Mark Pimentel Matheus Ribeiro

Uso da linguagem Rust para paralelismo em servidores de jogos

PROJETO FINAL

DEPARTAMENTO DE INFORMÁTICA
Programa de Graduação em Ciência da Computação

Rio de Janeiro
Julho de 2023

Pontifícia Universidade Católica
do Rio de Janeiro

Mark Pimentel Matheus Ribeiro

Uso da linguagem Rust para paralelismo em servidores de jogos

Relatório de Projeto Final

Relatório de Projeto Final, apresentado ao Programa de Ciência da Computação, do Departamento de Informática da PUC-Rio como requisito parcial para a obtenção do titulo de Bacharel em Ciência da Computação.

Orientador: Prof. Noemi de La Rocque Rodriguez

Rio de Janeiro
Julho de 2023

Mark Pimentel Matheus Ribeiro

Ficha Catalográfica

Ribeiro, Mark

Uso da linguagem Rust para paralelismo em servidores de jogos / Mark Pimentel Matheus Ribeiro; orientador: Noemi de La Rocque Rodriguez. – 2023.

40 f: il. color. ; 30 cm

Projeto Final - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2023.

Inclui bibliografia

1. Informática – Teses. 2. Rust. 3. paralelismo. 4. servidores. 5. jogos. I. Rodriguez, Noemi. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

Aos meus pais Edna Pimentel Matheus e Mauricio Ribeiro Sobrinho pelo apoio e suporte ao longo de toda faculdade.

A minha orientadora, Noemi Rodrigues pela inspiração e paciência não só durante esse projeto mas ao longo de toda a graduação.

Aos meus amigos Victor Martins, Maria Eduarda Venancio, Eduardo Ramanauskas, Nagib Suaid, Raks Olhovetchi, Mariana Rangel, Arthur Ozorio, Luiza Arantes, Gustavo Camerano, Luca Salgado, Daniel Oliveira, Nelson Donato, Felipe Metson, Danilo Catalão, Jonny Russo, Juliana Prado, Yuri Lemos, Michelle Santiago, Fernando dos Santos, João Pedro Kalil, Pedro Henrique Soares, Pedro Chamberlain, Fernanda Costa, Gabriela Ladeira, Carolina Cunha e muitos outros que fiz ao longo desses anos de curso, não teria chegado até aqui sem nenhum de vocês, muito obrigado por sempre terem acreditado no meu potencial muitas vezes até mais que eu mesmo.

Resumo

Ribeiro, Mark; Rodriguez, Noemi. **Uso da linguagem Rust para paralelismo em servidores de jogos**. Rio de Janeiro, 2023. 40p. Projeto Final – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Atualmente alguns dos jogos mais famosos do mercado comportam centenas de jogadores competindo online em simultâneo, por conta disso cada vez mais as empresas de jogos se preocupam em produzir jogos e serviços online. Tendo isso em mente, paralelismo se mostra necessario e neste projeto iremos explorar as facilidades de se usar Rust para a criacáo destes tais servidores.

Palavras-chave

Rust; paralelismo; servidores; jogos.

Abstract

Ribeiro, Mark; Rodriguez, Noemi (Advisor). **Use of Rust language for parallelism in video game servers**. Rio de Janeiro, 2023. 40p. Projeto Final – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Currently, some of the most famous games in the market accommodate hundreds of players competing online simultaneously. As a result, game companies are increasingly concerned about producing online games and services. Keeping this in mind, parallelism proves to be necessary, and in this project, we will explore the advantages of using Rust for the creation of these servers.

Keywords

Rust; parallelism; servers; games.

Sumário

Lista de figuras

Lista de Códigos

Lista de Abreviaturas

CMD – Command Prompt

TCP – Transmission Control Protocol

1
Introdução

Em 1997, antes do advento da banda larga, os jogos multiplayer online não eram tão populares devido à qualidade limitada das redes disponíveis na época. No entanto, com o surgimento da banda larga, o interesse do público por esse estilo de jogo aumentou significativamente, permitindo que diversos jogadores de várias partes do globo competissem entre si. Desde então, esse estilo de jogo continua sendo um sucesso.

Em 2021, jogos multiplayer ficaram tão populares que a grande maioria dos títulos novos têm algum tipo de recurso multiplayer online. Chegamos a ver casos onde o jogo todo se passa online com outros jogadores e não existe uma versão solo offline.

Devido ao volume crescente de jogadores, é essencial desenvolver servidores de jogos que possam aproveitar ao máximo o hardware disponível. Embora já tenhamos superado a barreira inicial de desenvolver servidores de jogos, a capacidade de acomodar um grande número de jogadores simultaneamente ainda é um desafio. Portanto, é crucial investir na otimização de servidores para garantir uma experiência de jogo mais fluida e satisfatória para os usuários.

O trabalho de A. Abdelkhalek e A. Bilas(1) investiga o comportamento e a escalabilidade de um servidor de jogo multiplayer interativo, o Quake. O objetivo é entender o comportamento desse tipo de aplicação e fornecer um novo benchmark para servidores escaláveis implementando e avaliando uma versão paralela do servidor.

O trabalho aponta que a largura de banda da rede não é um problema, pois o servidor de jogo circula apenas informações de controle e todas as operações gráficas são lidas pelos clientes. Além disso, as exigências de memória também não são um problema para sistemas modernos.

No entanto, a implementação de um servidor paralelo apresenta desafios.

Durante a análise do desempenho do servidor paralelo, foi observado um grande potencial para melhorias em termos de arquitetura do microprocessador. Embora não tenhamos nos concentrado especificamente no desempenho nesse projeto, foi possível identificar áreas que poderiam ser aprimoradas para obter um melhor aproveitamento dos recursos de processamento.

Além disso, a implementação do servidor paralelo trouxe à tona alguns desafios relacionados ao equilíbrio de carga entre as *threads* e ao tempo de sincronização. Esses aspectos são cruciais para garantir um processamento adequado e correto do jogo em um ambiente paralelo. A decomposição das tarefas de forma eficiente e a sincronização adequada entre as *threads* são áreas que requerem atenção para garantir a escalabilidade do servidor e um desempenho otimizado.

É importante destacar que, embora esses desafios tenham sido identificados, eles não comprometem os resultados promissores alcançados com a versão paralela do servidor utilizando a linguagem Rust. Na seção de conclusões, esses desafios serão retomados e discutidos em maior detalhe, destacando as abordagens utilizadas e as possíveis soluções para enfrentá-los.

Em resumo, os resultados obtidos até o momento mostram o potencial da implementação paralela do servidor em termos de desempenho e escalabilidade. No entanto, a decomposição eficiente das tarefas e a sincronização correta entre as *threads* surgiram como principais desafios, que serão aprofundados e discutidos posteriormente para explorar como a linguagem Rust aborda essas questões.

