Sistemas Distribuídos
Desenvolvendo uma aplicação distribuída

Thiago O. da Silva[1], **Rhuan M. O. M. de Carvalho**[2]

[1]*Departamento de Ciência da Computação – Universidade Federal do Piauí (UFPI)*
Teresina – PI – Brazil

[2]*Departamento de Ciência da Computação – Universidade Federal do Piauí (UFPI)*
Teresina – PI – Brazil Teresina – PI – Brazil

{thiago_silva,rhuanmourao}@ufpi.edu.br

Abstract. *This report aims to develop a distributed system application, considering the architectural styles of both the client and server parts.*

Resumo. *Este relatório tem como objetivo desenvolver uma aplicação de um sistema distribuído, levando em consideração os estilos arquitetônicos tanto da parte cliente quanto da parte servidora.*

1. Definição das tecnologias

A linguagem de programação e as tecnologias que serão utilizadas na implementação do aplicativo de bate-papo são:

- Linguagem de programação: Node.js
- Tecnologias: Express.js, Socket.IO, HTML, CSS e JavaScript

Node.js é uma linguagem de programação de código aberto baseada em eventos e um modelo de servidor para criar aplicações escaláveis e eficientes para a web. Esse modelo de eventos é um modelo de programação que permite que os processos sejam executados de forma assíncrona. Isso significa que os processos não precisam ser executados um após o outro, mas podem ser executados ao mesmo tempo, garantindo o paralelismo dos processos.

Express.js é uma estrutura de aplicativo web para Node.js que facilita a criação de aplicativos web simples e eficientes.

Socket.IO é um framework de comunicação em tempo real para aplicações web que facilita a comunicação entre clientes e servidores.

HTML, CSS e JavaScript são tecnologias usadas para criar interfaces de usuário para aplicações web.

A arquitetura escolhida para o aplicativo de bate-papo é uma arquitetura cliente-servidor. Isso significa que o aplicativo terá um servidor que processará as solicitações dos clientes e será armazenado em uma máquina separada dos Clientes. Portanto, essa arquitetura é adequada para sistemas distribuídos porque permite que o servidor seja facilmente escalado para atender a um maior número de clientes.

Contudo, essas tecnologias escolhidas são adequadas para a construção de sistemas distribuídos porque são escaláveis, eficientes e compatíveis com a arquitetura cliente-servidor.

2. Implementação da parte cliente

O cliente é implementado usando HTML, CSS e JavaScript. O HTML é usado para criar a estrutura da página da web, o CSS é usado para estilizar a página da web e o JavaScript é usado para adicionar interatividade à página da web.

Além disso, o cliente é capaz de se comunicar com o servidor e realizar as operações necessárias, sendo possível enviar e receber mensagens, conectar e desconectar do servidor.

2.1. Registro de usuário

Quando o usuário clica no botão ”Registrar”, o cliente envia a solicitação de registro para o servidor. O servidor então verifica o nome de usuário e a senha e, se forem válidos, o servidor registra o usuário e o redireciona para a página principal. Se o nome de usuário ou a senha forem inválidos, o servidor envia um erro para o cliente.

```
const registerButton = document.getElementById("register");
registerButton.addEventListener("click", () => {
  const username = document.getElementById("username").value;
  const password = document.getElementById("password").value;

  // Envia a requisição de registro ao servidor
  socket.emit("register", {
    username,
    password,
  });
});

// Quando o servidor emite o evento "registration", redireciona o usuário ao menu
↪  principal
socket.on("registration", (data) => {
  if (data.success) {
    window.location.href = "/";
  } else {
    alert(data.error);
  }
});
```

2.2. Comunicação com o servidor

O cliente se comunica com o servidor usando o protocolo WebSocket. Quando o usuário clica no botão ”Enviar”, o cliente envia a mensagem para o servidor. Por conseguinte, o servidor então envia a mensagem para todos os outros usuários conectados ao aplicativo. Os outros usuários então veem a mensagem na lista de mensagens.

```
const socket = io();

// Quando o usuário clica no botão "Send", a mensagem é enviada ao servidor
const sendButton = document.getElementById("send");
sendButton.addEventListener("click", () => {
  const username = document.getElementById("username").value;
  const message = document.getElementById("message").value;

  //Envia a mensagem ao servidor
  socket.emit("message", {
    username,
    message,
  });
});

// Quando o servidor emite o evento "message", acrescenta a mensagem na lista de
↪  mensagens
socket.on("message", (data) => {
  const li = document.createElement("li");
  li.innerText = data.username + ": " + data.message;
```

```
  document.getElementById("messages").appendChild(li);
});
```

3. Implementação da parte do servidor

O servidor é capaz de lidar com várias requisições simultâneas porque é implementado usando o framework Express.js, um framework web que é projetado para ser escalável e eficiente.

