



UNIVERSIDADE DO MINDELO
DEPARTAMENTO DE ENGENHARIA E RECURSOS DO MAR

CURSO DE LICENCIATURA EM
ENGENHARIA INFORMÁTICA E SISTEMAS COMPUTACIONAIS

RELATÓRIO

ANO LETIVO 2021/2022 – 3º ANO

TEMA: WEB SOCKET – EISC 3ºANO

AUTORES: MARVIN NEVES, Nº 4876

LEONARDO SABINO, Nº 4873

ORIENTADOR: FREDERICO SOARES

Mindeló, 2022

ÍNDICE

1. INTRODUÇÃO	1
1.1 OBJETIVO	2
1.2 METODOLOGIA.....	2
2.DESENVOLVIMENTO.....	3
2.1 COMPATIBILIDADE	3
2.2 PROTOCOLO WEB SOCKET	4
2.3 INTERFACE	7
2.4 CONSTRUTOR	8
2.5. MÉTODOS	9
2.6. EVENTOS.....	9
2.6. O CÓDIGO.....	11
3. CONCLUSÃO	14
4. BIBLIOGRAFIA	15

1. INTRODUÇÃO

Como sabemos a web tem sido construída com base no conhecido paradigma de solicitação/resposta de HTTP. Um cliente carrega uma página da web e, em seguida, nada acontece até que o usuário clique na próxima página. Por volta de 2005, o AJAX começou a deixar a web mais dinâmica. Mesmo assim, toda a comunicação HTTP era direcionada pelo cliente, o que exigia interação do usuário ou sondagem periódica para carregar novos dados do servidor.

As tecnologias que permitem que o servidor envie dados ao cliente no mesmo momento em que constata que novos dados estão disponíveis foram usadas por algum tempo. Elas eram conhecidas por nomes como "Push" ou "Comet". Um dos problemas mais comuns para criar a ilusão de uma conexão iniciada pelo servidor é a chamada sondagem longa. Com a sondagem longa, o cliente abre uma conexão HTTP com o servidor que permanece aberta até que a resposta seja enviada. Sempre que tem novos dados, o servidor envia a resposta (outras técnicas envolvem Flash, solicitações XHR multipart e os chamados htmlfiles). A sondagem longa e as outras técnicas funcionam muito bem. Você as utiliza todos os dias em aplicativos como o chat do Gmail.

No entanto, todas essas soluções compartilham um problema: elas carregam a sobrecarga de HTTP, que não é adequada para aplicativos de baixa latência. Pense em jogos com vários jogadores no navegador ou em qualquer outro jogo on-line com um componente em tempo real.

Por que a Web e aplicações em tempo real são tão importantes? Vivemos em um mundo em que tudo é feito em tempo real, então, é natural que a web e aplicações estejam se movendo nesta direção. As expectativas de quão rápido a internet deve entregar as informações mudou – atrasos de minutos são inaceitáveis.

Foi então que surgiu o **WebSocket** que é uma tecnologia que permite a comunicação bidirecional por canais full-duplex sobre um único soquete Transmission Control Protocol (TCP). Ele é projetado para ser executado em browsers e servidores web que suportem o HTML5, mas pode ser usado por qualquer cliente ou servidor de aplicativos. A API (Interface de Programação de Aplicações) WebSocket está sendo padronizada pelo W3C (World Wide Web Consortium) e o protocolo WebSocket está sendo padronizado pelo IETF (Internet Engineering Task Force).

Websocket foi desenvolvido para ser implementado em browsers web e servidores web, mas pode ser usado por qualquer cliente ou aplicação servidor. O protocolo Websocket é um protocolo independente baseado em TCP.

1.1 OBJETIVO

O presente relatório do trabalho web Socket da disciplina de sistemas distribuídos do curso da Universidade do Mindelo, possui como objetivo realizar um estudo sobre a tecnologia de comunicação em tempo real WebSockets, além de desenvolver uma aplicação para a realização de troca de mensagens em tempo real. Para tanto foi utilizado o software node.js, como forma de realizar o objetivo pretendido.

1.2 METODOLOGIA

A primeira etapa do trabalho consiste no levantamento bibliográfico necessário a fim de se compreender as características e os problemas relacionados ao desenvolvimento de aplicações em tempo real, buscando dar ênfase as novas tecnologias que estão sendo desenvolvidas no presente momento. Com o levantamento bibliográfico, foi feito um estudo teórico sobre o tema Web Socket e todas tecnologias envolvidas no processo.

