Análise de cenários que implementam API Websocket nos aspectos de segurança da informação

Pedro P. Vezalli, Carlos Pagani

[1]Instituto Federal de São Paulo (IFSP) – Campus Hortolândia, SP –Brasil
Av. Thereza Ana Cecon Breda, 1896 – Vila Sao Pedro – Hortolândia - SP – 13183-091

pedrovez@gmail.com, pagani@ifsp.edu.br

Abstract. *Websocket technology proposes a solution to the problem of abuse of HTTP requests provided by a client to a server, often with a source to obtain an update of some previously requested data. This work will explain how this technology works, how to implement it, and what are its advantages and disadvantages, focusing on security aspects. Throughout the article there are also three-based standards that use the Websocket API and these are subject to analysis based on the API standardization defined by the W3C. The methodology used has a comparative, descriptive objective and qualitative approach, focusing mainly on usability and security, where it will be possible and compared the scenarios with suggestions proposed by the W3C and RFC 6455.*

Resumo. *A tecnologia Websocket propõe uma solução para o problema do abuso de requisições HTTP fornecidas por um cliente a um servidor, muitas vezes com a finalidade de obter uma atualização de algum dado requisitado anteriormente. Neste trabalho será explicado como funciona essa tecnologia, como implementá-la, e quais suas vantagens e desvantagens, com foco em aspectos de segurança. No decorrer do artigo também são utilizados códigos baseados em três cenários que utilizam a API de Websocket e estes estarão sujeitos a análises baseando-se na padronização da API definida pela W3C. A metodologia utilizada tem finalidade comparativa, objetivo descritivo e abordagem qualitativa, focando principalmente na usabilidade e na segurança, onde serão analisados e comparados os cenários com as sugestões propostas pela W3C e RFC 6455.*

1. Introdução

A Internet possibilitou a integração de recursos com uma eficiência s em precedentes na história da humanidade. Ela é, ao mesmo tempo, uma capacidade mundial de comunicação e um mecanismo de disseminação de informaç ões extremamente poderoso. O que começou com um sonho de uma ”rede galáctica” documentada em uma série de memorandos escritos pelo MIT em 1962, vem se concretizando em uma rede que já faz muito mais do que o imaginável para a época [Leiner et al. 1997].

Essa evolução extraordinária na capacidade de comunicação possibilitada pela Internet, trouxe consigo preocupações e necessidades de evolução em diversas áreas do conhecimento. A tecnologia tratada nesse artigo está diretamente ligada à evolução da segurança da informação e à necessidade de eficiência na entrega dessas informaç ões.

Websocket é um protocolo que permite uma comunicação de via dupla entre um cliente e um servidor, sua concepção veio para suprir a necessidade do abuso de requisições HTTP enviadas por um aplicativo web para obter a atualização de alguma informação fornecida pelo servidor [IETF 2011].

Enfim chega-se no ponto estudado nesse trabalho, onde utilizando a metodologia de finalidade comparativa, objetivo descritivo e abordagem qualitativa, serão analisadas implementações da API Websocket. A W3C especifica sugestões de como deve ser a implementação da interface Websocket nas aplicações. O trabalho consiste no estudo de três cenários que implementam Websocket, verificando se eles seguem as recomendações da W3C, comparando entre eles os pontos comentados na documentação e propondo melhorias, sempre considerando a segurança da informação.

Os cenários escolhidos são simples, os três cenários utilizam bibliotecas que facilitam a implementação da tecnologia Websocket e seguem uma estrutura padronizada de implementação.

A construção do trabalho proporcionou uma vasta pesquisa sobre a tecnologia, que permitiu unir uma boa quantidade de informações disponibilizadas no artigo e as análises proporcionaram boas discussões sobre a aplicação das recomendações propostas pela W3C.

2. Referencial Teórico

Nesta seção será abordado os aspectos técnicos e teóricos que servem de base para a compreensão do trabalho realizado.

2.1. Modelo de referência em redes de computadores

A fim de compreender o Websocket, primeiro deve-se entender quais são e como funcionam as camadas definidas em redes de computadores e como elas se comunicam. Para apresentar este conteúdo serão utilizados dois modelos que são bem didáticos, o modelo OSI e o modelo TCP/IP.

> [...]Embora os protocolos associados ao modelo OSI raramente sejam usados nos dias de hoje, o modelo em si é de fato bastante geral e ainda válido, e as características descritas em cada camada ainda são muito importantes. O modelo TCP/IP contém características opostas: o modelo propriamente dito não é muito utilizado, mas os protocolos são bastante utilizados [Tanenbaum and Wetherall 2011, p.25].

2.2. Modelos de referência OSI e TCP/IP

Segundo [Tanenbaum and Wetherall 2011], o modelo de referência OSI ou modelo OSI propõe padronizar internacionalmente protocolos usados para a interconexão de sistemas abertos, separando-os em camadas, este modelo consiste em sete camadas.

A ideia é que cada camada possua um propósito bem definido, e que ela converse apenas com as camadas adjacentes, elas seguem na ordem do nível mais baixo para o nível mais alto, em uma estrutura de pilha.

A primeira camada, a camada física, é responsável por definir a transmissão dos bits, desde a definição dos sinais elétricos que representarão os bits zero e um, até a forma como a conexão inicial será estabelecida e como ela será encerrada.

Seguindo o modelo, a camada de enlace de dados é responsável por tratar os dados de entrada, delimitando-os em quadros com a finalidade de esconder os erros reais que podem ocorrer durante sua transmissão para que a camada de rede possa interpretar os dados da maneira correta.

Por sua vez, a camada de rede garante que os pacotes enviados tenham uma origem e destino definidos e que controle as rotas para evitar gargalos.

Na camada de transporte os dados das camadas superiores são tratados e divididos em unidades menores, seu objetivo é garantir que todas as unidades de dados cheguem ao seu destino.

A camada de sessão, gera os tokens, controla diálogos e a sincroniza os dados, permitindo que usuários em máquinas distintas mantenham uma sessão de comunicação.

A partir da camada de apresentação deixa-se de tratar o transporte dos bits e começa a trabalhar a estrutura das informações fornecidas pelas aplicações, definindo padrões e semânticas dessas informações de forma que possam ser interpretadas corretamente do outro lado.

Por fim têm-se a camada de aplicação, onde são definidos os protocolos que especificam as condições necessárias para que a comunicação possa ocorrer [Tanenbaum and Wetherall 2011].

