Relatório Técnico: Sistema Distribuído para Jogo de Cartas Multiplayer

Lucas Damasceno

[1] Universidade Estadual de Feira de Santana (UEFS)
Departamento de Tecnologia
Disciplina: Concorrência e Conectividade (TEC502)
Professor: José Amancio Macedo Santos
Feira de Santana – BA, Brasil
Outubro de 2024

lucas.damasceno.dev@gmail.com

Resumo. *Este trabalho apresenta a migração de um jogo de cartas multiplayer de arquitetura centralizada para distribuída. O sistema implementa múltiplos servidores colaborativos que gerenciam partidas cross-server, trocas atômicas de cartas usando protocolo Two-Phase Commit (2PC) e gerenciamento distribuído de estoque. A comunicação servidor-servidor utiliza API REST, enquanto a comunicação cliente-servidor emprega WebSocket com Redis Pub/-Sub. A solução demonstra escalabilidade horizontal, tolerância a falhas com Redis Sentinel, e consistência de dados através de transações ACID do PostgreSQL. Testes automatizados validam a corretude do sistema em cenários de concorrência distribuída e falhas de componentes.*

Sumário

1. Introdução

1.1. Contextualização

Sistemas distribuídos são essenciais para aplicações modernas que demandam escalabilidade e alta disponibilidade. Jogos multiplayer online apresentam desafios únicos: necessitam de baixa latência, gerenciamento de estado compartilhado, e garantias de consistência em operações críticas. A migração de arquiteturas centralizadas para distribuídas permite atender crescimento de usuários eliminando pontos únicos de falha.

1.2. Descrição do Problema

O protótipo inicial do sistema utilizava arquitetura centralizada com servidor único gerenciando lógica, estado e comunicação. Essa abordagem apresentava limitações de escalabilidade e disponibilidade. O projeto visa reengenharia para arquitetura distribuída com múltiplos servidores colaborativos, suportando: (1) escalabilidade horizontal para maior número de jogadores, (2) eliminação de ponto único de falha, (3) consistência de estado em ambiente distribuído, e (4) nova funcionalidade de troca de cartas entre jogadores.

1.3. Objetivos

1.3.1. Objetivo Geral

Desenvolver sistema distribuído de jogo de cartas multiplayer com múltiplos servidores colaborativos, garantindo escalabilidade, tolerância a falhas e consistência de dados.

1.3.2. Objetivos Específicos

- Implementar comunicação servidor-servidor via API REST para coordenação distribuída.
- Desenvolver protocolo cliente-servidor baseado em publisher-subscriber com WebSocket e Redis Pub/Sub.
- Criar mecanismo de gerenciamento distribuído de estoque de cartas sem solução centralizada.
- Implementar protocolo Two-Phase Commit para trocas atômicas de cartas entre servidores.
- Desenvolver sistema de pareamento cross-server para partidas entre jogadores em servidores distintos.
- Estabelecer estratégias de tolerância a falhas e alta disponibilidade com Redis Sentinel.
- Validar sistema através de testes automatizados em cenários de concorrência e falhas.

1.4. Organização do Relatório

A Seção 2 apresenta fundamentação teórica sobre sistemas distribuídos e protocolos de consistência. A Seção 3 detalha metodologia de desenvolvimento e decisões arquiteturais. A Seção 4 apresenta resultados de implementação e testes. A Seção 5 conclui com análise crítica e trabalhos futuros.

2. Fundamentação Teórica

2.1. Sistemas Distribuídos

Sistemas distribuídos são conjuntos de componentes autônomos conectados por rede que coordenam ações através de troca de mensagens [1]. Arquiteturas de microserviços decompõem aplicações em serviços independentes com deploy autônomo e escalabilidade granular [2]. Comunicação REST fornece interface síncrona entre servidores, enquanto WebSocket e Redis Pub/Sub implementam comunicação assíncrona de baixa latência [3].

2.2. Coordenação Distribuída

O protocolo Two-Phase Commit (2PC) garante atomicidade em transações distribuídas [4]: fase PREPARE valida e bloqueia recursos, fase COMMIT/ABORT efetiva ou reverte operação. Redisson implementa locks distribuídos sobre Redis para exclusão mútua [5]. Redis Sentinel provê alta disponibilidade através de monitoramento, failover automático e replicação [3]. PostgreSQL garante propriedades ACID com transações completas ou rollback [6].

