Dueling Protocol: Um Servidor de Jogo Concorrente

Lucas do Carmo Santos

13 de setembro de 2025

Resumo

Este trabalho apresenta o desenvolvimento de um servidor de jogo multiplayer para um jogo de cartas 1v1, denominado D̈ueling Protocol;. O servidor foi implementado em Java, utilizando comunicação via sockets TCP para a lógica do jogo e UDP para medição de latência. A arquitetura adota o padrão Facade para organizar a lógica de negócio e utiliza estruturas de dados *thread-safe* para garantir a consistência em ambiente concorrente. O projeto foi validado por meio de uma suíte de testes automatizados com Docker, demonstrando a capacidade de gerenciar múltiplas conexões simultâneas e a robustez contra falhas e condições de corrida.

1 Introdução

O mercado de jogos online tem experimentado um crescimento exponencial, com milhões de jogadores conectados simultaneamente em diversos tipos de jogos. Neste contexto, a arquitetura do servidor desempenha um papel crucial para garantir uma experiência de jogo fluida, justa e segura. Este trabalho propõe a implementação de um servidor de jogo multiplayer para um jogo de cartas 1v1, onde dois jogadores competem em duelos táticos.

O principal desafio do projeto foi desenvolver um backend robusto, escalável e concorrente, capaz de gerenciar múltiplas conexões simultâneas, lidar com ações concorrentes dos jogadores e garantir a integridade dos dados. Para isso, foi utilizada a linguagem Java, com comunicação via sockets TCP para a lógica crítica do jogo e sockets UDP para medição de latência.

A solução proposta, denominada D̈ueling Protocol;, adota uma arquitetura cliente-servidor, onde o servidor atende múltiplos clientes simultaneamente através de threads independentes. A lógica de negócio foi centralizada utilizando o padrão Facade, promovendo baixo acoplamento e facilitando a manutenção. Além disso, foram implementados mecanismos para persistência de dados dos jogadores em formato JSON e uma suíte de testes automatizados com Docker para validar a robustez e o desempenho da solução.

Este relatório está organizado da seguinte forma: a seção 2 apresenta a fundamentação teórica necessária para compreensão do trabalho; a seção 3 detalha a metodologia utilizada no desenvolvimento; a seção 4 apresenta os resultados obtidos com a execução dos testes; a seção 5 discute as contribuições e conclusões do trabalho; e, finalmente, a seção 6 apresenta as referências bibliográficas utilizadas.

2 Fundamentação Teórica

2.1 Arquitetura Cliente-Servidor

A arquitetura cliente-servidor é um modelo de computação distribuída onde os recursos e serviços são divididos entre provedores (servidores) e requerentes (clientes) [?]. No contexto de jogos online, o servidor é responsável por gerenciar o estado do jogo, processar as ações dos jogadores e manter a sincronização entre os clientes. Esta arquitetura oferece vantagens como centralização do controle, escalabilidade e segurança.

2.2 Comunicação de Rede com Sockets

Sockets são interfaces de programação que permitem a comunicação entre processos em rede. Eles fornecem um ponto de comunicação bidirecional entre aplicações em execução em hosts diferentes [?]. Em Java, a API de sockets permite criar clientes e servidores que podem se comunicar através de protocolos como TCP e UDP.

2.3 Protocolos TCP e UDP

TCP (Transmission Control Protocol) e UDP (User Datagram Protocol) são protocolos de transporte da camada de rede. TCP é um protocolo orientado à conexão, confiável e com controle de fluxo, garantindo a entrega ordenada dos dados [?]. UDP, por outro lado, é um protocolo não orientado à conexão, mais rápido mas sem garantias de entrega [?]. No D̈ueling Protocol, TCP foi utilizado para a lógica crítica do jogo, onde a confiabilidade é essencial, e UDP foi utilizado para medição de latência, onde a velocidade é mais importante.

2.4 Multithreading em Java

Multithreading é a capacidade de um programa executar múltiplas threads (linhas de execução) concorrentemente. Em Java, threads são criadas através da classe `Thread` ou implementando a interface `Runnable` [?]. No contexto de servidores de jogo, multithreading é essencial para atender múltiplos clientes simultaneamente, onde cada cliente é gerenciado por uma thread independente. O modelo üma thread por clienteẗoi adotado no D̈ueling Protocolp̈ara garantir isolamento e responsividade.

2.5 Serialização de Dados com JSON

JSON (JavaScript Object Notation) é um formato leve e legível para troca de dados. É amplamente utilizado em aplicações web e mobile devido à sua simplicidade e facilidade de parsing [?]. No D̈ueling Protocol, JSON foi utilizado para persistir os dados dos jogadores, como nome, pontos de vida e cartas, devido à sua legibilidade e facilidade de integração com Java.