Para explicar o próximo modelo de referência, o modelo TCP/IP aplicado nos primórdios da Internet, será utilizado novamente [Tanenbaum and Wetherall 2011].

Este modelo foi desenvolvido diante da evolução da comunicação de redes, com o propósito de servir de base para uma rede que seja capaz de transformar pacotes baseando-se em uma camada que se relaciona com serviços não orientados a conexão, passando por diferentes topologias de redes.

Este modelo, diferente do OSI, contempla apenas quatro camadas, porém estas camadas cumprem todos os propósitos das camadas apresentadas no modelo anterior.

Nele a camada de enlace, que segundo [Tanenbaum and Wetherall 2011] não é uma camada propriamente dita, descreve o que os enlaces, como ethernet ou linhas seriais, precisam fazer para se relacionar com serviços não orientados a conexões.

Já a camada de internet tem a mesma proposta da camada de rede no modelo anterior, permitir que os hosts apliquem pacotes em qualquer rede e garantir que eles trafegam, contendo um destino definido a partir de uma rota que seja boa para a rede, ela implementa um dos protocolos que dá nome ao modelo, o protocolo IP.

A camada de transporte, se comparado ao modelo anterior, tem o mesmo nome e objetivo, que é permitir que hosts de origem e de destino mantenham uma conversação, nessa camada é definido o outro protocolo que também dá nome ao modelo, o protocolo TCP.

A última camada, a camada de aplicação, contém todos os protocolos de nível mais alto, carregando consigo, se necessário, também a responsabilidade das camadas de sessão e apresentação definidas no modelo anterior [Tanenbaum and Wetherall 2011].

Na (Figura 1) é apresentado os dois modelos lado a lado, onde é possível comparar

as camadas definidas por cada modelo.

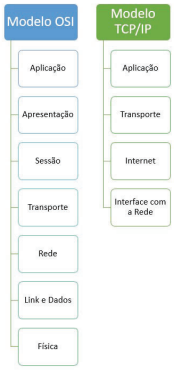


Figura 1. Imagem representativa das camadas definidas nos modelos OSI e TCP/IP.

Parece que o modelo TCP/IP é um modelo resumido do modelo OSI, pois ele possui menos camadas e as camadas que possui tem o nome igual ou parecido com as camadas do modelo OSI. O que ocorre na verdade é que para o modelo TCP/IP a camada de aplicação abrange também as responsábilidades das camadas de apresentação e sessão do modelo OSI e a camada de interface com a rede abrange as responsábilidades das camadas física e de dados.

2.3. Protocolo TCP

Segundo a [CCM 2017], o TCP é um protocolo da camada de transporte do modelo TCP/IP que gere os dados providos da camada de internet(que utiliza o protocolo IP) encapsulando-os em datagramas IP. Ele é um protocolo orientado a conexão, ou seja, sua responsabilidade é definir uma comunicação entre duas máquinas e controlar o estado da transmissão.

Sua qualidade é entregar os datagramas originários do protocolo IP na mesma ordem em que foram enviados, garantindo a sequência correta da informação além de quebrar os dados em segmentos de diversos tamanhos. Para garantir uma boa comunicação os dados são encapsulados, de forma que, junto ao pacote de dados vai um cabeçalho que sincroniza as transmissões e assegura a sua recepção.

O three ways handshake, exemplificado na (figura2), é o mecanismo que inicia a conexão TCP dividindo em três fases. Primeiro o cliente envia um segmento SYN com um número de ordem N, indicando que deseja iniciar uma sincronização, em seguida o servidor incrementa em um o número de aviso de recepção, que contém o número de ordem inicial do cliente, e responde com o número de ordem do servidor um segmento ACK SYN, pois se trata de uma sincronização com uma resposta de confirmação do recebimento do primeiro segmento SYN.

Em seguida, finalizando o handshake, o cliente responde com um segmento ACK confirmando a resposta do servidor e envia os dados. O número de aviso de recepção representa o número de ordem inicial do servidor, incrementado em um. Após a resposta seu número de ordem será incrementado para a próxima solicitação se sincronização [CCM 2017].

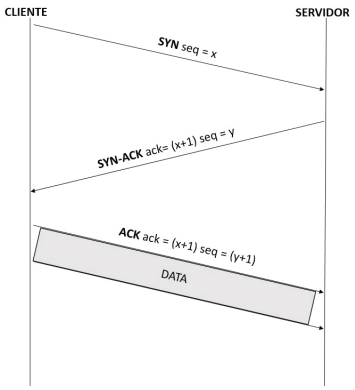


Figura 2. Imagem que representa o handshake do protocolo TCP.

2.4. Protocolo HTTP

HTTP é um protocolo da camada de aplicação que permite a obtenção de recursos na web. Sua arquitetura é cliente-servidor, ou seja, o cliente faz uma requisição ao servidor buscando resgatar algum dado ou informação que este possui.

A conexão não é responsabilidade do HTTP, este é enviado sobre o protocolo TCP que fica na camada de transporte e é o verdadeiro responsável por estabelecer a conexão.

Ele não possui estado, ou seja, não existe uma relação entre duas requisições sendo feitas através da mesma conexão, dependendo da utilização de cookies para criar esta relação.

O fluxo para a comunicação com o servidor pode ser dividido em quatro etapas, sendo que no primeiro deve ser aberto uma conexão TCP, que será utilizada para enviar requisições, e receber uma resposta.

Na segunda etapa são enviadas mensagens HTTP, que são encapsuladas em frames, essas mensagens possuem três atributos obrigatórios, a ação(método e versão do protocolo), o endereço do serviço e a linguagem dos dados requisitados como representa a (Figura 2).

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Figura 3. Representação de uma requisição HTTP [RFC6455 2011].

A terceira etapa é a resposta do servidor, que contém se a requisição foi aceita e seu status(por padrão, status na faixa 200-299 representa sucesso), informa a data da requisição, o servidor, a última modificação, o tipo do conteúdo, o tamanho do conteúdo e o conteúdo em si como apresentado na (Figura 3). Existem outros atributos no header que não são obrigatórios. E por fim é encerrado a conexão [colMDN 2020].

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Figura 4. Representação de um response HTTP [RFC6455 2011].

O ponto onde o Websocket melhora o HTTP é na diminuição de requisiç ões que precisam ser enviadas para obter a resposta do servidor quando o mesmo não consegue entregá-la no momento exigido. Por exemplo na utilização de webhooks, como o Websocket mantém a comunicação de via dupla aberta, não é necessário o cliente ficar enviando requests perguntando o status do pedido ao servidor.