2.DESENVOLVIMENTO

2.1 COMPATIBILIDADE

Como WebSockets não opera apenas no cliente, mas também no servidor web, deve-se necessariamente ter a interface da API JavaScript implementada no navegador, bem como o servidor web deve dar suporte ao novo protocolo WebSocket para se fazer o uso desse recurso. Atualmente a maioria dos navegadores já suporta a API JavaScript. O Chrome foi o pioneiro, e desde a versão 4.0, lançada em 2008 já dá suporte à tecnologia. Dentre os principais navegadores, somente o Opera mini e o navegador padrão do Android não suportam tal tecnologia. No Internet Explorer a tecnologia também só é acessível para a versão 10.0, que ainda não está totalmente concluída.

Desktop

	IE	Firefox	Chrome	Safari	Opera
Suporte	Sim	Sim	Sim	Sim	Sim
1ª versão a suportar	10.0	4.0	4.0	5.0	11.0
Versão atual	10.0	15.0	21.0	5.2	12.0

Tabela 1: Suporte do WebSockets em browsers desktop

Celular

	IOS	Opera Mini	Opera mobile	Android	Chrome Mobile
--	-----	------------	--------------	---------	---------------

Suporte	Sim	Não	Sim	Não	Sim
1ª versão a suportar	4.2		11.0		1.0
Versão atual	4.3		12.0		1.0

Tabela 2: Suporte do WebSockets em browsers mobile

Dos servidores web, os principais já realizam a implementação do web socket, entre eles:

- Apache
- Nginix
- Node
- Tornado
- JBoss
- Jetty
- Tomcat
- GlassFish

2.2 PROTOCOLO WEB SOCKET

O protocolo divide-se nas seguintes partes:

URI's (Uniform Resource Identifier) - é uma cadeia de caracteres compacta usada para identificar ou denominar um recurso na Internet



Figura 0-1 : Padrão WebSockets de URIs (Hansa, 2010)

Handshake - para se estabelecer uma conexão WebSocket é necessário que cliente e servidor iniciem usando o tradicional protocolo HTTP, e então realizem a migração para o protocolo WS;

Websockets executa o handshake de abertura ao estabelecer uma conexão WebSocket. Do lado do cliente, `connect()` o executa antes de retornar o protocolo para o chamador. No lado do servidor, ele é executado antes de passar o protocolo para a corrotina **ws_handler** que trata da conexão.

Enquanto o handshake de abertura é assimétrico — o cliente envia uma solicitação de atualização HTTP e o servidor responde com uma resposta HTTP Switching Protocols — websockets visa manter a implementação de ambos os lados consistentes entre si.

Durante o processo, os seguintes passos são executados:

No lado do cliente, `handshake()`:

- constrói uma solicitação HTTP com base no uri e nos parâmetros passados para `connect()`;
- grava a solicitação HTTP na rede;
- lê uma resposta HTTP da rede;
- verifica a resposta HTTP, valida extensões e subprotocolo e configura o protocolo de acordo;
- passa para o estado ABERTO.

No lado do servidor, `handshake()`:

- lê uma solicitação HTTP da rede;
 - chama `process_request()` que pode abortar o handshake do WebSocket e retornar uma resposta HTTP;
- este gancho só faz sentido no lado do servidor;
- verifica a solicitação HTTP, negocia extensões e subprotocolos e configura o protocolo de acordo;
 - constrói uma resposta HTTP com base no acima e nos parâmetros passados para `serve()`;
 - grava a resposta HTTP na rede;
 - passa para o estado ABERTO;

- retorna a parte do caminho do uri.



Figura 0-2: Processo de HandShake (Grego, 2009)

Framing - Toda comunicação é transmitida usando uma sequência de frames. Para não causar problemas em dispositivos de rede intermediários e também por questões de segurança, todos os dados enviados do cliente ao servidor devem passar pelo processo de masking.

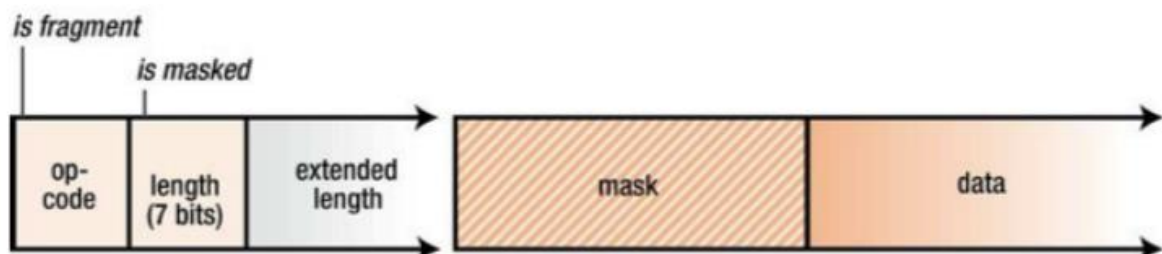


Figura 0-3: Framing (Grego, 2009)

O servidor deve fechar a conexão caso algum dado não esteja mascarado.