2.6 Virtualização com Contêineres (Docker)

Docker é uma plataforma de virtualização de contêineres que permite empacotar aplicações e suas dependências em unidades leves e portáteis [?]. No desenvolvimento do

D̈ueling Protocol;̈ Docker foi utilizado para criar ambientes isolados para o servidor e os clientes de teste, garantindo reprodutibilidade e facilitando a automação dos testes.

3 Metodologia

3.1 Arquitetura

A arquitetura do D̈ueling Protocol f̈oi projetada com foco em baixo acoplamento, escalabilidade e robustez. A Figura **??** apresenta um diagrama de componentes que ilustra a organização do sistema.

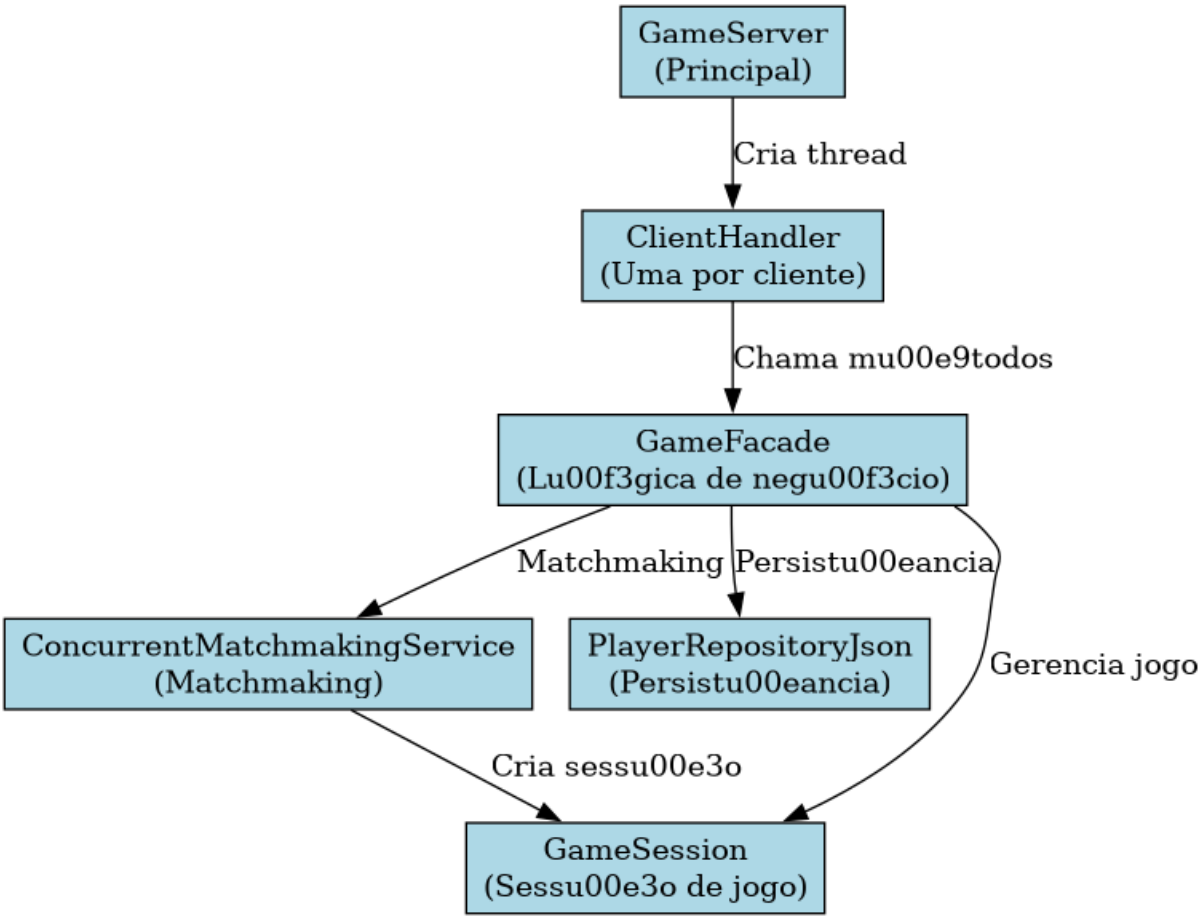

Figura 1: Diagrama de Componentes da Arquitetura do Dueling Protocol

O componente central da arquitetura é o `GameServer`, responsável por iniciar o servidor, escutar por conexões TCP e UDP, e criar uma thread `ClientHandler` para cada cliente conectado. O `ClientHandler` é responsável por gerenciar a comunicação com o cliente, interpretar os comandos recebidos e encaminhar as requisições para o `GameFacade`. O `GameFacade` atua como uma fachada, centralizando a lógica de negócio e coordenando as interações entre os serviços de matchmaking, persistência de dados e gerenciamento de sessões de jogo. Esta abordagem promove baixo acoplamento entre os componentes e facilita a manutenção e evolução do sistema.