O protocolo HTTP recebeu melhorias na sua versão 2.0, que veio para corrigir alguns problemas em soluç ões paliativas implementadas na versão 1.1. Os principais objetivos dessa versão são melhorias no desempenho, correç ões de problemas, redução da latência com a multiplexação completa de solicitação e resposta, melhor compressão da carga do protocolo, suporte a priorização de solicitaç ões e envio push de servidor [Grigorik and Surma 2021].

Devido as alteraç ões na versão dois do protocolo HTTP, principalmente por conta da multiplexação na comunicação que não permite campos de cabeçalho em toda a conexão ou códigos de status, a RFC 8441 estabelece como deve ser esse upgrade de protocolo para o Websocket [McManus 2018].

Mais recentemente temos a versão 3 do HTTP, que é o protocolo QUIC da Go-ogle traduzido para a semântica HTTP, onde o QUIC fornece negociação de protocolo, multiplexação baseada em stream e controle de fluidez [Bishop and Ed.Akamai 2020].

2.5. Padrões para a Internet

A [RFC 2016] contém documentos técnicos e organizacionais sobre a Internet, incluindo as especificaç ões e documentos de política produzidos por IETF (Internet Engineering Task Force), Internet Research Task Force (IRTF), Internet Architecture Task Force (IRTF), Internet Architecture Board (IAB) e envios independentes. A RFC surgiu em 1969 pelo Steve Crocker, para organizar os documentos do programa de pesquisa ARPAnet.

A RFC 6455 é a RFC responsável por documentar a tecnologia Websocket.

2.6. RFC 6455

Na seção dez da RFC 6455 encontra-se a documentaçãos sobre Websocket, onde explica como funciona, o que faz e seus problemas.

A tecnologia Websocket oferece soluç ões para o problema do abuso de solicitaç ões HTTP utilizados para manter uma comunicação bidirecional entre cliente e servidor(como ao requisitar uma atualização de uma informação). O objetivo do Websocket é abrir uma comunicação bidirecional e manter essa comunicação aberta até a solicitação(de qualquer um dos lados) de encerramento, evitando a abertura de várias conexões distintas, se eximindo da necessidade de enviar cada requisição com um cabeçalho

HTTP e tirando a necessidade do cliente de mapear as requisições para rastrear as respostas.

Para implementar essa tecnologia com segurança devemos considerar alguns pontos, o primeiro é evitar ”Non-Browser Clients”pois a aplicação Websocket sempre deve validar o origin da chamada para evitar que scripts maliciosos sejam executados na aplicação.

A seguir é comentado sobre um teste que foi executando durante o desenvolvimento da documentação, onde foi possível envenenar proxies em cache, a solução encontrada foi mascarar todos os dados enviados do cliente para o servidor, utilizando uma chave de mascaramento a cada mensagem, evitando que intermediários corrompam os dados.

Na RFC também comenta sobre implementar limite de tamanho das mensagens trocadas, para evitar ataques DoS. Também é recomendado autenticar o cliente, porém a RFC não especifica qual metodo de autenticação deve ser usado.

Por fim se recomenda que as conexões devem ser feitas sobre TLS(uris wss), e deve lidar com dados recebidos de frorma inválida finalizando a conexão.

2.7. Protocolo Websocket

O protocolo Websocket permite uma comunicação de via dupla entre um cliente e um servidor que aceitou esta comunicação. Ele se baseia na abertura de um handshake seguido de um ou mais blocos de mensagem posicionada sobre o protocolo TCP.

Seu objetivo é prover um mecanismo para aplicações web, que necessitam de uma comunicação de via dupla com o servidor, que não precisam depender de várias requisições HTTP.

O protocolo se divide em duas etapas, o handshake e a transferência dos dados, lembrando que após a abertura do canal Websocket proporcionado por um handshake bem sucedido a transferência de dados pode ser feita livremente por ambas as partes(cliente e servidor).

A primeira etapa, o handshake é iniciado com o cliente enviando uma requisição HTTP informando que deseja fazer um upgrade para o Websocket, como mostra a (Figura 3).

É importante ressaltar que esta requisição é um upgrade, como mostra o campo ”Connection”e este upgrade é para Websocket, como mostra o campo ”Upgrade”.

Além desses dois campos novos na requisição, existem outros três adicionais, onde ”Sec-WebSocket-key”contém a chave que o servidor deve extrair como prova para que a partir desse ponto, se acordado a solicitação de upgrade, apenas conexões Websocket serão aceitas, o campo ”Sec-WebSocket-Protocol”que informa uma lista de sub protocolos sugeridos para o servidor, que deve concordar com pelo menos um (a lista também pode ser vazia, informando que não existe um subprotocolo exigido pelo cliente), e o campo ”Sec-WebSocket-Version”que informa a versão do Websocket utilizada.

Em contra partida a resposta HTTP deve sinalizar que será trocado de protocolo contendo os dois campos anteriores ”Connection”e ”Upgrade”e um campo novo ”Sec-

WebSocket-Accept”que informa para o cliente que a solicitação foi bem sucedida, a (Figura 4) contém um exemplo de uma resposta [RFC6455 2011].

2.8. Padronização WEB

W3C ou World Wide Web Consortium é uma comunidade internacional em que as organizaç ões membros, a equipe de tempo integral e o público trabalham juntos para desenvolver padrões da Web. Liderada pelo inventor e diretor da Web, Tim Berners-Lee, e pelo CEO Jeffrey Jaffe [W3C].

2.9. API Websocket

Agora será utilizado a definição da W3C para explicar como deve ser a API que implementa o protocolo Websocket, na (Figura 5) é representado sua interface.

```
[Constructor(DOMString url, optional (DOMString or 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
          attribute EventHandler onopen;
          attribute EventHandler onerror;
          attribute EventHandler onclose;
  readonly attribute DOMString extensions;
  readonly attribute DOMString protocol;
  void close([Clamp] optional unsigned short code, optional DOMString reason);

  // messaging
          attribute EventHandler onmessage;
          attribute DOMString binaryType;
  void send(DOMString data);
  void send(Blob data);
  void send(ArrayBuffer data);
  void send(ArrayBufferView data);
};
```

Figura 5. Representação da interface da API Websocket [W3C 2015].

Será utilizado a (Figura 5) para explicar como ela funciona, primeiro deve-se olhar para seu construtor, pode-se perceber que ele recebe um ou dois parâmetros, o primeiro é a URL, que deve conter o host, a porta, o nome da aplicação e seu caminho, já o segundo é opcional, consiste na lista de sub protocolos informados na requisição HTTP dentro do atributo ”Sec-WebSocket-Protocol”, caso este parâmetro não seja informado uma lista vazia é interpretada pela aplicação (Figura 6).