Para maximizar a escalabilidade, é necessário trabalhar nas principais limitações, incluindo a atribuição dinâmica de carga de trabalho a *threads*, a redução de desequilíbrios de carga e a reestruturação da arquitetura interna.

A programação dessas técnicas é complexa e sujeita a particularidades do ambiente de programação utilizado. Neste trabalho, investigamos como a linguagem Rust, com suas ferramentas e particularidades, pode facilitar o,

desenvolvimento de servidores *multithread*.

Rust é uma linguagem de programação moderna e eficiente que foi projetada com segurança, desempenho e escalabilidade em mente, tornando-a uma escolha convidativa para a realização de tarefas paralelas. Com suporte nativo para concorrência, é possível dividir o trabalho em várias *threads*. que podem ser executadas simultaneamente, aproveitando ao máximo o poder do hardware disponível. Além disso, a linguagem tem ferramentas e mecanismos incorporados que garantem que o código seja livre de falhas comuns, como corrupção de memória e travamentos, o que é crucial para aplicações paralelas.

Para investigar as funcionalidades e facilidades de Rust, neste projeto vou desenvolver um servidor de jogos com paralelismo em Rust. Além desse servidor, também criarei um jogo em javascript que o servidor será responsável por atualizar e uma ferramenta de teste, também em Rust, que permitirá testar o servidor com muitos jogadores.

2
Visão Geral

2.1
Desafios

Na área de programação paralela, um dos desafios mais significativos é lidar com a necessidade de sincronização entre *threads*. O modelo mais comum é o de memória compartilhada, onde várias *threads* acessam e modificam os mesmos dados simultaneamente. No entanto, essa abordagem pode levar a problemas graves de sincronização.

Além disso, as linguagens tradicionais muitas vezes apresentam uma sintaxe complexa ao lidar com recursos de paralelismo e concorrência. Refatorar ou entender o código após algum tempo de sua criação pode se tornar uma tarefa inviável devido à complexidade da sintaxe.

Diante desses desafios, algumas linguagens têm buscado facilitar o trabalho dos desenvolvedores no contexto de programação paralela. Uma dessas linguagens é Rust, que combina segurança de memória e alto desempenho. Rust oferece um sistema de tipos poderoso e um modelo de propriedade de empréstimo que permite evitar erros comuns de sincronização e garantir a segurança do acesso a dados compartilhados entre *threads*.

Com sua abordagem inovadora, Rust se destaca como uma opção promissora para lidar com a complexidade da programação paralela, fornecendo ferramentas que facilitam o desenvolvimento de sistemas paralelos seguros e eficientes.

2.2
Rust

Rust é uma linguagem de programação desenvolvida pela Mozilla Research, projetada para ser "segura, concorrente e prática". Ela se baseia em três princípios fundamentais: segurança sem coletor de lixo, concorrência sem cor-

ridas de dados e abstração sem *overhead*. Esses princípios permitem que Rust seja eficiente tanto em aplicações de baixo nível quanto em projetos de alto nível.

Um dos aspectos distintivos de Rust é sua abordagem para evitar falhas de segmentação, dificultando o uso de ponteiros nulos ou ponteiros soltos. A linguagem gerencia automaticamente a memória e os recursos, garantindo a segurança e prevenindo erros comuns. Além disso, Rust torna difícil a ocorrência de condições de corrida, onde duas *threads* tentam modificar o mesmo valor simultaneamente. Para isso, oferece diversas técnicas seguras de comunicação entre *threads*, como canais e troca de mensagens.

Ao lidar com paralelismo, Rust adota uma abordagem diferenciada na passagem de mensagens entre *threads*. A linguagem não permite o compartilhamento de objetos mutáveis, o que evita problemas de modificação concorrente indesejada. Essa restrição é uma medida de segurança importante para garantir a integridade dos dados.

Além disso, em Rust, o acesso a ponteiros não é tão direto como em outras linguagens, pois a prioridade da linguagem é fornecer um ambiente seguro de desenvolvimento para os usuários. Isso impede o uso inadequado de ponteiros e contribui para a prevenção de erros de memória e vazamentos.

No contexto de servidores de jogos, onde é fundamental ter um grande número de jogadores interagindo simultaneamente, o uso de Rust tem o potencial de simplificar como esses servidores são desenvolvidos. Sua abordagem segura, eficiente e concorrente, aliada à capacidade de troca de mensagens entre *threads*, possibilita a construção de servidores de jogos robustos e escaláveis, proporcionando uma experiência de jogo mais fluída e interativa para os jogadores.

3
Objetivo e Proposta do Trabalho

O projeto proposto consiste em três partes distintas, que serão abordadas da seguinte forma:

1. Construção de um servidor *multithread* em Rust: O objetivo dessa parte do projeto é desenvolver um servidor *multithread* voltado para jogos utilizando a linguagem Rust. Serão exploradas as facilidades oferecidas pela linguagem para lidar com concorrência e paralelismo. Serão utilizados recursos como canais de comunicação entre *threads*, mutexes e semáforos para garantir a sincronização adequada e evitar condições de corrida. Essa etapa será uma oportunidade de estudar as vantagens e características únicas da linguagem Rust no contexto de servidores *multithread*.

2. Construção de um cliente em JavaScript: Nessa parte do projeto, será desenvolvido um cliente em JavaScript que funcionará como um jogo. O cliente fará um grande número de requisições ao servidor, simulando a interação de diversos jogadores. Essa abordagem permitirá testar e avaliar a eficiência do servidor em lidar com múltiplas conexões simultâneas. Serão utilizadas as capacidades de comunicação em tempo real oferecidas pelos WebSockets para estabelecer uma comunicação bidirecional entre o cliente e o servidor. Serão implementadas as funcionalidades do jogo, como movimentação dos personagens e pintura dos quadrados no mapa, para fornecer uma experiência interativa aos jogadores.

3. Criação de mecanismos de teste de desempenho: Essa etapa visa criar mecanismos de teste de desempenho para avaliar o funcionamento do servidor *multithread* em Rust. Serão desenvolvidos programas de teste que simulem um grande número de jogadores conectados simultaneamente ao servidor, gerando uma carga de requisições significativa. Serão medi-

dos e analisados indicadores de desempenho, como tempo de resposta, capacidade de processamento e consumo de recursos, a fim de avaliar a eficiência e a escalabilidade do servidor. Esses testes serão fundamentais para validar a capacidade do servidor em lidar com uma carga realista de jogadores e identificar possíveis gargalos de desempenho.

Ao final do projeto, o objetivo é demonstrar por meio das partes desenvolvidas o quão fácil é criar um servidor *multithread* de jogos em Rust. Será apresentado um servidor robusto, capaz de lidar com um grande número de jogadores simultâneos, além de oferecer um cliente em JavaScript que proporcione uma experiência de jogo interativa. Os testes de desempenho permitirão avaliar a eficiência do servidor em condições reais, fornecendo dados concretos sobre seu funcionamento.