3. Metodologia

3.1. Arquitetura e Stack Tecnológica

A arquitetura implementa microserviços com shared-database: múltiplos servidores stateless (Java 21 + Spring Boot 3.2.0) compartilham PostgreSQL 16 como fonte única de verdade. Redis Sentinel Cluster (master + 2 replicas + 3 sentinels) provê cache, Pub/Sub e locks distribuídos via Redisson 3.25.0. NGINX atua como gateway reverso. Componentes principais: (1) Cliente JavaFX via WebSocket, (2) Gateway NGINX, (3) Cluster de servidores, (4) PostgreSQL para persistência ACID, (5) Redis Sentinel para coordenação. Docker containeriza deployment.

3.2. Protocolos de Comunicação

Servidor-Servidor (REST): API REST com endpoints `/api/matchmaking/find-and-lock-partner` (matchmaking cross-server), `/api/trade/prepare` (2PC fase 1), `/api/trade/commit` (2PC fase 2), e `/api/servers/register` (auto-descoberta).

Cliente-Servidor (WebSocket): Protocolo textual `GAME:{playerId}:{action}:{params}` com ações CHARACTER_SETUP, MATCHMAKING, BUY_CARD, TRADE_PROPOSE/ACCEPT/REJECT. Redis Pub/Sub propaga eventos via canais `{playerId}:events` e `CROSS_SERVER:{playerId}`.

3.3. Soluções Distribuídas Implementadas

Gerenciamento de Estoque: Sem coordenador centralizado. Transações ACID no PostgreSQL com `SELECT FOR UPDATE` bloqueiam linha, decrementam stock e alocam cartas atomicamente. Rollback automático em falhas.

Trocas com 2PC: Fase PREPARE valida recursos e bloqueia cartas com locks Redisson (voto PREPARED/ABORT). Fase COMMIT executa transação local ou reverte. Timeout de 30s força rollback. Garante atomicidade cross-server.

Matchmaking Cross-Server: Tarefa agendada no líder busca parceiros via REST com lock de 10s. Cooldown previne condições de corrida. Jogadores notificados via Pub/Sub.

Tolerância a Falhas: Redis Sentinel com quorum de 2 detecta falhas e promove replica. HikariCP mantém pool PostgreSQL. Leader election via lock Redis (TTL 60s) com re-eleição automática. Transações Spring com rollback coordenado.

3.4. Testes Automatizados

Scripts Bash automatizam testes de cenários distribuídos:

test_cross_server_trade.sh: Conecta dois clientes a servidores diferentes, simula proposta de troca, valida execução atômica e verifica consistência final dos inventários.

test_cross_server_match.sh: Entra dois jogadores em matchmaking em servidores distintos, valida pareamento cross-server e criação de partida.

test_card_purchase.sh: Executa compras concorrentes do mesmo pacote único por múltiplos clientes, valida que apenas um sucede.

Testes validam corretude em concorrência, atomicidade de 2PC, e recuperação de falhas.

4. Resultados

4.1. Implementação da Arquitetura Distribuída

O sistema foi implementado com sucesso em arquitetura de microserviços distribuídos. Configuração padrão executa 2 servidores de jogo (portas 8080 e 8083), 1 PostgreSQL, 1 Redis master com 2 replicas, 3 Redis Sentinels, e 1 gateway NGINX. Todos componentes executam em contêineres Docker isolados. Sistema suporta adição dinâmica de servidores sem downtime através de auto-registro.

4.2. Endpoints REST Implementados

A Tabela 1 lista principais endpoints da API REST para comunicação servidor-servidor.

Tabela 1. Endpoints REST para coordenação distribuída

Método	Endpoint	Função
GET	/api/matchmaking/find-and-lock-partner	Busca jogador disponível com lock temporário
POST	/api/trade/prepare	Valida e prepara troca (2PC Fase 1)
POST	/api/trade/commit	Efetiva ou reverte troca (2PC Fase 2)
POST	/api/servers/register	Registra servidor no cluster
GET	/api/health	Verifica saúde do servidor

4.3. Protocolo WebSocket e Pub/Sub

Sistema utiliza WebSocket para comunicação cliente-servidor com protocolo textual customizado. Redis Pub/Sub propaga eventos cross-server. Biblioteca Redisson fornece cliente Redis com suporte a Sentinel. Escolha justificada por: (1) maturidade e estabilidade, (2) suporte nativo a locks distribuídos e Sentinel, (3) API reativa para operações assíncronas, (4) integração com Spring Boot.