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

Figura 6. Representação dos parâmetros definidos na interface da API Websocket [W3C 2015].

A seguir encontra-se a representação dos estados definidos na aplicação, nela existem duas variáveis, a primeira ”readyState”, numérica retrata o estado da conexão, onde zero simboliza conectando, um representa aberta, dois fechando e três fechada, e a segunda ”bufferedAmount”que exprime o tamanho de todo o conteúdo trocado pelo cliente e servidor (Figura 7).

Sobre os estados deve-se perceber que, conectando significa que a conexão ainda não foi estabelecida, aberta define que a conexão está estabelecida e é possível a comunicação, fechando representa que há um solicitação de encerramento da conexão e fechada quando a conexão foi finalizada.

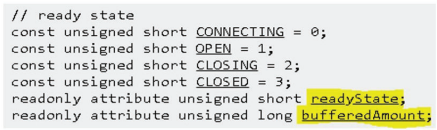


Figura 7. Representação dos estados da conexão definidos na interface da API Websocket [W3C 2015].

A seguir é estabelecido os handler dos estados apresentados acima, onde deve ser especificado as funções responsáveis por lidar com a abertura da conexão(onOpen), quando ocorre algum erro(onError) e quando é solicitado o fechamento da conexão(onClose), onde este último chama a função close que pode receber dois parâmetros opcionais(o código do erro e a mensagem de erro). Na mesma sessão são definidas as variáveis que receberão o subprotocolo selecionado e a extensão (Figura 8).

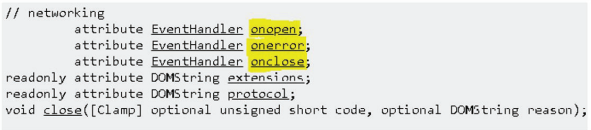


Figura 8. Representação dos handlers dos estados definidos na interface da API Websocket [W3C 2015].

Encerra-se com a definição da mensagem trocada entre o cliente e servidor, aqui

é definido uma variável que receberá a mensagem e a função responsável por lidar com a ela (Figura 9) [W3C 2015].

```
// messaging
         attribute EventHandler onmessage;
         attribute DOMString binaryType;
void send(DOMString data);
void send(Blob data);
void send(ArrayBuffer data);
void send(ArrayBufferView data);
```

Figura 9. Representação da mensagem definida na interface da API Websocket [W3C 2015].

Segue um exemplo da própria W3C de uma implementação do handler da mensagem (Figura 10).

```
mysocket.onmessage = function (event) {
  if (event.data == 'on') {
    turnLampOn();
  } else if (event.data == 'off') {
    turnLampOff();
  }
};
```

Figura 10. Exemplo handler de mensagem websocket[W3C 2015].

2.10. Java EE

Primeiro precisa-se entender que Java é uma linguagem de programação e plataforma computacional lançada pela Sun Microsystems em 1995 [Java].

O Java EE são especificações de como implementar um software que atende a infraestrutura da Web utilizando a linguagem de programação Java.

Ele também disponibiliza uma série de APIs para o desenvolvimento web, como Java Server Pages (JSP), Java Servlets, Java Server Faces (JSF), Enterprise Javabeans Components (EJB), Java Persistence API (JPA), dentre outras [Caelum 2015].

2.11. Javascript

JavaScript é uma linguagem de programação, utilizada principalmente em aplicações web, interpretada e baseada em objetos com funções de primeira classe(permite pas-sar funções como argumentos), além de ser baseada em protótipos, multi-paradigma e dinâmica, suportando estilos de orientação a objetos [colMDN 2019].

2.12. NodeJS

NodeJS é uma biblioteca JavaScript utilizada para aplicaç ões back-end, também pode ser definida como um ambiente de execução Javascript server-side segundo [Lenon 2018] é muito utilizado na criação de APIs(Application Programming Interface).

2.13. ReactJS

React é uma biblioteca javascript baseada em componentes desenvolvida para ajudar na construção de interfaces com o usuário [Facebook 2020].

A componentização no React é muito importante, componentes são trechos de códigos que formam uma funcionalidade isolada que pode ser reaproveitada durante todo o projeto, por exemplo pode-se ter um componente que é um botão, este botão pode ser reaproveitado durante todo o projeto, ele especifica todo o seu estilo e suas funcionalidades, como um callback para o clique por exemplo. Componentes também possuem estados próprios.

3. Metodologia

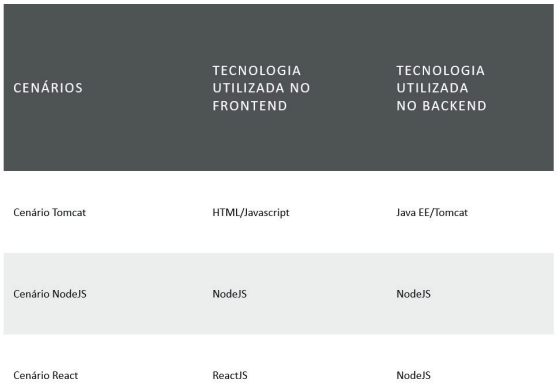
A metodologia utilizada tem finalidade comparativa, objetivo descritivo e abordagem qualitativa, focando principalmente na segurança. A proposta é de aprofundar o conhecimento sobre Websocket, descrevendo seu funcionamento e analisando a segurança por meio da comparação entre três implementações, verificando se elas seguem o que propõe a W3C e se contemplam tratamentos para possíveis vulnerabilidades de segurança.

Será utilizado na comparação dos cenários a interface e as dicas fornecidas pela W3C encontrado na página da própria [W3C 2015], além de uma análise sobre a segurança desta tecnologia documentada na RFC 6455 e sugestões de como evitar eventuais vulnerabilidades proporcionada pela tecnologia.

Sobre os cenários implementados pode-se perceber três níveis de aplicação realizadas em linguagens diferentes, onde na base encontra-se a aplicação em nodejs que utiliza a biblioteca Websocket da npm, que apenas cria uma conexão Websocket e envia números aleatórios para o servidor que funciona como um eco respondendo os mesmos números recebidos. Para fins comparativos foi utilizado a extensão Web Socket Testing para simular um client e deixar o funcionamento desta implementação mais parecida com as outras.