4
Atividades realizadas

4.1
Início

As versões do servidor, jogo e ferramenta de testes desenvolvidas para este projeto estão disponíveis neste repositório do GitHub[1].

Para iniciar o projeto, realizou-se um estudo aprofundado da linguagem Rust, dada sua natureza peculiar, tanto em termos semânticos quanto comportamentais, que a diferencia das linguagens comumente utilizadas, como C e Python. A abordagem preferencial adotada para a familiarização com a linguagem consistiu na consulta à documentação oficial, "The Rust Programming Language"(2), devido à sua qualidade didática e à facilidade de compreensão proporcionada pelo material.

O referido livro apresenta, intencionalmente, erros concebidos para ilustrar o funcionamento das peculiaridades da linguagem, além de conter capítulos dedicados explicitamente a servidores e à comunicação com clientes. Especificamente, um desses capítulos desempenhou um papel fundamental no desenvolvimento deste projeto, pois ofereceu um exemplo básico de servidor e cliente, que serviu como base para a implementação aqui proposta. No entanto, vale ressaltar que o processo de aprendizagem ocorreu concomitantemente à execução do projeto, o que exigiu consultas frequentes à documentação oficial de Rust e leituras complementares.

No entanto, apenas o domínio da linguagem não foi o bastante; foi essencial adquirir um embasamento teórico sobre servidores, compreendendo seu funcionamento e como se comunicam com os clientes. Esse processo foi realizado por meio do estudo de exemplos de servidores e tentativas de replicação dos mesmos. Adicionalmente, destaca-se a relevância da disciplina

[1] https://github.com/MarkRibeiro/Uso_da_linguagem_Rust_para_paralelismo_em_servidores_de_jogos

INF1406 - Programação Distribuída Concorrente, que desempenhou um papel fundamental nesta etapa da pesquisa, ao propor a criação de diversos servidores e clientes utilizando a linguagem Rust.

Por fim, foi necessário adquirir conhecimento sobre os processos de teste de servidores e compreender os critérios que determinam sua eficiência. Realizou-se uma pesquisa abrangente sobre as ferramentas de teste disponíveis para servidores, buscando compreender sua aplicabilidade no contexto deste projeto. Além disso, foram exploradas as abordagens para desenvolver uma ferramenta de teste adequada às necessidades específicas da presente pesquisa.

4.2 Configurando o Ambiente

Antes de iniciar a codificação do servidor propriamente dita, foi necessário configurar o ambiente de desenvolvimento para Rust. Esse processo envolveu o download e a instalação da linguagem por meio do site oficial. A instalação foi realizada de forma intuitiva, bastando baixar o instalador e seguir as opções "next" algumas vezes. Após a conclusão desse processo, o computador ficou pronto para compilar códigos em Rust.

Após essa etapa, tornou-se viável compilar códigos em Rust e executar o programa por meio do prompt de comando (CMD) do Windows. Inicialmente, o projeto foi desenvolvido seguindo esse método descrito acima. No entanto, essa abordagem revelou-se demorada e pouco prática. Consequentemente, foi necessário realizar uma pesquisa em busca de IDEs compatíveis com Rust.

Inicialmente, a pesquisa revelou que as IDEs Visual Studio e Eclipse pareciam ser as melhores opções disponíveis no mercado. No entanto, devido à falta de compatibilidade nativa com Rust ou à sua limitada integração, foi necessário aprofundar ainda mais a pesquisa. Após uma análise criteriosa, a IDE que melhor atendia aos requisitos do projeto acabou sendo o IntelliJ[2].

Desenvolvido pela JetBrains, o IntelliJ é uma IDE de destaque que pro-

[2]https://www.jetbrains.com/idea/

porciona uma experiência de desenvolvimento de software altamente robusta e abrangente. Amplamente adotada por desenvolvedores em âmbito global, essa ferramenta é essencial para aqueles que almejam aumentar sua produtividade e eficiência no processo de criação de software. Seus recursos e funcionalidades contribuíram para uma experiência de desenvolvimento mais agradável.

Com o ambiente de desenvolvimento devidamente configurado, deu-se início à implementação do servidor. Nesse processo, foram realizadas numerosas consultas à documentação de Rust, bem como leituras complementares, a fim de aprofundar o entendimento das particularidades da linguagem. Essas atividades foram fundamentais para assegurar um desenvolvimento consistente e adequado do servidor.

Devido à ênfase do servidor no aspecto de paralelismo, foi tomada a decisão de atribuir uma *thread* para representar o estado atual do jogo e uma *thread* adicional para cada jogador. Essa abordagem permitirá que o servidor gerencie simultaneamente múltiplos jogadores e atualize o estado do jogo de forma paralela, garantindo uma execução eficiente e otimizada.

4.3 Ideia do jogo criado

O jogo final desenvolvido para este projeto é uma versão minimalista e com visualização *top-down* (vista de cima) inspirada em um jogo do Nintendo Switch chamado Splatoon. No jogo original da Nintendo, existem duas equipes compostas por dois a oito jogadores cada. Cada equipe possui cores distintas, sendo as principais opções roxo e amarelo, conforme ilustrado na figura 4.1.

O objetivo do jogo é pintar a maior porcentagem possível do mapa com as cores da equipe. A equipe que conseguir pintar a maior área com suas cores será a vencedora. Essa mecânica de jogo traz uma abordagem divertida e competitiva, onde os jogadores devem trabalhar em equipe para conquistar territórios e dominar o mapa.

No contexto deste projeto, uma versão simplificada desse conceito foi

Figura 4.1: Splatoon 3, jogo referencia do Nintendo Switch

implementada, mantendo o aspecto de competição entre as equipes, porem sendo compostas só por um jogador cada, e a mecânica de pintura do mapa com as cores respectivas. O objetivo principal foi testar e demonstrar a capacidade do servidor e do cliente de se comunicarem em tempo real, permitindo que vários jogadores participem simultaneamente e interajam uns com os outros em um ambiente *multiplayer*.

4.4 Criar Servidor

Como o servidor modelado pretende ser usado em jogos *multiplayer*, foram criadas estruturas de dados específicas para lidar com essa finalidade. Essas estruturas foram projetadas para armazenar e manipular informações relevantes ao jogo, como dados dos jogadores, estado do jogo, pontuações, entre outros elementos necessários para a interação entre o servidor e os clientes. Essas estruturas de dados foram projetadas visando otimizar o desempenho e garantir a integridade dos dados durante o processamento no servidor.

Foi desenvolvida a estrutura de dados chamada "Point" (Ponto). Essa estrutura representa um quadrado no tabuleiro do jogo e é composta por dois elementos essenciais:

- Um atributo "x" do tipo usize, que armazena a posição horizontal do ponto no eixo cartesiano.
- Um atributo "y" do tipo usize, que armazena a posição vertical do ponto no eixo cartesiano.

