but deep copies instead. Thus, if a JS object is passed between contexts, a deep copy of the object will be created and sent which might create a bottleneck.

Also, this implementation cannot be extended for older browsers because of lack of support for Web workers.

9. CONCLUSION

In a nutshell, WebSockets is powerful protocol which allows us to initiate a full-duplex connection between the server and a client. However, it suffers in terms of scalability. One of the reasons contributing to this issue is the requirement of a dedicated connection for each context (tab/window) on the client machine.

We can utilize the power of SharedWorkers and Broadcast Channels to limit the number of required connections to just one dedicated connection. We then utilize the Broadcast Channel API to communicate the messages between contexts. Thus, mitigating this issue of multiple dedicated connections for a single client. Final diagrammatic representation of our system will look something like this:



REFERENCES

[1] MDN, "Using Web Workers ", 2020, https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers

[2] MDN, "SharedWorkers", 2020, https://developer.mozilla.org/en-US/docs/Web/API/SharedWorker

[3] MDN, "Broadcast Channel", 2020, https://developer.mozilla.org/en-US/docs/Web/API/Broadcast_Channel_API

[4] Lakshminarasimhan Srinivasan, Julian Scharnagl, Klaus Schilling, Analysis of WebSockets as the New Age Protocol for Remote Robot Tele-operation, 2010