Um nível acima se posiciona a aplicação em java utilizando o tomcat, que também funciona como um servidor eco, porém nesta implementação é criado a mensagem que será enviada, além de conter uma interface gráfica.

No último nível a aplicação em react, utilizando também a biblioteca Websocket da npm, que utiliza os mesmos conceitos dos cenários anteriores, de um servidor eco, mas com a finalidade de criar um chat onde várias pessoas podem trocar mensagens entre elas. Na (Figura 11). contém uma tabela representando as três implementações.



CENÁRIOS	TECNOLOGIA UTILIZADA NO FRONTEND	TECNOLOGIA UTILIZADA NO BACKEND
Cenário Tomcat	HTML/Javascript	Java EE/Tomcat
Cenário NodeJS	NodeJS	NodeJS
Cenário React	ReactJS	NodeJS

Figura 11. Tabela representativa das tecnologias dos cenários.

4. Desenvolvimento

Nesta seção será documentado o passo a passo de três cenários diferentes que implementam a tecnologia Websocket e uma comparação a nível de código sobre as implementações, comparando os cenários entre si e se seguem os padrões sugeridos pela W3C.

Todos os cenários possuem a implementação do frontend e backend, porém cada um possui linguagem e complexidade diferente dos demais. o primeiro é com o backend em JavaEE utilizando o tomcat como servidor web, já no frontend é uma combinação simples de HTML e javascript, simulando uma troca de mensagem entre o cliente e o servidor, com o servidor respondendo o cliente como um eco(exatamente com a mesma mensagem que o cliente lhe enviou).

O segundo cenário é o mais simples dos três, com o frontend e o backend desenvolvidos em nodejs, onde não possui um interface para o usuário, tendo que executar o código do lado do cliente que gera números aleatórios e envia ao servidor e espera sua resposta, encerrando a execução assim que atinge a condição de parada.

Já o terceiro utiliza um misto de nodejs para o backend e react para o frontend, no backend utiliza a mesma biblioteca Websocket que a segunda implementação, porém a estrutura é um pouco diferente, tratando falhas que o segundo não trata por exemplo. Este possui o frontend mais complexo das três implementações, com o objetivo de criar uma sala de conversa, é possível criar várias sessões e trocar mensagens utilizando o servidor nodejs no backend, os usuários nesta implementação são autenticados para que se consiga identificar o autor da mensagem.

4.1. Implementação cenário Tomcat

O primeiro cenário é uma implementação do Tomcat em java com o JDK 7, utilizando o Eclipse JEE Kepler e o Apache v7. O howto dessa implementação foi retirado da referência [Boadas 2016]. O projeto consiste em uma classe java para implementar o servidor e um arquivo HTML executando um javascript para o cliente.

Começando pelo servidor, é possível criá-lo a partir de um objeto Websocket, para isso deve-se importar a biblioteca “javax.websocket.*”, assim consegue-se acesso as annotations do Websocket. Então pode-se utilizar a annotation “@ServerEndpoint” para passar o caminho onde o servidor estará rodando.

```
@ServerEndpoint("/websocketendpoint")
public class WsServer
```

Figura 12. Implementação Tomcat. Importações da biblioteca javax.websocket

Dentro do servidor é declarado os quatro handlers definidos na api de Websocket, onOpen, onCLose, onMessage e onError, onde neste exemplo o onOpen e onClose apenas exibem uma mensagem da conexão, o onError dispara uma exception e o onMessage exibe a mensagem recebida e exibe um eco como resposta do servidor.

Finalizado a implementação do servidor segue-se para a implementação do cliente, que será um HTML com um formulário que contém um campo de texto, um botão para enviar o texto ao servidor, um botão para encerrar a conexão com o servidor e um campo para exibir as mensagens trocadas com o servidor.

O script do HTML contempla a implementação dos mesmos métodos Websocket definidos no servidor. No onOpen, que é executado quando é estabelescida uma conexão com o servidor, é exibido um log avisando que a conexão foi aberta, como mostra o código abaixo.

```
function wsOpen(message){
 echoText.value += "Connected ... \n";
}
```

Figura 13. Implementação Tomcat. Implementação do metodo wsOpen no cliente

Se ocorrer algum erro durante a estabilização da conexão o método onError é invocado, exibindo um log avisando que ocorreu um erro durante a tentativa de conexão. E caso a conexão esteja aberta e algum dos lados queira finalizá-la, o método onClose é invocado, encerrando a conexão e informando com um log que o cliente foi desconectado.

Na (Figura 14) é possível ver o resultado da nossa aplicação executando.

Figura 14. Execução cliente da aplicação Tomcat.

4.2. Implementação cenário NodeJS

O segundo cenário utiliza a biblioteca ”websocket”disponibilizada pelo npm, a implementação foi realizada utilizando o tutorial básico da biblioteca, encontrado na referência [Castillo 2019].

Para a implementação foi utilizado o NodeJS na versão 12.14.0 e o npm na versão 6.13.4.

Iniciando pelo servidor, o primeiro passo é criar um objeto servidor, para isso é utilizado a função createServer da biblioteca HTTP com um callback que contém um log informando a url do request. Em seguida deve-se informar a porta na qual o servidor estará executando.

Para criar uma instância do objeto WebSocketServer é definido como servidor http o objeto servidor criado na etapa anterior. Exemplo na (Figura 15)

```
var server = http.createServer(function(request, response) {
 console.log((new Date()) + ' Received request for ' + request.url);
 response.writeHead(404);
 response.end();
});

server.listen(7777, function() {
    console.log((new Date()) + ' Server is listening on port 7777');
});

wsServer = new WebSocketServer({
httpServer: server,
autoAcceptConnections: false
});
```

Figura 15. Implementação NodeJS. Criação do servidor

Nessa implementação também é criado uma função para filtrar quais origins são permitidas para abrir a conexão Websocket com servidor, o código abaixo tem um exemplo de como seria um filtro de origin(no caso ele permite qualquer origin).

```
function originIsAllowed(origin) {
return true;
}
```

Figura 16. Implementação NodeJS. Implementação da validação de origin

Para criar a lógica da conexão, o primeiro passo é utilizar nossa função que verifica se o origin é valido para abrir a conexão, caso não seja é disparado um reject, do contrário um accept e segue-se com a lógica.

É definido um handler para o onMessage, toda vez que o servidor receber uma mensagem ele irá printa-la e responder um eco dessa mensagem e por fim tem a definição do handler onClose, que encerra a conexão.

Para implementar o cliente deve se importar a biblioteca ”websocket”e utilizar o objeto w3cwebsocket, que é um objeto Websocket com os padrões da W3C.