Esses elementos permitem identificar a localização exata do ponto dentro do tabuleiro do jogo, facilitando a manipulação e a interação com outros elementos ou jogadores presentes no mesmo. A estrutura de dados "Point" foi projetada visando fornecer uma representação eficiente e precisa das posições dos quadrados no tabuleiro do jogo.

Foi desenvolvida a estrutura de dados chamada "Player" (Jogador). Essa estrutura representa um jogador dentro do jogo e é composta por cinco elementos essenciais:

- Um atributo "name" do tipo string, responsável por armazenar o nome desse jogador.
- Um atributo "id" do tipo usize, responsável por armazenar um número identificador único para esse jogador.
- Um atributo "color" do tipo string, responsável por armazenar a cor desse jogador no formato hexadecimal.
- Um atributo "position" do tipo "Point" (Ponto), responsável por armazenar a posição atual desse jogador no tabuleiro do jogo.
- Um atributo "score" do tipo u32, responsável por armazenar a pontuação atual desse jogador.

Esses elementos permitem representar e armazenar informações importantes sobre cada jogador, incluindo seu nome, identificação, cor, posição no tabuleiro e pontuação. A estrutura "Player" foi projetada para facilitar o gerenciamento e a interação dos jogadores dentro do jogo, permitindo o acesso e a atualização dessas informações de forma eficiente.

Foi desenvolvida a estrutura de dados chamada "State" (Estado). Essa estrutura representa o estado completo do mapa do jogo e é composta por seis elementos essenciais:

- Um vetor de objetos "Player" chamado "players", responsável por armazenar todas as informações de todos os jogadores presentes no jogo.
- Uma matriz de strings chamada "map", responsável por armazenar as cores de cada quadrado do tabuleiro.
- Um atributo "canvas_height" do tipo usize, responsável por armazenar a altura do tabuleiro.
- Um atributo "canvas_width" do tipo usize, responsável por armazenar a largura do tabuleiro.
- Um atributo "match_time" do tipo u64, responsável por armazenar a duração de uma partida em segundos.
- Um atributo booleano chamado "game_is_over", responsável por indicar se a partida terminou ou não.

Esses elementos juntos representam o estado atual do jogo, incluindo informações dos jogadores, cores dos quadrados do tabuleiro, dimensões do tabuleiro, tempo da partida e o estado de conclusão da partida. A estrutura "State" é responsável por manter e atualizar todas essas informações durante a execução do jogo.

No capítulo anterior, mencionou-se que a criação do servidor se baseou em um exemplo de servidor em Rust disponibilizado na documentação oficial da linguagem. O código utilizado foi o seguinte:

Código 1: Exemplo de servidor Rust

```
1 use hello::ThreadPool;
2 use std::fs;
3 use std::io::prelude::*;
4 use std::net::TcpListener;
```

```
use std::net::TcpStream;
use std::thread;
use std::time::Duration;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:7878").unwrap();
    let pool = ThreadPool::new(4);

    for stream in listener.incoming().take(2) {
        let stream = stream.unwrap();

        pool.execute(|| {
            handle_connection(stream);
        });
    }

    println!("Shutting down.");
}

fn handle_connection(mut stream: TcpStream) {
    let mut buffer = [0; 1024];
    stream.read(&mut buffer).unwrap();

    let get = b"GET / HTTP/1.1\r\n";
    let sleep = b"GET /sleep HTTP/1.1\r\n";

    let (status_line, filename) = if buffer.starts_with(get) {
        ("HTTP/1.1 200 OK", "hello.html")
    } else if buffer.starts_with(sleep) {
        thread::sleep(Duration::from_secs(5));
        ("HTTP/1.1 200 OK", "hello.html")
    } else {
        ("HTTP/1.1 404 NOT FOUND", "404.html")
    };

    let contents = fs::read_to_string(filename).unwrap();

    let response = format!(
```

```
43          "{}\r\nContent-Length: {}\r\n\r\n{}",
44          status_line,
45          contents.len(),
46          contents
47      );
48
49      stream.write_all(response.as_bytes()).unwrap();
50      stream.flush().unwrap();
51  }
```

A partir dessa base inicial, foram realizados incrementos e modificações no código do servidor para torná-lo mais compatível com os requisitos e objetivos do projeto em questão.

Neste exemplo de código, é criado um servidor web simples capaz de lidar com solicitações HTTP GET. O servidor está configurado para escutar na porta 7878 do endereço IP local (127.0.0.1). Quando uma solicitação é recebida, o servidor lida com ela em uma *thread* separada, utilizando um pool de *threads*.

O servidor realiza verificações adicionais na solicitação recebida para determinar o tipo de resposta a ser enviada. Se a solicitação for para obter um arquivo específico ou para fazer o servidor dormir, o servidor realiza ações diferentes. Caso contrário, uma resposta de erro 404 é enviada.

Quando o arquivo é lido com sucesso, o servidor gera uma resposta HTTP completa que inclui o status da resposta, o comprimento do conteúdo e o próprio conteúdo do arquivo. Essa resposta é então enviada de volta para o cliente.

Na versão final, foi desenvolvido um servidor de WebSocket em Rust que implementa um jogo *multiplayer*. Esse servidor é capaz de aceitar conexões WebSocket na porta 3012 e mantém uma lista de jogadores conectados. A cada 100 milissegundos, o servidor envia continuamente uma mensagem JSON contendo o estado do jogo para todos os jogadores conectados.

O jogo em si possui um mapa bidimensional com largura e altura

especificadas pelo usuário (ou um valor padrão, caso não sejam especificados), e também uma duração de partida especificada pelo usuário (ou um valor padrão, se não for especificada).

A função create_map cria um mapa vazio com altura e largura especificadas.

Código 2: Função create_map

```
fn create_map(height:usize, width:usize) -> Vec<Vec<String>> {

  let mut matrix = vec![];
  let mut vector = vec![];
  for _ in 0..height {
    vector.push("white".to_string());
  }
  for _ in 0..width {
    matrix.push(vector.clone());
  }
  return matrix;
}
```

A função handle_connection lida com a conexão WebSocket de um jogador, adicionando-o à lista de jogadores e recebendo mensagens do jogador. Quando uma mensagem é recebida, ela é interpretada como um movimento do jogador e atualiza a posição do jogador no estado do jogo. Quando um jogador é desconectado, ele é removido da lista de jogadores.