3.2 Comunicação e API Remota

A comunicação entre cliente e servidor é baseada em mensagens de texto simples, trocadas através de sockets TCP. Cada mensagem é composta por um comando e, opcionalmente, parâmetros. A Tabela **??** apresenta os principais comandos da API.

Tabela 1: Comandos da API Remota

Comando	**Descrição**	**Exemplo**
JOIN	Conecta um jogador ao servidor	JOIN nome
LEAVE	Desconecta um jogador do servidor	LEAVE
DRAW	Solicita carta do deck	DRAW
PLAY	Joga uma carta	PLAY id
DECK	Lista cartas do deck	DECK
HAND	Lista cartas na mão	HAND
HEALTH	Exibe pontos de vida	HEALTH
CHAT	Envia mensagem de chat	CHAT mensagem
PING	Solicita medição de latência	PING

A Figura **??** apresenta um diagrama de sequência que ilustra o fluxo de comunicação para o processo de matchmaking.

3.3 Encapsulamento e Concorrência

Para garantir a integridade dos dados em um ambiente concorrente, foram utilizadas estruturas de dados *thread-safe* como `ConcurrentHashMap` e `ConcurrentLinkedQueue`. O `ClientHandler` é responsável por interpretar os comandos recebidos dos clientes e encaminhar as requisições para o `GameFacade`. Comandos inválidos são tratados com o envio de uma mensagem de erro ao cliente. O `GameFacade` centraliza a lógica de negócio, garantindo que as operações sejam realizadas de forma segura e consistente.

3.4 Latência, Partidas e Pacotes

O servidor implementa um mecanismo de medição de latência através de um servidor UDP dedicado (`PingServer`). Os clientes enviam pacotes UDP para o servidor, que responde com o mesmo conteúdo, permitindo que o cliente calcule o tempo de ida e volta. Para a criação de partidas, o sistema utiliza um serviço de matchmaking (`ConcurrentMatchmakingService`) que emparelha jogadores com base na disponibilidade. Foi identificada e corrigida uma condição de corrida no processo de matchmaking, garantindo que partidas sejam criadas de forma justa e consistente.

3.5 Testes e Emulação

Uma suíte de testes automatizados foi desenvolvida utilizando Docker e Docker Compose. O `docker-compose.yml` define serviços para o servidor e múltiplos clientes de teste. Scripts shell como `run_all_tests.sh` automatizam a execução dos testes de cenário, incluindo D̈esconexão na Fila¨; R̈ace Condition¨; M̈alicious Botë um teste de estresse com 10 clientes simultâneos. Os resultados dos testes são registrados em logs para análise posterior.

