

UNIVERSIDADE DE ITAÚNA

DAVI VENTURA CARDOSO PERDIGÃO
ERIC HENRIQUE DE CASTRO CHAVES

Trabalho Final - Redes de Computadores I
Aplicação em Real-Time com Socket TCP/IP

ITAÚNA
2022

SUMÁRIO

1. INTRODUÇÃO.....	2
2. IMPLEMENTAÇÃO.....	3
3. FUNCIONAMENTO.....	5
4. CONCLUSÃO.....	8
5. REFERÊNCIAS BIBLIOGRÁFICAS.....	8

1. INTRODUÇÃO

- OBJETIVO

O principal objetivo do trabalho é praticar a programação com a biblioteca **Socket** e utilizando o protocolo **TCP**. O trabalho consiste em desenvolver uma aplicação com comunicação **cliente-servidor** em **tempo real**. Este tipo de comunicação pode ser muito importante em determinadas aplicações e, dependendo do projeto, pode ser um recurso primordial. A aplicação desenvolvida neste trabalho é um exemplo clássico disso, pois se trata de um **Chat** onde será necessário atualizar uma tabela de mensagens de uma conversa, assim que a mensagem é enviada por um outro cliente.

- O QUE É UM SOCKET TCP/IP?

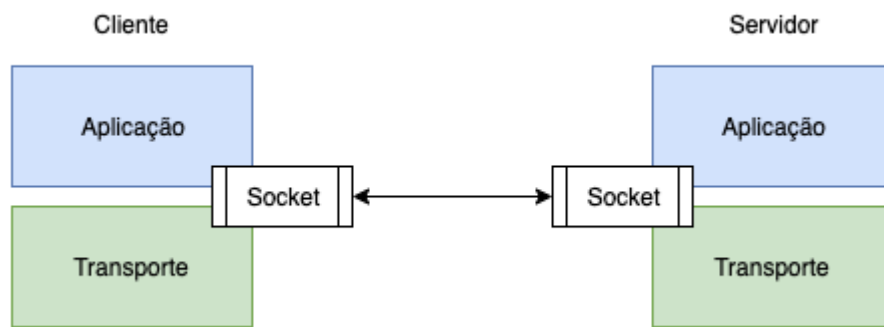
Socket provê a comunicação entre duas pontas cliente-servidor (fonte e destino) - *também conhecido como two-way communication* - entre dois processos que estejam na mesma máquina ou na rede (TCP/IP Sockets). Na rede, a representação de um socket se dá por **ip:porta**, por exemplo: 127.0.0.1:4477 (IPv4). Um **socket** que usa **rede** é um **Socket TCP/IP**, que é o caso dessa aplicação.



Exemplo do fluxo de uma requisição e resposta para um servidor.

Fonte: GABRIEL, João (2020).

Sabendo que TCP/IP é base da nossa comunicação na internet, considerando o modelo de rede **OSI**, os sockets estão entre a **camada de aplicação** e a de **transporte**. Para os processos envolvidos, a sensação é que a comunicação está acontecendo diretamente entre eles, no entanto, ela está passando pelas camadas da rede. Essa abstração provida pelos Sockets é o que chamamos de **comunicação lógica**.



Localização dos sockets no modelo de rede OSI.
Fonte: TEDESCO, Kennedy (2019).

- O QUE É O SOCKET.IO ?

Socket.io é uma **biblioteca** que **facilita** a implementação de Socket e está disponível para linguagens de programação utilizadas em back-end e front-end. Além de permitir **baixa-latência**, ser **bidirecional** e baseado em **eventos de comunicação** entre cliente e um servidor, suas principais características são: confiabilidade, suporte para conexão automática, detecção de desconexão, suporte de sala, suporte binário, etc.

Um servidor que implementa socket possui uma lista de clientes sockets conectados e cada cliente possui um Socket ID, assim, o servidor pode selecionar quais clientes socket vão receber uma determinada mensagem. O Socket.io funciona basicamente com dois métodos: o de **emitir** e o de **escutar**. Porém, vamos nos aprofundar na explicação dessa implementação mais adiante.

2. IMPLEMENTAÇÃO

Como dito anteriormente, o funcionamento do Socket.io é baseado, resumidamente, em dois métodos (emitir e escutar). Esses métodos recebem dois parâmetros, o **evento** e os **dados**. No exemplo abaixo de um trecho da aplicação proposta, temos um socket emitindo (método **emit**) um evento “**sendMessage**” e o dado enviado é um texto com o conteúdo da mensagem:

```
socket.emit('sendMessage', messageObject) //enviando o evento da mensagem
```

Trecho do código exemplificando um evento emitido.
Fonte: Autoria própria (2022).