Código 3: Função handle_connection

```
fn handle_connection(websocket: Arc<Mutex<WebSocket<TcpStream>>>,
    current_state: Arc<Mutex<State>>){
  println!("Conectei");
  loop {
    let msg = websocket.lock().unwrap().read_message();
    match msg{
      Ok(conteudo) => {
        let copy_current_state = current_state.clone();
        /*{
```

```
9               let mut state = current_state.lock().unwrap();
10              *state = msg.clone();
11          }*/
12          _process_message(websocket.clone(), conteudo,
                copy_current_state);
13      }
14      Err(Error::Io(ref e)) if e.kind() == io::ErrorKind::WouldBlock
            => {
15          //println!("WouldBlock");
16          continue;
17      },
18      Err(Error::Io(ref e)) if e.kind() == io::ErrorKind::
            ConnectionReset => {
19          println!("Connection reset");
20      },
21      Err(Error::AlreadyClosed) => {
22          println!("Conexao encerrada");
23          break;
24      },
25      Err(Error::ConnectionClosed) => {
26          println!("Conexao encerrada");
27          break;
28      },
29      Err(e) => panic!("encountered IO error: {e}")
30    }
31  }
32 }
```

A função main é a função principal que inicia o servidor. Ela define a altura e largura do mapa, a duração da partida e cria um TcpListener que escuta na porta 3012. Em um loop infinito, ela aceita conexões de clientes e cria uma nova *thread* para lidar com cada conexão, chamando a função handle_connection. Além disso, é executada uma *thread* separada que envia continuamente o estado do jogo para todos os jogadores conectados.

4.5 Criar Cliente

Inicialmente, para testar o servidor, foi criada uma página da web contendo um botão. Sempre que um usuário pressionava o botão, a cor de fundo da página era alterada para a próxima cor em um conjunto de cores predefinidas. Todos os usuários que estavam na página visualizavam simultaneamente a nova cor de fundo. Essa abordagem foi utilizada inicialmente para testar a capacidade do servidor em receber informações de diferentes jogadores em *threads* separadas e transmitir suas ações para os demais jogadores.

Os resultados obtidos com esse servidor inicial serão descritos em mais detalhes no Capítulo 5 - Conclusão e Trabalhos Futuros, onde serão analisadas as funcionalidades do servidor, sua capacidade de lidar com múltiplas conexões e a sincronização das ações dos jogadores em tempo real.

Com o servidor devidamente construído e funcional, tornou-se necessário desenvolver um cliente mais robusto capaz de enviar e receber requisições de forma eficiente. Para atender a esse requisito, optou-se por criar um jogo, pois essa solução se adequava melhor à realidade e ao escopo do projeto. O objetivo final não se limitava apenas à construção de um cliente, mas sim à implementação de um jogo *multiplayer* em JavaScript, que se comunicasse em tempo real com o servidor e com os demais jogadores.

Dessa forma, o desenvolvimento do cliente focou em proporcionar uma experiência interativa aos jogadores, permitindo a interação simultânea e em tempo real entre os participantes. O jogo criado serviu como uma plataforma para testar as funcionalidades do servidor, bem como a comunicação eficiente entre os diferentes clientes conectados.

Para desenvolver o cliente em JavaScript, foi necessário adquirir um conhecimento mais aprofundado da linguagem. Durante o processo de desenvolvimento, a documentação oficial da linguagem JavaScript foi uma fonte essencial de referência, assim como a documentação de Rust para o servidor. A escolha do JavaScript se deu, em parte, pela facilidade de uso, pois não requer uma

instalação formal para compilação - basta um editor de texto e um navegador para executar o código. Além disso, o JavaScript possui recursos nativos para manipulação de elementos gráficos, o que facilita a implementação de um jogo.

Ao optar pelo JavaScript, foi possível aproveitar as funcionalidades gráficas oferecidas pela linguagem sem a necessidade de instalar bibliotecas adicionais para lidar com a parte visual do jogo. Caso fosse escolhida uma linguagem como C para o desenvolvimento do cliente, seria necessário instalar bibliotecas gráficas adicionais para trabalhar com elementos visuais. Alternativamente, seria possível utilizar o CMD do Windows, mas isso resultaria em uma interface baseada em caracteres, o que não seria tão agradável visualmente para os usuários. Portanto, o uso do JavaScript foi uma escolha adequada para desenvolver o cliente do jogo de forma mais prática e com uma experiência visual agradável.

No jogo desenvolvido para este projeto, a progressão segue o seguinte fluxo: ao iniciar, o jogador é solicitado a escolher um nome e uma cor para seu personagem. Após fazer essas escolhas, o jogador pode pressionar o botão "Jogar" para começar o jogo, como ilustrado na Figura 4.2.

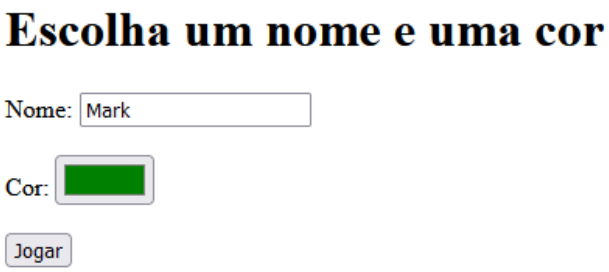

Figura 4.2: Criação de jogador

A partir desse momento, o jogador tem controle sobre seu personagem e pode movê-lo utilizando as setas do teclado, como mostrado na Figura 4.3. Conforme o jogador se move pelo mapa, o quadrado em que o personagem se encontra será automaticamente pintado com a cor escolhida, concedendo um ponto ao jogador, como exemplificado na Figura 4.4.