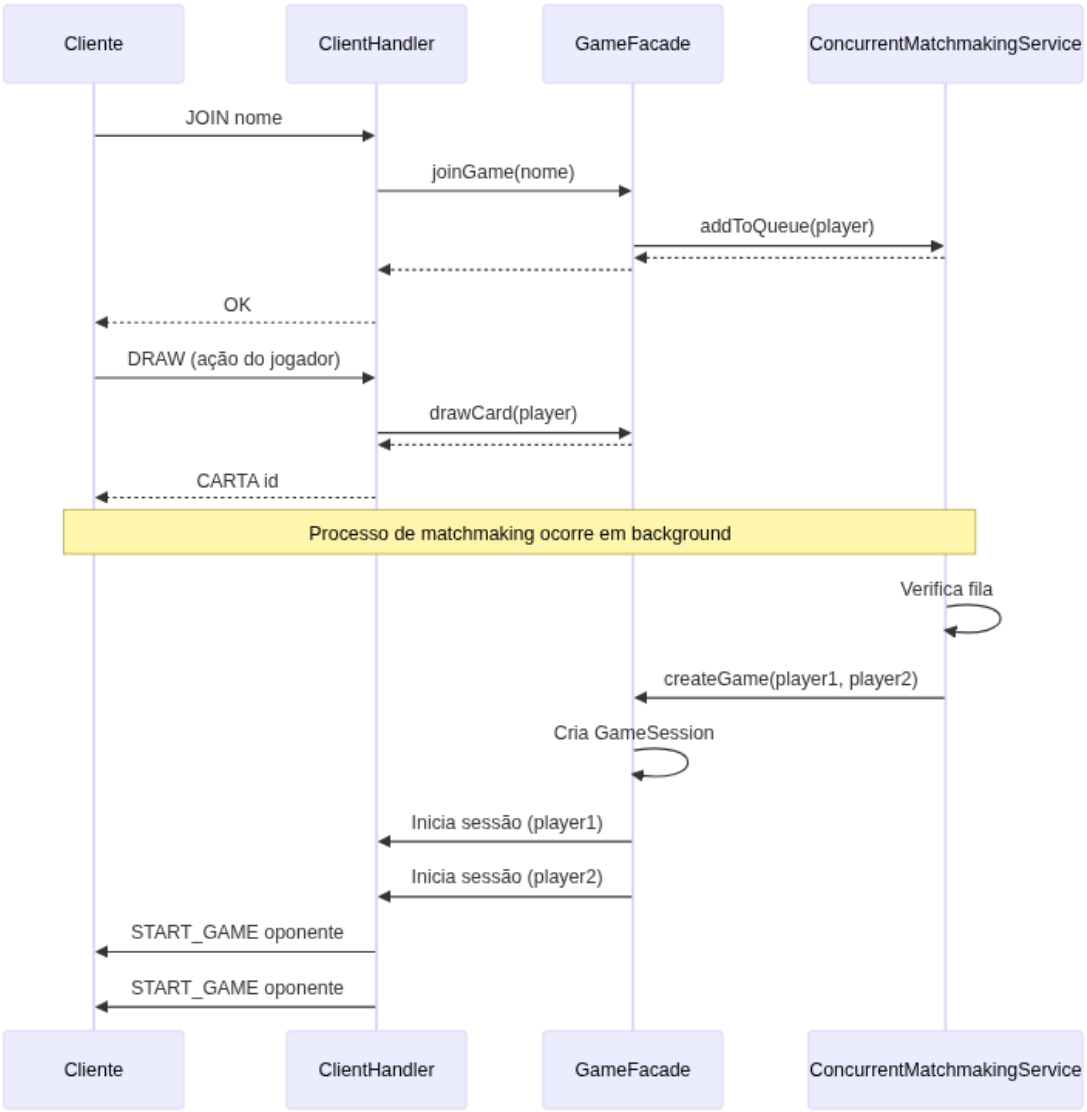

Figura 2: Diagrama de Sequência do Processo de Matchmaking

4 Resultados

Os testes realizados demonstraram a robustez e a eficácia da solução proposta. A Tabela ?? apresenta um resumo dos testes executados e seus resultados.

Os testes de estresse demonstraram a capacidade do servidor de gerenciar múltiplas conexões simultâneas sem falhas. O teste D̈esconexão na Filaf̈oi particularmente importante, pois permitiu identificar e corrigir uma condição de corrida no sistema de matchmaking, validando a robustez da solução final.

5 Conclusão

Este trabalho apresentou o desenvolvimento de um servidor de jogo multiplayer para um jogo de cartas 1v1, denominado D̈ueling Protocol.̈ A solução foi implementada em Java, utilizando comunicação via sockets TCP e UDP, e adotando uma arquitetura cliente-servidor com multithreading. A centralização da lógica de negócio através do padrão

Tabela 2: Resultados dos Testes

Teste	**Objetivo**	**Resultado**
Desconexão na Fila	Verificar se o servidor lida com desconexão antes da partida	**Sucesso**: Part
Jogada Simultânea	Testar o comportamento com ações concorrentes	**Sucesso**: Serv
Malicious Bot	Validar a robustez contra comandos malformados	**Sucesso**: Serv
Teste de Estresse	Avaliar desempenho com 10 clientes simultâneos	**Sucesso**: Serv

Facade e o uso de estruturas de dados *thread-safe* garantiram a consistência e a robustez do sistema em um ambiente concorrente.

A suíte de testes automatizados com Docker provou ser uma ferramenta valiosa para validar a solução, permitindo a identificação e correção de uma condição de corrida crítica. Os resultados obtidos demonstram que o servidor é capaz de gerenciar múltiplas conexões simultâneas, lidar com ações concorrentes e manter a integridade dos dados.

Como trabalhos futuros, sugere-se a implementação de funcionalidades adicionais ao jogo, como um sistema de Ënergiaëm tempo real, expansão do sistema de melhorias com novas habilidades, criação de efeitos de cartas mais complexos e estratégicos, e implementação de um sistema de montagem de decks. Além disso, a arquitetura proposta pode ser estendida para suportar outros tipos de jogos multiplayer.