O cliente é uma instância desse objeto w3cwebsocket definido anteriormente, no qual se passa o ”path”onde o servidor estará escutando e o protocolo que será utilizado para a troca de mensagens, lembrando que o protocolo definido no cliente deve estar contido na lista de protocolos permitidos na implementação do servidor.

```
var client = new W3CWebSocket('ws://localhost:7777/', 'echo-protocol');
```

Figura 17. Implementação NodeJS. Criação do cliente

O cliente contém a implementação dos quatro métodos Websocket igual o cliente do cenário anterior, onOpen, onClose, onMessage e onError.

Nesse cenário o onOpen é um pouco diferente, seu encargo é o mesmo, porém nele é criado uma função que gera números aleatórios para enviar como mensagem ao servidor a cada segundo, além de como escolha do autor deste artigo, no onOpen chama-se o método onClose para encerrar a conexão ao decorrer de cinco segundos. Esta decisão foi tomada já que no howto original não existia a implementação do onClose e consequentemente não existe uma lógica de encerramento.

```
client.onopen = function() {
 console.log('WebSocket Client Connected');

 function closeConnection() {
  client.close();
 }

 function sendNumber() {
  if (client.readyState === client.OPEN) {
   var number = Math.round(Math.random() * 0xFFFFFF);
    client.send(number.toString());
    setTimeout(sendNumber, 1000);
  }
 }
 sendNumber();
 setTimeout(closeConnection, 5000);
};
```

Figura 18. Implementação NodeJS. Implementação da lógica de abertura da conexão no cliente

Após a implementação do cliente e servidor é possível visualizar a execução da aplicação, na figura (Figura 19) encontra-se o resultado da execução, que foi enviado cinco números aleatórios para o servidor, que funcionando como eco respondeu os cinco números para o cliente.

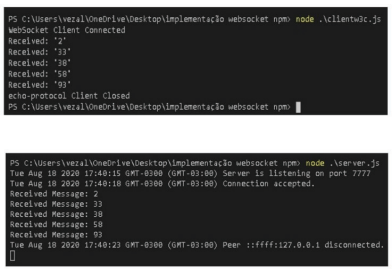


Figura 19. Execução do cliente e do servidor da aplicação em Nodejs.

Este cenário foi executado também utilizando a ferramenta Web Socket Testing

para google chrome onde simula um cliente para conectar ao servidor e trocar as mensagens, como representa a (Figura 20).

Figura 20. Execução da aplicação com o client Web Socket Testing.

4.3. Implemantação cenário React

O cenário que se segue apresenta um servidor WebSocket em Nodejs e um cliente em React, com o objetivo de demonstrar uma aplicação de transmissão de mensagem em tempo real a partir do passo a passo apresentado na referência [Meenakshi 2019].

A biblioteca utilizada nesse cenário é a mesma utilizada no cenário anterior, a ”websocket”encontrada na npm.

No servidor, assim como no cenário anterior, é criado o objeto WebSocket utilizando como servidor http a instância do objeto http.

Porém, diferente do cenário anterior, o servidor especificará um ID para cada cliente que estabeleça uma conexão com ele, para que consiga diferenciar quem é o autor da mensagem, no trecho de código abaixo contém a lógica do gerador de IDs.

```
const getUniqueID = () => {
 const s4 =()=> Math.floor((1 + Math.random())
 * 0x10000).toString(16).substring(1);
 return s4() + s4() + '-' + s4();
};
```

Figura 21. Implementação React. Criação do servidor e gerador de ID do usuário

A lógica da inicialização é demonstrada no trecho de código abaixo, onde antes do cliente estabelecer uma conexão se deve definir um ID para ele utilizando a nossa função do trecho de código acima, após o ID gerado se cria a conexão. Dentro da conexão tem declarado o handler onMessage que além de logar a mensagem recebida mapeia o cliente que enviou a mensagem por conta do método que gera os IDs.

```
wsServer.on('request', function (request) {
 var userID = getUniqueID();
 console.log((new Date()) + ' Recieved a new connection from origin '
  + request.origin + '.');

 const connection = request.accept(null, request.origin);
 clients[userID] = connection;
```

```
 console.log('connected: ' + userID + ' in '
 + Object.getOwnPropertyNames(clients));

 connection.on('message', function(message) {
  if (message.type === 'utf8') {
   console.log('Received Message: ', message.utf8Data);

   for(key in clients) {
    clients[key].sendUTF(message.utf8Data);
    console.log('sent Message to: ', clients[key]);
   }
  }
 })
                                                                    });
```

Figura 22. Implementação React. Implementação da lógica de conexão no servidor

O cliente é uma aplicação em react e utiliza a mesma biblioteca Websocket do servidor. Ele contém uma tela para login que pede para o usuário informar seu nome e uma tela para a troca de mensagens, que contém um espaço onde serão mostrados os cards com as mensagens, um campo de texto para digitar a mensagem e um botão para enviá-la .

O primeiro passo da implementação do cliente é criar o objeto cliente como uma instância do W3CWebSocket, que é a variável que recebe o objeto exposto da biblioteca Websocket, passando o caminho do servidor.

Como lógica do react é criado um state para guardar as informações do usuário, como seu nome, um booleano para verificar se está logado e um vetor de todas as mensagens trocadas com ele.

O botão de submit envia um objeto para o servidor contendo o tipo da mensagem, o texto e o usuário que enviou essa mensagem, como exemplificado no trecho de código abaixo.

```
onButtonClicked = (value) => {
 client.send(JSON.stringify({
   type: "message",
   msg: value,
   user: this.state.userName
 }));
 this.setState({ searchVal: '' })
}
```

Figura 23. Implementação React. Função de envio da mensagem

O componentDidMount, que é a função do React executada após cada renderização do componente é um ótimo lugar para requisitar a conexão com o servidor e definir o handler onMessage para lidar com a resposta do servidor.

```
componentDidMount() {
 client.onopen = () => {
  console.log('WebSocket Client Connected');
 };
 client.onmessage = (message) => {
  const dataFromServer = JSON.parse(message.data);
```

```
  console.log('got reply! ', dataFromServer);
  if (dataFromServer.type === "message") {
   this.setState((state) =>
    ({
     messages: [...state.messages,
     {
     msg: dataFromServer.msg,
     user: dataFromServer.user
     }]
    })
    );
  }
 };
}
```

Figura 24. Implementação React. Definição da lógica de abertura da conexão no lado do cliente

Por fim uma aplicação react precisa de um metodo ”render”para renderizar um HTML gerado pelo React, neste caso o render valida se o usuário está logado para saber se deve renderizar o canal de bate papo ou a tela de login.