O objetivo do jogo é acumular o maior número de pontos possível em

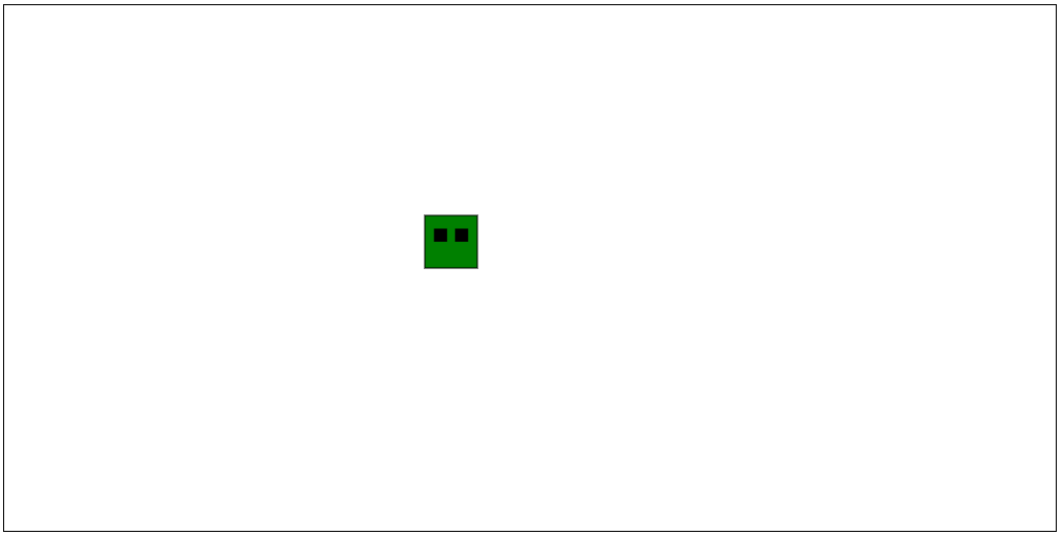

Figura 4.3: Inicio do Jogo

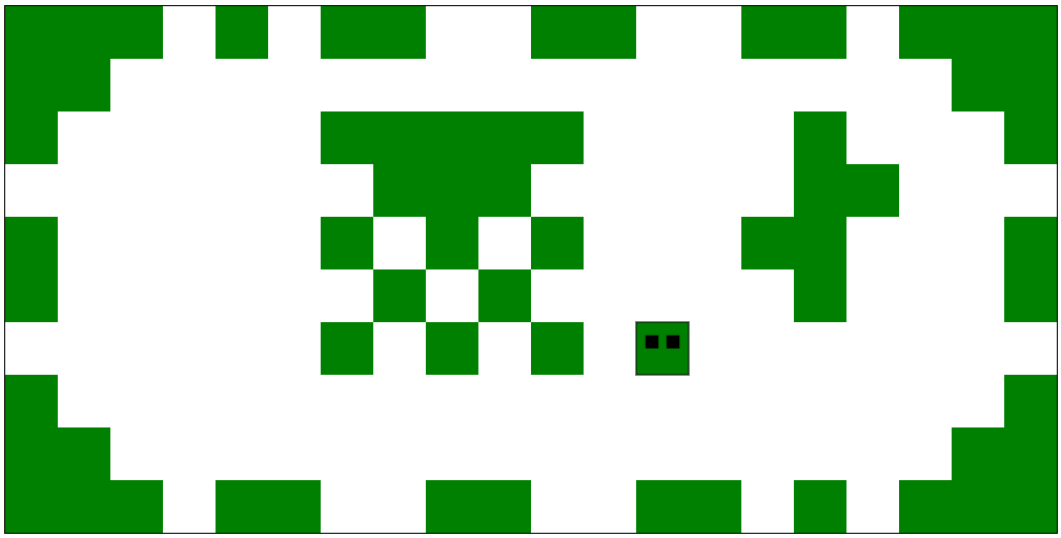

Figura 4.4: Cenário sendo pintado pelo jogador

um limite de tempo. Após 3 minutos de jogo, o jogador que tiver acumulado a maior pontuação será declarado vencedor.

Essa progressão do jogo permite aos jogadores explorarem o mapa, interagirem com outros jogadores e competirem pela conquista de territórios e pela maior pontuação. A mecânica simples, porém desafiadora, estimula a estratégia e a cooperação entre os jogadores para alcançarem melhores resultados.

A comunicação entre o cliente e o servidor no jogo foi estabelecida utilizando o protocolo WebSocket. O WebSocket é um protocolo de comunicação

baseado em TCP (Transmission Control Protocol) e foi projetado para ser utilizado em navegadores web.

O TCP é um protocolo de comunicação orientado à conexão que permite a troca de mensagens confiáveis entre dispositivos em uma rede. Ele estabelece uma conexão bidirecional e confiável entre o cliente e o servidor, utilizando um processo chamado Three-way Handshake, como ilustrado na Figura 4.5. Esse processo envolve três etapas de comunicação entre o cliente e o servidor para sincronizar e estabelecer a conexão de forma segura.

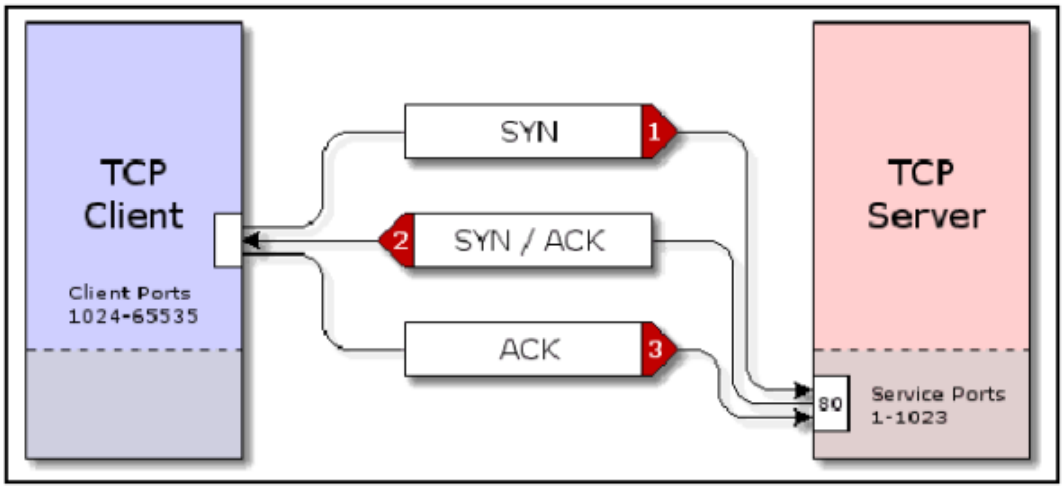

Figura 4.5: Demonstração do Three-way Handshake

Uma vez estabelecida a conexão TCP, o WebSocket utiliza essa conexão para facilitar a comunicação bidirecional em tempo real entre o cliente e o servidor. Isso permite que o servidor envie mensagens para o cliente e vice-versa, sem a necessidade de estabelecer uma nova conexão a cada interação.

No caso do jogo desenvolvido, o servidor WebSocket envia continuamente mensagens JSON contendo o estado atual do jogo para todos os jogadores conectados. Essas mensagens são recebidas pelos clientes e utilizadas para atualizar a interface do jogo no navegador do jogador.

A utilização do protocolo WebSocket proporciona uma comunicação eficiente e em tempo real entre o cliente e o servidor, permitindo uma experiência de jogo *multiplayer* fluida e sincronizada.

O protocolo WebSocket foi selecionado como a melhor opção para viabilizar a comunicação entre o servidor e o cliente devido às suas características de baixa complexidade e alta compatibilidade com diferentes linguagens, incluindo Rust e JavaScript. Essa escolha foi respaldada pelo fato de o cliente do projeto ser desenvolvido em JavaScript. O protocolo WebSocket oferece uma solução eficiente e adequada para a troca contínua de mensagens em tempo real, sendo especialmente adequado para aplicativos como jogos *multiplayer*. Sua compatibilidade nativa com os navegadores modernos elimina a necessidade de instalar bibliotecas adicionais, facilitando a implementação do cliente e garantindo uma ampla compatibilidade com os navegadores utilizados pelos usuários.

4.6 Criar ferramenta de teste

A criação da ferramenta de teste foi uma parte relativamente simples do desenvolvimento, uma vez que as etapas anteriores forneceram experiência suficiente com servidores e clientes. Não exigiu um esforço adicional significativo de pesquisa, uma vez que os conceitos e técnicas já haviam sido explorados. Além disso, a familiaridade adquirida com a documentação da linguagem Rust foi essencial para compreender suas particularidades.