O exemplo abaixo escuta o evento acima utilizando o método **on**:

```
socket.on('sendMessage', data =>{ // "ouvir" evento e dados do front-end
  messages.push(data); // armazenando as mensagens no array
  socket.broadcast.emit('receivedMessage', data); // enviando mensagem para todos conectados na aplicação
});
```

Trecho do código exemplificando o evento sendo escutado e emitindo outro evento.

Fonte: Autoria própria (2022).

Neste caso de escuta, sempre quando o evento **"sendMessage"** for escutado, a mensagem enviada será armazenada em um array com o parâmetro **data**, que representa os dados que foram enviados na mensagem. Além disso, temos outro método emit, que faz um **broadcast**, isto é, para todos os receptores **simultaneamente**, do evento **"receivedMessage"**. Para escutar essa mensagem no **front-end**, utilizamos novamente o método on, chamando a função **renderMessage()** para renderizar as mensagens na tela com jquery:

```
socket.on('receivedMessage', function(message){
  renderMessage(message);
})
```

Trecho do código exemplificando o evento sendo escutado e chamando a função.

Fonte: Autoria própria (2022).

Para exemplificar a implementação do servidor da aplicação, pode-se observar o seguinte trecho da aplicação:

```
//Definindo a forma de conexão do usuário com o servidor de socket
io.on('connection', socket =>{ //toda vez que um novo cliente se conectar, recebemos o socket
  const ip = socket.handshake.address // atribui para a variável ip qual é o ip da máquina que está acessando
  console.log(`Socket conectado: ${socket.id}`, `Ip do Client: ${ip.replace('::ffff:', '')}`);

  /*enviando todas as mensagens anteriores assim que o socket conectar na aplicação
  assim, as mensagens apenas serão perdidas se o servidor for reiniciado*/
  socket.emit('previousMessages', messages);

  socket.on('sendMessage', data =>{ // "ouvir" evento e dados do front-end
    data['ip'] = ip.replace('::ffff:', ''); //armazena o ip da máquina dentro do Dicionário onde está todas as
    console.log(data) // Informa no console qual foi a mensagem, por quem foi encaminhada e qual o ip de origem
    messages.push(data); // armazenando as mensagens no array
    socket.broadcast.emit('receivedMessage', data); // enviando mensagem para todos conectados na aplicação
  });
});
```

Trecho do código exemplificando a implementação do servidor.

Fonte: Autoria própria (2022).

Ao observar esse trecho é possível notar a presença de alguns métodos comuns em outros momentos da aplicação, como os métodos **on** (porém dessa vez para receber o socket quando um novo usuário se conectar), **emit** e **broadcast**. Nesse trecho também há uma forma de exibir no terminal o IP de cada usuário no momento em que ele se conectar à aplicação, que é através do método **socket.handshake.address**, responsável por acessar o JSON gerado pelo socket e localizar o endereço IP que é informado.

Por último, informamos a porta em que a aplicação poderá ser acessada:

```
server.listen(3000); //definindo porta
```

Definindo a porta no código.
Fonte: Autoria própria (2022).

3. FUNCIONAMENTO

Antes de explicar sobre o funcionamento, é importante entender como funcionam as requisições e respostas do usuário com o servidor. Para isso, vale destacar algumas características do TCP:

- Orientado à conexão (só transmite dados se uma conexão for estabelecida depois de um Three-way Handshake);
- É Full-duplex, ou seja, permite que as duas máquinas envolvidas transmitam e recebam ao mesmo tempo;
- Garante a entrega, sequência (os dados são entregues de forma ordenada), não duplicação e não corrupção;
- Automaticamente divide as informações em pequenos pacotes;
- Garante equilíbrio no envio dos dados (para não causar “sobrecarga” na comunicação);