Nesse caso, o servidor deve enviar um quadro de fechamento, com código de status 1002 – que significa um erro no protocolo. Pra não ter problemas de codificação, todas mensagens devem estar no formato UTF-8.

2.3 INTERFACE

Além do protocolo WebSocket, a especificação também inclui uma interface para o cliente definida em JavaScript.

```
[Constructor(DOMString url, optional DOMString protocols),
Constructor(DOMString url, optional DOMString[] protocols)]
interface WebSocket : EventTarget {
    readonly attribute DOMString url;

    // ready state
    const unsigned short CONNECTING = 0;
    const unsigned short OPEN = 1;
    const unsigned short CLOSING = 2;
    const unsigned short CLOSED = 3;
    readonly attribute unsigned short readyState;
    readonly attribute unsigned long bufferedAmount;

    // networking
    [TreatNonCallableAsNull] attribute Function? onopen;
    [TreatNonCallableAsNull] attribute Function? onerror;
    [TreatNonCallableAsNull] attribute Function? onclose;
    readonly attribute DOMString extensions;
    readonly attribute DOMString protocol;
    void close([Clamp] optional unsigned short code, optional DOMString reason);

    // messaging
    [TreatNonCallableAsNull] attribute Function? onmessage;
    attribute DOMString binaryType;
    void send(DOMString data);
    void send(ArrayBuffer data);
    void send(Blob data);
};
```

Figura 0-4:: Interface da Api WebSockets (Hickson, 2009)

Percebe-se que a interface é bastante simples, mas oferece o necessário para se abrir, fechar e enviar mensagens, bem como tratar os eventos.

2.4 CONSTRUTOR

Para se abrir uma conexão com um socket basta criar um objeto `WebSocket` acessível globalmente. Para tal, deve ser enviada como parâmetro a URL do socket, no formato padrão segundo a especificação (Hansa,2010).

```
webSocket = new WebSocket('ws://localhost:8080/echo');
```

Quando se conecta em um `WebSocket` há ainda a possibilidade de enviar um segundo parâmetro, que pode ser um `String` ou um `array`, que contém os nomes dos sub-protocolos que a aplicação entende e deseja usar para se comunicar.

```
webSocket = new WebSocket(url, protocolo);  
webSocket = new WebSocket(url,[protocolo1, protocolo2]);
```

2.5. MÉTODOS

São oferecidos dois métodos para o cliente:

1. Send;
2. Close.

O método send envia um dado ao servidor web.

```
websocket.send(mensagem);
```

O método close é disparado quando uma conexão é encerrada

```
websocket.close();
```

2.6. EVENTOS

Segundo Hansa (2010), a API oferece alguns eventos para o tratamento da conexão e para receber as mensagens vindas do servidor:

- Open;
- Message;
- Close;
- Error.

O evento Open é disparado quando uma conexão é estabelecida.

```
websocket.onOpen = function() {  
    console.log('conexão estabelecida');  
}
```

O evento Message indica o recebimento de uma mensagem do servidor.

```
websocket.onMessage = function() {  
  console.log('Mensagem');  
}
```

O evento Close disparado quando a conexão é fechada.

```
websocket.onClose = function() {  
  console.log('conexão encerrada');  
}
```