O objetivo principal desta ferramenta é avaliar a robustez do servidor e possibilitar testes de desempenho, incluindo a medição da quantidade de requisições recebidas por segundo. Para isso, foi necessário criar um programa capaz de simular vários jogadores usando entradas simples.

Ao desenvolver essa ferramenta, dois objetivos principais foram definidos:

- Testar o comportamento do servidor com um número elevado de jogadores simultâneos de forma rápida e eficiente.
- Garantir que os jogadores simulados se comportassem de maneira semelhante a um jogador real, proporcionando uma avaliação mais precisa do desempenho do servidor.

A criação dessa ferramenta de teste foi essencial para verificar a capacidade do servidor de lidar com um grande volume de conexões e garantir sua estabilidade e escalabilidade em um ambiente realista.

Para a primeira etapa, foi desenvolvido um cliente em Rust que recebe como entrada o número de jogadores a serem simulados e a porta em que eles devem atuar. Com base nessas informações, o programa cria múltiplos jogadores simulados que se conectam ao servidor. Essa abordagem permitiu simular a presença de vários jogadores simultâneos durante os testes de desempenho do servidor.

O método de input utilizado pode ser visualizado na Figura 4.6, onde os parâmetros necessários são fornecidos ao programa no momento de sua execução. Isso possibilitou controlar facilmente o número de jogadores simulados e a porta em que eles seriam conectados, proporcionando flexibilidade e facilidade na realização dos testes de carga no servidor.