É possível ver a execução da aplicação na (Figura 25).

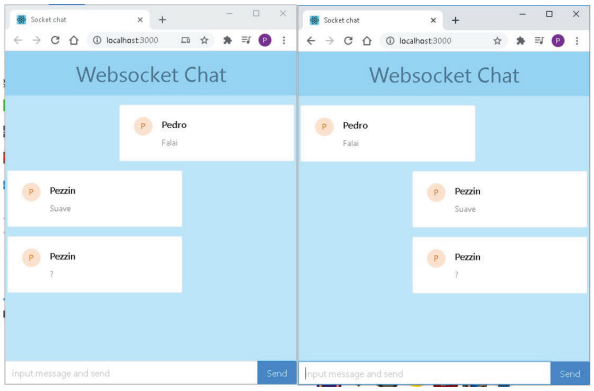


Figura 25. Execução da Aplicação React.

4.4. Comparação entre os cenários

Pode-se perceber que os três cenários são bem parecidos mesmo com suas complexidades diferentes, pois a finalidade deles é a mesma, a troca de mensagem com o servidor que funciona como um eco.

Embora esteja claro que os três cenários tem um objetivo apenas didático, é possível diferenciar o nível de aplicação no mundo real de cada um deles.

Comparando com os padrões da [W3C 2015], percebe-se que os três cenários seguem bem o que está definido, se compará-los com a interface Websocket definida na W3C como mostra a (Figura 5).

Poucos pontos são deixados de lado, como a lista de subprotocolos permitidos pelo servidor, onde apenas no segundo cenário foi especificada essa lista. Ou a implementação do onError, que a aplicação em react não implementou.

É possível observar que alguns steps de invocação do construtor do servidor não foram seguidos, por exemplo, o tratamento dos origins permitidos, onde apenas a aplicação react implantou.

E por fim o tratamento do tamanho da mensagem, na qual nenhum cenário aplica, onde na interface fica definida na variável bufferedAmount, que armazena o valor em bits do tamanho das mensagens trocadas e se for muito longo deveria solicitar o encerramento da conexão.

No geral observa-se que os três cenários por mais simples que sejam seguem bem os padrões definidos pela W3C, com poucas implementaç ões faltando, mesmo sendo muito simples para aplicá-los no mundo real a parte do Websocket está bem sólida, a api Websocket utilizada em todos os cenários deixam a implementação bem simples, onde basicamente se define os steps onOpen, onMessage, onClose e onError e a lógica da conexão.

Encontrou-se um artigo na Devmedia [Medeiros 2014] que contém um tutorial de implementação do Websocket em JEE e Glassfish, onde é apresentado um chat pare-cido com o do Tomcat apresentado anteriormente nesse artigo, com o acréscimo de uma autenticação do tipo BASIC para qualquer GET efetuado ao servidor. O artigo explica também como criptografar os dados e como exigir conexões WSS.

Mais recentemente foi publicado uma implementação de Websocket na Java Magazine [Juneau 2021] utilizando Java EE e Jakarta EE, nela encontra-se uma implementação um pouco mais sofisticada de um chat, onde é possóvel identificar alguns pontos de melhorias implementados em comparação com os cenários analisados, como por exemplo a utilização de mascaramento de dados com o encode e decode, adicionando uma segurança na troca de mensagens entre cliente e servidor.

Há também uma implementação Websocket consultada durante o desenvolvimento do artigo oferecida pelo [Tasarz 2016], a implementação utiliza o JavaEE e o Payara como servidor de aplicação.

4.5. Análise de segurança

Como a tecnologia Websocket foi pensada muito mais para demanda de desempenho(evitando eventuais chamadas do lado do cliente) do que na de segurança, ela tem alguns problemas que precisam ser resolvidos na implementação dos cenários.

Ataques DoS(muitas requisições de conexão ao mesmo tempo que sobrecarrega o servidor), falta de autorização e autenticação do cliente, canais TCP não criptografados, problemas com mascaramento de dados(dados originais ocultados, o que impede a distinção do dado trafegado), ataques de input de dados(injetar um script que será executado dentro do sistema) e tunneling(protocolo de rede encapsula um protocolo de carga diferente) são as vulnerabilidades mais comuns segundo o portal NeuraLegion [Kovacic 2020].

Nenhum dos cenários tratados neste artigo trabalham esses pontos de segurança, o terceiro cenário apresentado(com o front em react) implementa um tratamento para CORS(Cross-Origin Resource Sharing) e limita o tamanho do conteúdo enviado nas mensagens.

Segundo [Kovacic 2020] também apresenta soluções para as vulnerabilidades citadas. Para tratar da autenticação e autorização pode-se utilizar o Ticket-based authentication(Autenticação baseada em tíquete), onde o antes de abrir a conexão Websocket o servidor exige que o cliente crie um ticket contendo um ID de usuário, seu IP, um carimbo de data/hora e outros registros internos. Este ticket é armazenado no lado do servidor e retornado para o cliente que utilizará ele no handshake do upgrade para Websocket.

Sobre o tunneling é recomendado que não utilize, basta evita-lo sempre que possível, pois o Websocket não consegue lidar com a segurança dos dados trafegados nesse caso.

Para evitar ataques de input de dados é recomendado que utilize dados arbitrários, ou seja dados com um formato predefinido, esses dados precisam de validação, assim como qualquer outro que venha de um cliente antes de serem processados.

As mensagens enviadas pelo servidor também precisam de validação, sempre evite adicionar à DOM a resposta do servidor, utilize validadores de objetos, por exemplo em caso de uma mensagem em formato JSON, sempre utilize JSON.parse() para evitar que um script seja injetado no cliente.

Como recomendado também na W3C sempre deve-se utilizar origins com HTTPS, utilizando conexões wss.

5. Considerações Finais

O objetivo deste artigo foi analisar a tecnologia Websocket focando na parte da segurança da informação, por se tratar de uma tecnologia nova utilizada para troca de mensagens entre cliente e servidor na web, com a evolução das conexões até a Internet como é conhecida hoje trouxe novos conceitos que necessitam de melhorias. Verificando a proposta e implementação do Websocket, no decorrer do artigo foi demonstrado como a tecnologia funciona e como seria sua implementação, foram três cenários analisados durante o processo, estes com complexidades e linguagens distintas.