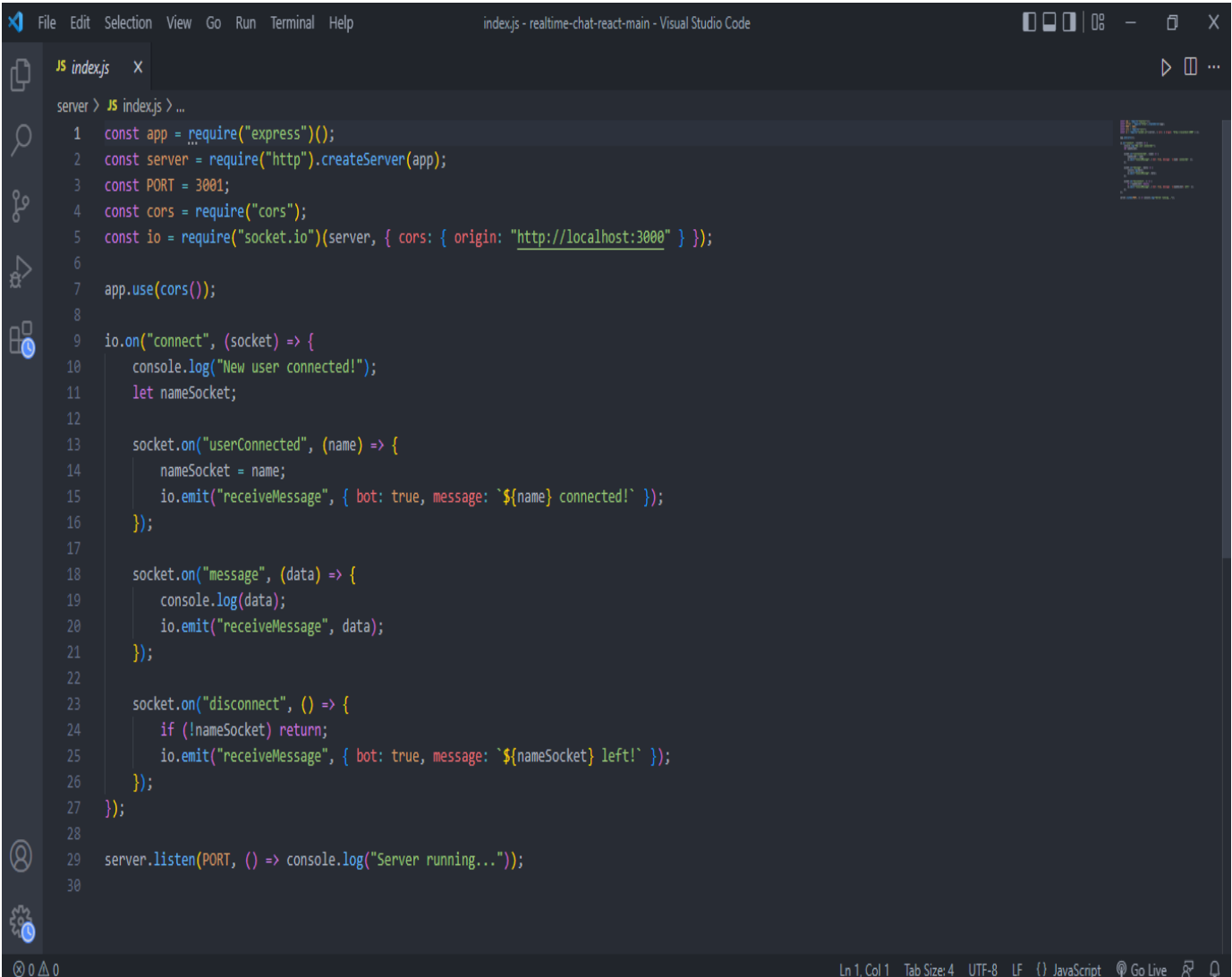
Por fim, o evento Error indica o recebimento de uma mensagem de erro do servidor.

```
websocket.onError = function() {  
  console.log('mensagem de erro recebida');  
}
```

2.6. O CÓDIGO

Ele é dividido em duas partes: o Servidor e o Cliente. Em que como vimos acima funcionará da seguinte maneira: o cliente envia uma solicitação de handshake de **WebSocket** para o servidor de aplicações. A aplicação pode aceitar mensagens binárias e de JSON dos clientes conectados na sessão e transmitir as mensagens para todos os clientes conectados.

Servidor



```
server > JS index.js > ...
1  const app = require("express")();
2  const server = require("http").createServer(app);
3  const PORT = 3001;
4  const cors = require("cors");
5  const io = require("socket.io")(server, { cors: { origin: "http://localhost:3000" } });
6
7  app.use(cors());
8
9  io.on("connect", (socket) => {
10     console.log("New user connected!");
11     let nameSocket;
12
13     socket.on("userConnected", (name) => {
14         nameSocket = name;
15         io.emit("receiveMessage", { bot: true, message: `${name} connected!` });
16     });
17
18     socket.on("message", (data) => {
19         console.log(data);
20         io.emit("receiveMessage", data);
21     });
22
23     socket.on("disconnect", () => {
24         if (!nameSocket) return;
25         io.emit("receiveMessage", { bot: true, message: `${nameSocket} left!` });
26     });
27 });
28
29 server.listen(PORT, () => console.log("Server running..."));
30
```

O cliente por sua vez está dividido em 4 partes: Join, Chat, Message e pelo App.