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.05s
     Running `target\debug\bots.exe`
Quantos jogadores voce quer simular?
500
Qual a porta desejada?
3012
```

Figura 4.6: Funcionamento da ferramenta de teste

Na segunda etapa, não era necessário criar uma inteligência artificial complexa, pois o objetivo era ter vários jogadores simulados com comportamentos mínimos. Levando isso em consideração, decidiu-se que a IA desses bots seria baseada em números aleatórios. Dependendo do número sorteado, o bot realizaria um movimento específico e pintaria o quadrado em que se encontrava ao final do movimento.

Essa abordagem simples permitiu que os bots tivessem comportamentos variados durante o jogo, sem exigir um algoritmo de tomada de decisão elaborado. Ao depender de números aleatórios, cada bot teria uma chance

igual de realizar diferentes ações, proporcionando uma experiência diversificada durante os testes.

Essa solução foi adequada para a etapa de teste, pois o objetivo era verificar a capacidade do servidor em lidar com um grande número de jogadores simultâneos, independentemente do comportamento individual de cada bot.

5
Conclusão e Trabalhos Futuros

Neste capítulo, apresentaremos as conclusões obtidas a partir do desenvolvimento do projeto proposto, que teve como objetivo a construção de um servidor *multithread* em Rust para jogos, juntamente com um cliente em JavaScript. Também abordaremos as contribuições e os aprendizados adquiridos ao longo do processo.

5.1
Resumo das realizações

Durante o projeto, alcançamos diversos resultados. Inicialmente, construímos um servidor *multithread* em Rust, explorando as facilidades oferecidas pela linguagem para lidar com concorrência e paralelismo. Utilizamos recursos como canais de comunicação entre *threads*, mutexes e semáforos para garantir a sincronização adequada e evitar condições de corrida. O servidor demonstrou ser robusto e eficiente, capaz de lidar com um grande número de jogadores simultâneos.

Em seguida, desenvolvemos um cliente em JavaScript que funcionou como um jogo, estabelecendo uma comunicação em tempo real com o servidor por meio de WebSockets. Implementamos as funcionalidades do jogo, permitindo que os jogadores interagissem entre si no ambiente virtual. O cliente em JavaScript possibilitou testar e avaliar a eficiência do servidor em lidar com múltiplas conexões simultâneas.

Além disso, criamos mecanismos de teste de desempenho para avaliar o funcionamento do servidor. Conduzimos 15 testes com diferentes números de jogadores simultâneos, variando de 10 a 50, e duração da partida entre 1 e 3 minutos. Esses testes nos permitiram avaliar a quantidade de requisições enviadas por cada jogador ao servidor durante uma partida. Na figura 5.1, apresentamos os resultados obtidos a partir desses testes.

Os testes foram conduzidos em uma máquina que executava simultaneamente o servidor e a ferramenta de testes. A máquina utilizada possuía um processador AMD Ryzen 5 3600, 16GB de memória RAM DDR4 3200MHz, placa de vídeo RTX 3060 Ti e sistema operacional Windows 10 64 bits. No que diz respeito ao código, após a ferramenta de teste enviar um comando específico para um jogador, o contador daquele jogador era incrementado, e ao final da partida, o valor total no contador de cada jogador era exibido no terminal. Dessa forma, garantimos que todas as requisições individuais de cada jogador de teste fossem contabilizadas corretamente.

TESTE 1	
Elemento	Quantidade
Tempo	60 seg
Numero de bots	10
Media de Requisições	1192
Requisições por seg	19.9

TESTE 2	
Elemento	Quantidade
Tempo	60 seg
Numero de bots	20
Media de Requisições	1172
Requisições por seg	19.5

TESTE 3	
Elemento	Quantidade
Tempo	60 seg
Numero de bots	30
Media de Requisições	1043
Requisições por seg	17.4

TESTE 4	
Elemento	Quantidade
Tempo	60 seg
Numero de bots	40
Media de Requisições	1091
Requisições por seg	18.2

TESTE 5	
Elemento	Quantidade
Tempo	60 seg
Numero de bots	50
Media de Requisições	1031
Requisições por seg	17.2

TESTE 6	
Elemento	Quantidade
Tempo	120 seg
Numero de bots	10
Media de Requisições	2363
Requisições por seg	19.7

TESTE 7	
Elemento	Quantidade
Tempo	120 seg
Numero de bots	20
Media de Requisições	2390
Requisições por seg	19.9

TESTE 8	
Elemento	Quantidade
Tempo	120 seg
Numero de bots	30
Media de Requisições	2200
Requisições por seg	18.3

TESTE 9	
Elemento	Quantidade
Tempo	120 seg
Numero de bots	40
Media de Requisições	2179
Requisições por seg	18.2

TESTE 10	
Elemento	Quantidade
Tempo	120 seg
Numero de bots	50
Media de Requisições	2211
Requisições por seg	18.4

TESTE 11	
Elemento	Quantidade
Tempo	180 seg
Numero de bots	10
Media de Requisições	3541
Requisições por seg	19.7

TESTE 12	
Elemento	Quantidade
Tempo	180 seg
Numero de bots	20
Media de Requisições	3648
Requisições por seg	20.3

TESTE 13	
Elemento	Quantidade
Tempo	180 seg
Numero de bots	30
Media de Requisições	3537
Requisições por seg	19.7

TESTE 14	
Elemento	Quantidade
Tempo	180 seg
Numero de bots	40
Media de Requisições	3329
Requisições por seg	18.5

TESTE 15	
Elemento	Quantidade
Tempo	180 seg
Numero de bots	50
Media de Requisições	3355
Requisições por seg	18.6

Media de requisicoes por segundo em todos os testes: 18.9

Figura 5.1: Testes realizados

O servidor demonstrou a capacidade de receber e responder aproximadamente 18 requisições por segundo de cada jogador. Esses testes permitiram avaliar a eficiência e escalabilidade do servidor, permitindo identificar possíveis áreas de melhoria e otimização.

É importante ressaltar que os testes realizados têm mais como objetivo

demonstrar o uso da ferramenta de teste do que apresentar uma avaliação completa da qualidade do servidor. Com mais tempo, seria possível utilizar a ferramenta para realizar testes mais abrangentes e aprofundados, oferecendo uma análise mais precisa e elucidativa sobre o desempenho e a capacidade de resposta do servidor em diferentes cenários e condições de carga.

5.2 Contribuições e aprendizados

O projeto apresentou contribuições tanto do ponto de vista prático quanto do ponto de vista acadêmico.

Em termos práticos, desenvolvemos um servidor *multithread* em Rust que se mostrou eficiente e capaz de lidar com um grande número de jogadores simultâneos. Isso abre possibilidades para a criação de jogos multijogador escaláveis, proporcionando uma experiência interativa e envolvente para os usuários. A escolha da linguagem Rust se mostrou acertada, uma vez que suas características de segurança, concorrência e abstração sem *overhead* facilitaram o desenvolvimento de um servidor confiável e de alto desempenho.

Do ponto de vista acadêmico, aprendemos sobre os desafios e as soluções envolvidos na construção de servidores *multithread*. Lidar com concorrência e paralelismo requer uma atenção especial para evitar condições de corrida e garantir a sincronização adequada entre as *threads*. Durante o desenvolvimento, experimentamos uma curva de aprendizado relativamente desafiadora em relação à linguagem Rust, dada sua abordagem única e rigorosa em relação à segurança e ao paralelismo.

Aprofundar nossos conhecimentos sobre a linguagem Rust nos permitiu explorar suas características exclusivas. Através do uso de recursos como canais de comunicação, mutexes e semáforos, conseguimos criar um servidor confiável e escalável, capaz de lidar com a concorrência de maneira eficiente.

Além disso, a interação com o cliente em JavaScript nos proporcionou uma compreensão mais abrangente sobre a comunicação em tempo real e a

interação entre o servidor e os jogadores. Essa interação dinâmica entre o servidor e o cliente exigiu uma compreensão aprofundada dos protocolos de comunicação e da implementação dos recursos necessários para manter uma experiência de jogo fluida e responsiva.

Embora a curva de aprendizado em Rust possa ser desafiadora devido às suas particularidades e abordagem de segurança rigorosa, as recompensas são significativas. A linguagem Rust oferece uma base sólida para a construção de servidores eficientes e seguros, proporcionando aos desenvolvedores ferramentas poderosas para enfrentar os desafios do paralelismo e da concorrência.

5.3 Limitações do projeto

Apesar das conquistas obtidas ao longo do projeto, também encontramos algumas limitações que podem ser abordadas em trabalhos futuros para aprimorar ainda mais a solução desenvolvida.

Uma das limitações identificadas está relacionada à IA dos bots utilizados nos testes de desempenho. Embora tenhamos implementado uma abordagem simples baseada em números aleatórios para simular comportamentos mínimos de jogadores, seria interessante explorar técnicas mais avançadas de inteligência artificial, permitindo aos bots tomar decisões mais inteligentes e realistas. Isso poderia adicionar um novo nível de desafio e interação aos jogos.

Outro aspecto relevante a ser considerado é a escalabilidade do servidor em ambientes de alta demanda, especialmente quando se trata de jogos com tabuleiros de maior escala. Embora tenhamos realizado testes de desempenho com variações no tempo de partida e no número de jogadores, é fundamental expandir nossos testes para incluir escalas de tabuleiro maiores. Isso permitirá avaliar a capacidade do servidor de lidar eficientemente com um número crescente de jogadores, garantindo uma experiência de jogo fluida e sem degradação de desempenho. Além disso, explorar estratégias de otimização e distribuição de carga, como o balanceamento de carga e o dimensionamento

automático de recursos, será essencial para aprimorar ainda mais a capacidade de resposta do servidor em situações de pico e garantir uma jogabilidade estável e satisfatória para todos os jogadores envolvidos.

Além disso, é válido mencionar que o jogo desenvolvido para o projeto possui uma versão minimalista e top-down em comparação com o jogo Splatoon do Nintendo Switch. Uma melhoria adicional poderia envolver a adição de recursos e elementos de jogabilidade mais avançados, tornando o jogo mais envolvente e atrativo para os jogadores.

Outra área que pode ser explorada é a implementação de recursos de segurança adicionais no servidor. Embora tenhamos abordado questões de concorrência e sincronização, é importante considerar medidas de proteção contra ataques maliciosos, como autenticação de usuários, criptografia de dados e prevenção de ataques de negação de serviço (DDoS). Essas medidas garantiriam a integridade e a segurança da experiência de jogo dos usuários.

5.4 Conclusão geral

Em suma, o projeto apresentou resultados satisfatórios, cumprindo os objetivos propostos de construir um servidor *multithread* em Rust, um cliente em JavaScript e mecanismos de teste de desempenho. Através do desenvolvimento dessas partes, pudemos explorar as facilidades oferecidas pela linguagem Rust no contexto de jogos, bem como a interação entre o servidor e o cliente em tempo real.

As contribuições práticas e acadêmicas deste projeto destacam a viabilidade de utilizar a linguagem Rust para o desenvolvimento de servidores *multithread* eficientes e seguros. A experiência adquirida ao lidar com desafios de concorrência, comunicação em tempo real e otimização de desempenho representa um conhecimento valioso para futuros projetos nessa área.

Com base nas limitações e possíveis melhorias identificadas, é encorajador continuar a explorar e aprimorar a solução desenvolvida, levando em considera-

ção aspectos como IA avançada, escalabilidade, segurança e aprimoramentos no jogo em si. Com isso, podemos criar servidores de jogos ainda mais robustos, interativos e atrativos para os jogadores, proporcionando experiências envolventes e de alto desempenho.

Vale lembrar que as versões do servidor, jogo e ferramenta de testes desenvolvidas para este projeto estão disponíveis neste repositório do GitHub[1].

[1] https://github.com/MarkRibeiro/Uso_da_linguagem_Rust_para_paralelismo_em_servidores_de_jogos

6
Referências bibliográficas

1 ABDELKHALEK, A.; BILAS, A. Parallelization and performance of interactive multiplayer game servers. In: **18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.** [S.l.: s.n.], 2004. p. 72–.

2 KLABNIK, S.; NICHOLS, C. **The Rust Programming Language**. [S.l.]: The Starch Press, 2019.