Durante a construção do artigo foi feita uma vasta pesquisa sobre a tecnologia abordada, que permitiu unir uma boa quantidade de informações que estão disponibilizadas no capítulo de referencial teórico, além de implementar três cenários em linguagens diferentes que aplicam o Websocket, que podem servir de exemplo de sua aplicação.

A proposta era analisar esses cenários e comparar suas implementações com a documentação de padrões definidas na W3C, verificando se seguem as recomendações de implementação, que por sinal conclui-se que sim, mesmos os cenários sendo relativamente simples, introdutórios e de aplicação didática, seguem muitas das definições recomendadas pela W3C, muito também por utilizarem bibliotecas que já trazem muitos dos conceitos implementados.

Porém a tecnologia Websocket traz com ela alguns pontos de segurança que precisam de atenção, dos quais os cenários analisados não se atentaram, como ataques DoS(denial of service), falta de autorização e autenticação do cliente, canais TCP não criptografados, problemas com mascaramento de dados, ataques de input de dados e tunneling, que são as vulnerabilidades mais comuns segundo [Kovacic 2020].

Dos conhecimentos adquiridos no curso de Análise e Desenvolvimento, foram utilizados, principalmente conceitos de redes de computadores, serviços de redes e segurança da informação, além da lógica de programação, a linguagem Java, programação web, estruturas de dados e metodologia de pesquisa, .

Como trabalhos futuros o artigo servirá de base para comparações mais específicas de implementações mais complexas desta tecnologia, já que possui uma boa base sobre a documentação da tecnologia e uma comparação a nível de recomendações dessas implementações, possibilitando novas comparações em outras áreas como o desempenho por exemplo. Há outros cenários citados no artigo que podem ser implementados, como o exemplo da revista Java Magazine [Juneau 2021] e o do Payara [Tasarz 2016].

Referências

Bishop, M. and Ed.Akamai. Hypertext transfer protocol version 3 (http/3). Acessado em 24-04-2021 ao URL https://quicwg.org/base-drafts/draft-ietf-quic-http.html.

Boadas, J. (2016). Apache tomcat websocket tutorial. Acessado em 17-08-2019 ao URL https://examples.javacodegeeks.com/enterprise-java/tomcat/apache-tomcat-websocket-tutorial/.

Caelum. O que É java ee? Acessado em 15-07-2020 ao URL https://www.caelum.com.br/apostila-java-web/o-que-e-java-ee.

Castillo, I. B. (2019). Websocket client and server implementation for node. Acessado em 17-08-2020 ao URL https://www.npmjs.com/package/websocket.

CCM (2017). O protocolo tcp. Acessado em 13-07-2020 ao URL https://br.ccm.net/contents/284-o-protocolo-tcp.

colMDN (2019). Javascript. Acessado em 15-07-2020 ao URL https://developer.mozilla.org/pt-BR/docs/Web/JavaScript.

colMDN (2020). Uma visão geral do http. Acessado em 15-07-2020 ao URL https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview.

Facebook (2020). React. Acessado em 15-07-2020 ao URL https://pt-br.reactjs.org/docs/getting-started.html.

Grigorik, I. and Surma. Introdução a http/2. Acessado em 24-04-2021 ao URL https://developers.google.com/web/fundamentals/performance/http2?hl=pt-br.

IETF (2011). The websocket protocol. Acessado em 07-07-2020 ao URL https://tools.ietf.org/html/rfc6455.

Java. O que é a tecnologia java e porque preciso dela? Acessado em 15-07-2020 ao URL https://www.java.com.

Juneau, J. (2021). How to build applications with the websocket api for java ee and jakarta ee. Acessado em 25-02-2021 ao URL https://blogs.oracle.com/javamagazine/how-to-build-applications-with-the-websocket-api-for-java-ee-and-jakarta-ee.

Kovacic, D. (2020). Websocket security: Top 7 websocket vulnerabilities. Acessado em 04-12-2020 ao URL https://www.neuralegion.com/blog/websocket-security-top-vulnerabilities/.

Leiner, B. M., Cerf, V. G., Clark, D. D., Kahn, R. E., Kleinrock, L., Lynch, D. C., Postel, J., Roberts, L. G., and Wolff, S. (1997). Brief history of the internet. Acessado em 07-07-2020 ao URL https://www.internetsociety.org/internet/history-internet/brief-history-internet/.

Lenon. Node.js – o que é, como funciona e quais as vantagens. Acessado em 05-09-2018 ao URL https://www.opus-software.com.br/node-js/.

McManus, P. Bootstrapping websockets with http/2. Acessado em 24-04-2021 ao URL https://datatracker.ietf.org/doc/html/rfc8441.

Medeiros, H. Java websockets: Introdução. Acessado em 14-06-2021 ao URL https://www.devmedia.com.br/java-websockets-introducao/30443.

Meenakshi, A. (2019). Websockets tutorial: How to go real-time with node and react. Acessado em 19-08-2019 ao URL https://blog.logrocket.com/websockets-tutorial-how-to-go-real-time-with-node-and-react-8e4693fbf843/.

RFC. The rfc series. Acessado em 15-07-2020 ao URL https://www.rfc-editor.org/.

RFC6455 (2011). The websocket protocol. Acessado em 13-07-2020 ao URL https://tools.ietf.org/html/rfc6455.

Tanenbaum, A. S. and Wetherall, D. (2011). *Redes de Computadores*. Pearson Education, Inc., 5th edition.

Tasarz, D. (2016). Jsf 2.3 - the websocket quickstart under payara server. Acessado em 26-03-2021 ao URL https://blog.payara.fish/jsf-2.3-the-websocket-quickstart-under-payara-server.

W3C. About w3c. Acessado em 15-07-2020 ao URL https://www.w3.org/Consortium/.

W3C (2015). The websocket api. Acessado em 13-07-2020 ao URL https://www.w3.org/TR/websockets/.

Documento Digitalizado Público

Artigo de Trabalho de Conclusão do Curso

Assunto: Artigo de Trabalho de Conclusão do Curso
Assinado por: Carlos Pagani
Tipo do Documento: Outro
Situação: Finalizado
Nível de Acesso: Público
Tipo do Conferência: Cópia Simples

Documento assinado eletronicamente por:

- **Carlos Eduardo Pagani**, **PROFESSOR ENS BASICO TECN TECNOLOGICO**, em 19/08/2021 17:47:43.

Este documento foi armazenado no SUAP em 19/08/2021. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifsp.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 748391
Código de Autenticação: 9e56486ad2