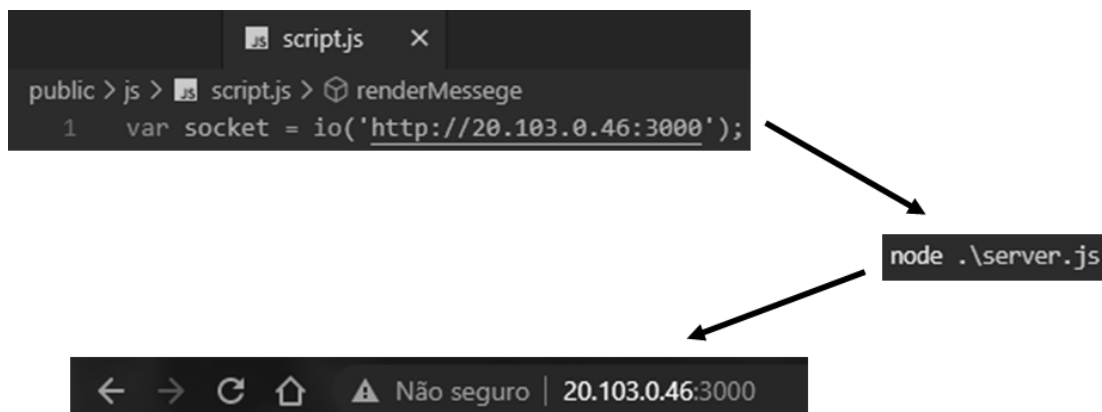


Exemplificando o fluxo de requisições e respostas no TCP.
Fonte: Autoria própria (2022).

Agora que já compreendemos como funciona os métodos da aplicação, vamos à sua execução:

- [Link para o repositório completo no GitHub](#)

Para compilar a aplicação, basta que uma máquina com o Node instalado inicialize o servidor. Para isso, deve-se **inserir o ip** da máquina, mais a **porta 3000** na primeira linha do código “**script.js**”. Após editar essa informação, basta executar o **comando “node .\server.js”** no terminal do projeto, e assim o servidor será inicializado. Feito isso, para qualquer usuário acessar, seja por computador ou telefone, basta inserir no **navegador de internet** essa mesma informação:



Fluxo de ações necessárias para inicializar a aplicação.
Fonte: Autoria própria (2022).

Ao acessar a aplicação, o usuário irá se deparar com a seguinte tela:

A tela de interface do usuário apresenta um formulário com os seguintes elementos:

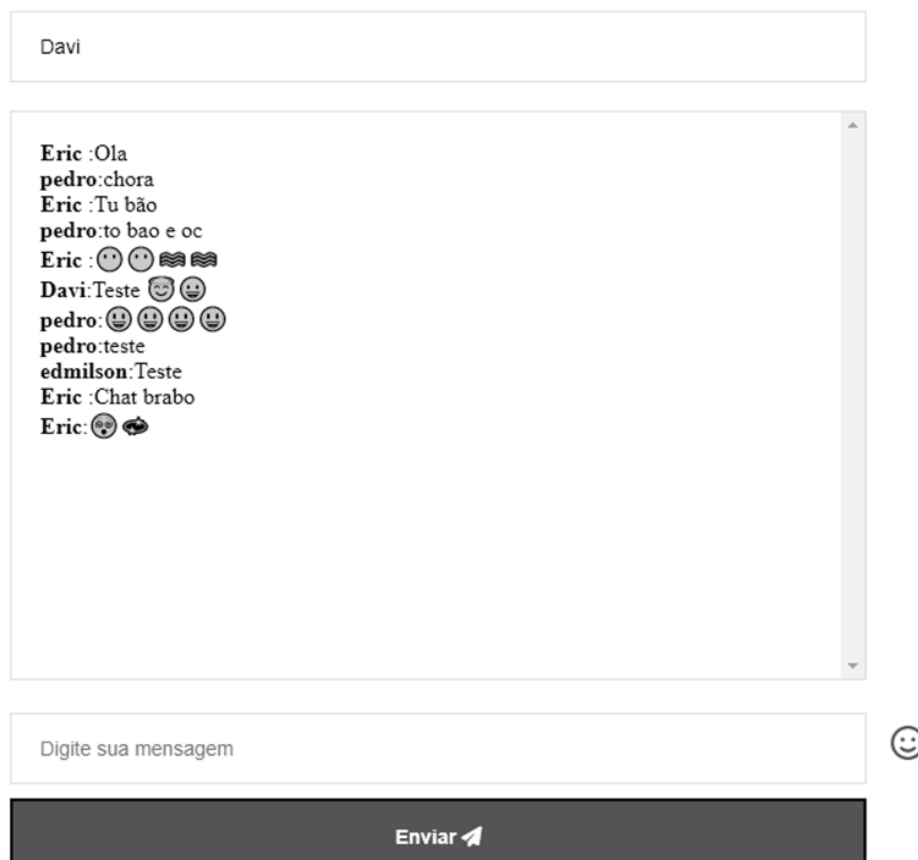
- Um campo de entrada no topo com o placeholder "Digite seu usuário".
- Um grande campo de texto centralizado para mensagens, com uma barra de rolagem à direita.
- Um campo de entrada na base com o placeholder "Digite sua mensagem".
- Um ícone de emoji de sorriso à direita do campo de mensagem.
- Um botão cinza na base com o texto "Enviar" e um ícone de seta para cima.