Join é responsável para fazer com os utilizadores entrem no chat ao digitarem o seu nome.

```
client > src > components > Join > JS Join.js > ...
1  import React, { useState } from "react";
2  import "../Join.css";
3  import { Input, Button } from "@material-ui/core";
4
5  export default function Join({ socket, setVisibility }) {
6    const [name, setName] = useState("");
7
8    const handleSubmit = () => {
9      if (name.trim() === "") return;
10     socket.name = name;
11     setVisibility(true);
12     socket.emit("userConnected", name);
13   };
14
15   return (
16     <div className="join-container">
17       <div className="join-header">
18         <h1>Join</h1>
19       </div>
20       <div className="join-body">
21         <Input type="text" onChange={(e) => setName(e.target.value)} placeholder="Enter your name..." />
22         <Button className="btn-join" variant="contained" color="primary" onClick={() => handleSubmit()}>
23           Enter
24         </Button>
25       </div>
26     </div>
27   );
28 }
29
```

O **Chat** é onde o utilizador é levado após para pelo Join e é onde são armazenadas todas as mensagens e também onde é colocada uma condição, se o utilizador digitar uma mensagem em branco há mensagem não será enviada.

```
client > src > components > Chat > JS Chat.js > Chat
1  import React, { useState, useEffect } from "react";
2  import Message from "../Message/Message";
3  import "../Chat.css";
4  import { Input } from "@material-ui/core";
5  import SendIcon from "@material-ui/icons/Send";
6
7  export default function Chat({ socket }) {
8    const [message, setMessage] = useState("");
9    const [messageList, setMessageList] = useState([]);
10
11    useEffect(() => {
12      socket.on("receiveMessage", (data) => {
13        setMessageList((list) => [...list, data]);
14        scrollDown();
15      });
16    }, [socket]);
17
18    const sendMessage = () => {
19      if (message.trim() === "") return;
20      socket.emit("message", { userId: socket.id, name: socket.name, message });
21      setMessage("");
22      clearInput();
23    };
24
25    const clearInput = () => {
26      document.querySelector("#input").value = "";
27    };
28
29    const scrollDown = () => {
30      const div = document.querySelector('.chat-body');
31      div.scrollTop = div.scrollHeight;
32    }
33  }
```

Message como nome já diz é a mensagem e toda a mensagem tem um texto e o autor que a escreveu.

```
import React from "react";

export default function Message({ text, author, bot, socket, authorId }) {
  return (
    <>
      {bot ? (
        <span className="message-bot">{text}</span>
      ) : (
        <span className="message-container " + (authorId === socket.id ? "message-mine" : "")>
          <p className="author">{author}</p>
          <span className="message">{text}</span>
        </span>
      )}
    </>
  );
}
```

E por último o **APP** que faz todas as importações do Join e Chat e é onde a visibilidade é configurada. Quando o utilizador está na página Join, o chat é escondido até que ele insira o seu nome e ao entrar no chat a página Join é escondida. E também é onde é declarada a variável **const socket** que é responsável por se conectar ao servidor através da porta que foi declarada no Servidor.

```
const socket = io.connect ("http://localhost:3001")
```

```
import './App.css';
import io from "socket.io-client";

import { useState } from "react";

import Join from "../components/Join/Join";
import Chat from "../components/Chat/Chat";

const socket = io.connect("http://localhost:3001");

function App() {
  const [chatVisible, setChatVisible] = useState(false);

  return <div className="App">{chatVisible ? <Chat socket={socket} /> : <Join socket={socket} setVisibility={setChatVisible} />}</div>;
}

export default App;
```

3. CONCLUSÃO

A finalidade desse trabalho foi realizar um estudo sobre a tecnologia de comunicação em tempo real WebSockets, além de desenvolver uma aplicação de troca de mensagens em tempo real.

O que mais dificultou a execução da aplicação foi o desempenho, já tivemos um problema com a conectividade com outros dispositivos para que estes pudessem se conectar e trocar as mensagens. Entretanto, de uma forma geral, todos os objetivos foram concluídos com sucesso.

4. BIBLIOGRAFIA

<https://www.youtube.com/watch?v=xEpE7DSOvVw&t=2s>

<https://www.wallarm.com/what/a-simple-explanation-of-what-a-websocket-is>

<https://pt.wikipedia.org/wiki/WebSocket>

<https://lume.ufrgs.br/handle/10183/86431>

<https://repositorio.ufsc.br/bitstream/handle/123456789/184634/desenvolvimento-de-aplicacoes-com-websockets.pdf?sequence=-1>

<https://www.youtube.com/watch?v=rRkWYx6QK58>