4.4. Validação das Soluções

Estoque: 10 clientes simultâneos comprando mesmo pacote. Apenas 1 sucede, demais recebem erro. 100 execuções sem duplicação.

2PC: Cenário sucesso - troca cross-server executada atomicamente. Cenário rollback - carta inexistente abortada corretamente. Cenário timeout - simulação de 35s aborta em 30s sem inconsistência.

Matchmaking: Jogadores em servidores distintos pareados com sucesso. Lock de 10s previne duplicação. 50 execuções sem condições de corrida.

Failover Redis: Master desligado, Sentinels elegem replica em 5s, reconexão automática. Downtime: 8s. Teste de servidor desligado durante 2PC: timeout aborta, sem corrupção. Leader election: re-eleição em 10s sem duplicação de tarefas.

4.5. Resultados dos Testes Automatizados

A Tabela 2 sumariza resultados dos testes automatizados.

Tabela 2. Resultados dos testes automatizados

Teste	Execuções	Taxa Sucesso
Cross-Server Trade (sucesso)	100	100%
Cross-Server Trade (rollback)	50	100%
Cross-Server Matchmaking	100	100%
Compra Concorrente de Pacote	100	100%
Failover Redis Sentinel	20	100%

Todos testes executados com sucesso sem falsos positivos. Sistema demonstra comportamento correto em cenários normais e adversos.

5. Conclusão

Este trabalho apresentou migração bem-sucedida de arquitetura centralizada para distribuída em sistema de jogo de cartas multiplayer. Implementou-se: múltiplos servidores colaborativos (REST), protocolo cliente-servidor (WebSocket/Pub/Sub), gerenciamento distribuído de estoque (transações ACID), trocas atômicas cross-server (2PC), matchmaking distribuído (cooldown), e alta disponibilidade (Redis Sentinel/leader election). Resultados demonstram escalabilidade horizontal, eliminação de ponto único de falha, e consistência em operações distribuídas. Contribuições incluem implementação completa de 2PC em contexto de jogo, estratégia de estoque sem coordenador centralizado, e suite de testes automatizados. Limitações: bloqueio 2PC em falha do coordenador,

acoplamento via shared-database, ausência de sharding. Trabalhos futuros: Three-Phase Commit (3PC), database-per-service, sharding PostgreSQL, circuit breakers, distributed tracing.

Referências

[1] Tanenbaum, A. S.; Van Steen, M. **Distributed Systems: Principles and Paradigms**. 3rd edition. CreateSpace Independent Publishing Platform, 2017.

[2] Newman, S. **Building Microservices: Designing Fine-Grained Systems**. O'Reilly Media, 2015.

[3] Carlson, J. L. **Redis in Action**. Manning Publications, 2013.

[4] Gray, J. **The Transaction Concept: Virtues and Limitations**. In: Proceedings of the 7th International Conference on Very Large Data Bases (VLDB), p. 144-154, 1981.

[5] Sanfilippo, S. (antirez). **Is Redlock safe?** Redis Labs Technical Blog, 2015. Disponível em: `http://redis.io/topics/distlock`

[6] Obe, R.; Hsu, L. **PostgreSQL: Up and Running**. 3rd edition. O'Reilly Media, 2017.

[7] Kleppmann, M. **Designing Data-Intensive Applications**. O'Reilly Media, 2017.

[8] Coulouris, G.; Dollimore, J.; Kindberg, T.; Blair, G. **Distributed Systems: Concepts and Design**. 5th edition. Addison-Wesley, 2011.

[9] Redisson Project. **Redisson - Redis Java Client**. Disponível em: `https://redisson.org/` Acesso em: outubro de 2024.

[10] Spring Framework Documentation. **Spring Boot Reference Guide**. Pivotal Software, 2024. Disponível em: `https://spring.io/projects/spring-boot`