3.1. Gerenciamento de conexões

O servidor é implementado usando Express.js e Socket.IO. O Express.js é um framework web para Node.js que facilita a criação de aplicativos web simples e eficientes. Paralelamente, o Socket.IO é um framework de comunicação em tempo real para aplicações web que facilita a comunicação entre clientes e servidores.

3.2. Lógica do chat

A lógica do chat é implementada na função register. Essa função verifica o nome de usuário e a senha e, se forem válidos, o servidor registra o usuário e o redireciona para a página principal. Se o nome de usuário ou a senha forem inválidos, o servidor envia um erro para o cliente.

3.3. Persistência dos dados

O banco de dados foi implementado usando o módulo mysql do Node.js, que fornece uma API para se conectar a bancos de dados MySQL e executar consultas. Com isso, o módulo mysql foi usado para se conectar a um banco de dados MySQL chamado chat e criar duas tabelas:

- Tabela users: Armazena os detalhes dos usuários, como nome de usuário e senha.
- Tabela messages: Armazena as mensagens do chat.

Essas tabelas permitem que os usuários se registrem no aplicativo, enviem e recebam mensagens e visualizem a lista de usuários do aplicativo.

3.4. Código

```
const express = require("express");
const io = require("socket.io");
const cors = require("cors");
const mysql = require("mysql");

const app = express();

app.use(cors());

// Cria o servidor Socket.IO
const server = io.listen(3000);

// Cria uma conexão com o banco de dados MySQL
const connection = mysql.createConnection({
  host: "localhost",
  user: "root",
  password: "password",
  database: "chat",
});
```

```
// Define a lógica do chat
app.post("/register", (req, res) => {
  // Autenticar o username e password
  const { username, password } = req.body;

  // Se o username e o password forem válidos, registrar o usuário
  if (username && password) {
    // Inserior o usuário no banco de dados
    connection.query(
      `INSERT INTO users (username, password) VALUES (?, ?)`,
      [username, password],
      (err, rows) => {
        if (err) {
          res.status(500).json({ error: err.message });
        } else {
          res.json({ success: true });
        }
      }
    );
  } else {
    res.status(400).json({ error: "Usuário ou senha inválidos" });
  }
});

const users = connection.query("SELECT * FROM users");

// Define a comunicação com os clientes
server.on("connection", (socket) => {
  // Quando um cliente conecta, adicione na lista de usuários
  users.push({
    username: socket.handshake.username,
  });

  // Quando um cliente enviar uma mensagem, salvar no banco de dados
  socket.on("message", (message) => {
    connection.query(
      `INSERT INTO messages (username, message) VALUES (?, ?)`,
      [socket.handshake.username, message],
      (err, rows) => {
        if (err) {
          console.error(err);
        }
      }
    );
  });

  // Quando um cliente se desconectar, remover da lista de usuários
  socket.on("disconnect", () => {
    users = users.filter(user => user.id !== socket.id);
  });
});

app.listen(3000, () => {
  console.log("App listening on port 3000");
});
```

4. Processo de implementação

4.1. Decisões tomadas

Ao implementar o chat, foram tomadas as seguintes decisões:

- Usar Node.js como a linguagem de programação do servidor.
- Usar WebSockets para comunicação bidirecional entre o servidor e os clientes.
- Usar o React.js como o framework de front-end.
- Usar o banco de dados MySQL.

4.2. Desafios enfrentados

Os desafios mais significativos enfrentados durante a implementação foram:

- Criar uma arquitetura que fosse escalável e confiável, podendo lidar com um grande número de usuários.
- Garantir que o aplicativo seja seguro para usar.

4.3. Soluções adotadas

Para superar esses desafios, adotamos as seguintes soluções:

- Foi usado uma arquitetura distribuída com vários servidores. O Express.js é um framework web escalável que é projetado para lidar com um grande número de usuários.
- Para tornar o aplicativo seguro, o protocolo HTTPS e o framework Express.js foram usados. O protocolo HTTPS é um protocolo de segurança que criptografa as conexões entre o cliente e o servidor. O framework Express.js fornece suporte para segurança, como autenticação e autorização.

4.4. Arquitetura implementada

A arquitetura do chat é distribuída com vários servidores. O servidor principal é responsável por lidar com as conexões dos clientes e distribuir as mensagens para os servidores de back-end. Os servidores de back-end são responsáveis por armazenar as mensagens e fornecer serviços de autenticação e autorização. Com isso, esses servidores se comunicam entre si usando a rede, permitindo que o aplicativo seja executado em várias máquinas e atenda a um grande número de usuários.

4.5. Tecnologias e padrões usados

As tecnologias e padrões usados na implementação do chat incluem:

- Node.js
- WebSockets
- HTML e CSS
- MySQL
- Express.js
- HTTPS