Tela da aplicação ao iniciar.
Fonte: Autoria própria (2022).

Nela observamos alguns componentes comuns a um chat: input de texto, emoji, botão para enviar, etc. Para interagir no chat, é necessário que o usuário informe seu **nome** e a **mensagem** que deseja enviar.

Em paralelo à isso, enquanto os usuários vão se conectando e enviando suas mensagens, podemos acompanhar todas as interações e informações (**Socket, IP do Cliente, nome do autor, mensagem e IP da máquina que enviou a mensagem**) feitas, através do terminal do projeto:

```
Socket conectado: cCToD7UkuqzQFcsVAAAB - Ip do Client: 20.103.0.16
Socket conectado: oPC84x1Evd2ATiZMAAAD - Ip do Client: 20.103.0.16
{ author: 'Eric ', message: 'Ola ', ip: '20.103.0.16' }
Socket conectado: Zarg0nYrbEgmAbQgAAAF - Ip do Client: 20.103.0.75
{ author: 'pedro', message: 'chora', ip: '20.103.0.75' }
{ author: 'Eric ', message: 'Tu bão ', ip: '20.103.0.16' }
Socket conectado: nFemYmFFDhjvI2ESAAAH - Ip do Client: 20.103.0.46
{ author: 'pedro', message: 'to bao e oc', ip: '20.103.0.75' }
{ author: 'Eric ', message: '😄😄😄😄', ip: '20.103.0.16' }
Socket conectado: 2K8ioelvHLDQdJBAAAJ - Ip do Client: 20.103.0.30
{ author: 'Davi', message: 'Teste 😄😄', ip: '20.103.0.46' }
{ author: 'pedro', message: '😄😄😄😄', ip: '20.103.0.75' }
{ author: 'pedro', message: 'teste', ip: '20.103.0.75' }
{ author: 'edmilson', message: 'Teste', ip: '20.103.0.30' }
```



Interações e informações na aplicação e no terminal.

Fonte: Autoria própria (2022).

4. CONCLUSÃO

Como dito anteriormente, a maioria das aplicações na Internet hoje é baseada na arquitetura **Cliente-Servidor**, sendo que essas duas entidades são completamente diferentes uma da outra devido à natureza das tarefas que executam. Por exemplo, os clientes em navegadores geralmente se comunicam com os servidores por meio de requests e respostas HTTP. O **problema** com essa comunicação é que apenas uma solicitação ou uma resposta pode ser enviada por vez (como um half-duplex). Além disso, os cabeçalhos HTTP contêm muitas e muitas **informações redundantes** que são inúteis uma vez que uma conexão entre o cliente e o servidor é feita.

O Socket.io, por outro lado, nos **permitiu** trabalhar com o mesmo conceito, porém com a **comunicação bidirecional** entre clientes e servidores web, além de trazer recursos como **confiabilidade**, **suporte para reconexão automática**, **deteção de desconexão**, etc. Para essa aplicação, por exemplo, essa ferramenta nos permitiu uma **personalização incrível** e um suporte realmente **simples** para a comunicação com a **API** desenvolvida. Em suma, o Socket.io foi lançado há cinco anos e mesmo assim ainda continua sendo a **melhor opção** para comunicação em **tempo real** quando utiliza-se **node**, foi uma experiência muito **satisfatória** estudar e implementar todos os recursos que essa tecnologia nos oferece, e em paralelo a isso aplicar os **conceitos** apresentamos em aula na disciplina de **Redes de Computadores I**.

5. REFERÊNCIAS BIBLIOGRÁFICAS

- GABRIEL, João. **Comunicação cliente-servidor em tempo real com Socket.io**. Disponível em: <<https://medium.com/digitalproductsdev/comunica%C3%A7%C3%A3o-cliente-servidor-em-tempo-real-com-socket-io-9d3930484b80>>. Acesso em: 27 de outubro de 2022.
- MATHIAS, Pedro. **Aplicações Real Time com Node.js**. Disponível em: <<https://blog.getty.io/aplica%C3%A7%C3%B5es-real-time-com-node-js-8389dae329be>>. Acesso em: 27 de outubro de 2022.
- SOCKET.IO, versão 4.x. **Documentação - Introdução ao Socket.io**. Disponível em: <<https://socket.io/pt-br/docs/v4/>>. Acesso em: 03 de novembro de 2022.
- THEARCANE. **Biblioteca de Emojis - emoji.js**. Disponível em: <<https://github.com/AnonymousXC/emoji.js>>. Acesso em: 10 de novembro de 2022.