Jogo de Cartas Multiplayer com Blockchain

[1] **Cláudio Daniel Figueredo Peruna , [1] David Neves Dias**

[1]Departamento de Tecnologia – Universidade Estadual de Feira de Santana (UEFS)
44036–900 – Feira de Santana – Bahia

danielperuna2012@gmail.com

davidneves852@gmail.com

Abstract. *This report describes the third delivery of "Attribute War", in which the distributed Go cluster gains a blockchain-backed economy. A private Ethereum node and the PackRegistry smart contract now mediate pack scarcity, NFT-style card ownership, trades, and match audit, while the token ring, RESTful server-to-server APIs, and TCP publish/subscribe bus preserve matchmaking and gameplay. Transaction-receipt checks, pack and cluster stress harnesses, and UUID-tagged cards deliver verifiable stock control, traceable collections, and resilient matches across the Dockerised trio of servers and the blockchain node.*

Keywords: *Blockchain, Ethereum, Smart contracts, Multiplayer games, Go.*

Resumo. *Este relatório apresenta a terceira entrega do "Attribute War", na qual o cluster distribuído em Go recebe uma economia ancorada em blockchain. Um nó Ethereum privado e o contrato* PackRegistry *passam a intermediar escassez de pacotes, posse de cartas com identificadores únicos, trocas e auditoria de partidas, enquanto o token ring, as APIs REST interservidores e o barramento* publish/subscribe *para clientes TCP mantêm o* matchmaking *e a jogabilidade. Checagens de recibo de transação, testes de estresse de pacotes e de* cluster *e cartas etiquetadas com UUID asseguram controle verificável de estoque, coleções rastreáveis e partidas resilientes nos três servidores em contêineres e no nó blockchain.*

Palavras-chave: *Blockchain, Ethereum, Contratos inteligentes, Jogos multiplayer, Go.*

1. Introdução

Esta terceira etapa do projeto "Attribute War" acrescenta uma camada de blockchain ao *cluster* distribuído, eliminando a lacuna de custódia e auditoria observada no PBL anterior. A economia de pacotes e a troca de cartas passam a exigir transações confirmadas, evitando que a abertura de pacotes, a duplicação de cartas ou a fraude em trocas dependam de confiança no servidor anfitrião. O objetivo desta entrega é demonstrar como a cadeia privada mantém escassez verificável, rastreabilidade de proprietários e registro imutável de resultados sem quebrar a experiência em tempo real.

A infraestrutura mantém três servidores de jogo em Go orquestrados com Docker e adiciona um nó Ethereum (Geth) e o contrato *PackRegistry* compilado

via Hardhat/abigen. Cada servidor conecta-se ao nó por variáveis de ambiente (`BLOCKCHAIN_NODE_URL`, `CONTRACT_ADDRESS`, `ADMIN_PRIVATE_KEY`), preserva o token ring para o estoque global de cartas em partidas e continua a usar *matchmaking* RESTful (`/api/find-opponent`, `/api/request-match`) e o barramento *publish/subscribe* para clientes TCP. O contrato expõe operações de estoque (`getStock`, `decrementStock`), posse (`assignCards`, `transferCard`) e trilha de partidas (`recordMatch`), consumidas pelo cliente Go `blockchain.Client` com espera ativa de recibos para garantir finalização.

O fluxo de abertura de pacotes debita o estoque *on-chain*, gera cartas base via `PackSystem`, anexa um UUID a cada carta e registra a posse antes da entrega ao cliente (`PACK_OPENED`). A troca de cartas (`/trade`) executa `transferCard` e devolve confirmações tanto ao remetente quanto ao destinatário se estiver online; a coleção (`/collection`) consulta `getUserCards` para exibir NFTs já registrados. Resultados de partidas invocam `recordMatch` em *goroutine*, mantendo o jogo responsivo enquanto a auditoria é selada no *ledger*. A suíte de testes reutiliza cenários de concorrência e estresse (`packs_test.go`, `stress_cluster_test.go`) para verificar controle de estoque e continuidade do *cluster*, agora subordinados ao modelo de confirmação de transações.

A **Figura 1** ilustra a topologia com o anel de servidores, o nó Ethereum e o contrato *PackRegistry*, destacando os fluxos REST, TCP e chamadas *on-chain*.

Para orientar o leitor, a Seção 2 revisita fundamentos de blockchain aplicados a jogos, contratos Ethereum e consistência distribuída. A Seção 3 descreve a metodologia e a implementação do cliente Go, integração com o nó Geth e ajustes no protocolo de jogo. A Seção 4 discute os resultados experimentais e implicações de segurança. A Seção 5 encerra com conclusões e extensões futuras, como reconciliar token ring e *ledger* público para *marketplaces* externos.

2. Fundamentação Teórica

A evolução de arquiteturas para jogos massivos multijogador (MMO) tem migrado de modelos puramente centralizados para ecossistemas distribuídos que priorizam a propriedade de ativos digitais e a transparência das regras de negócio. O projeto atual adota uma abordagem híbrida, combinando a performance de servidores de jogo distribuídos com a segurança e imutabilidade de uma rede *Blockchain* para o gerenciamento de estados críticos.

2.1. Blockchain e Livros-Razão Distribuídos

Diferente do protótipo anterior, que dependia de um anel lógico de servidores para consistência, a nova solução utiliza uma *Blockchain* como camada base de verdade. Uma *blockchain* é, essencialmente, um livro-razão distribuído e anexado sequencialmente (*append-only*), onde transações são validadas por consenso entre nós participantes, eliminando a necessidade de uma autoridade central confiável [Zheng et al. 2017].

No contexto do projeto, a rede Ethereum é utilizada para garantir que o "estoque global de pacotes" e a propriedade das cartas sejam geridos de forma descentralizada. A arquitetura da Ethereum implementa uma máquina de estados quase-Turing-completa (*EVM - Ethereum Virtual Machine*), permitindo não apenas o registro de trans-

ferências de valor, mas a execução de código arbitrário replicado em todos os nós da rede [Wood 2014]. Isso resolve nativamente o problema de *double-spending* (ou dupla abertura de pacotes) sem a complexidade de algoritmos de eleição de líder ou recuperação de tokens perdidos.

2.2. Smart Contracts como Regras de Negócio

A lógica de distribuição de cartas e registro de partidas foi encapsulada em **Smart Contracts** escritos na linguagem Solidity. O conceito de contratos inteligentes, introduzido teoricamente por Nick Szabo, refere-se a protocolos computacionais que facilitam, verificam ou impõem a negociação ou execução de um contrato digitalmente [Szabo 1997].

O contrato `PackRegistry.sol` atua como um terceiro confiável e autônomo. Ao invés de confiar que um servidor de jogo respeite o limite de estoque, o código imutável na *blockchain* rejeita matematicamente qualquer transação que tente decrementar o estoque abaixo de zero ou transferir uma carta sem a assinatura criptográfica do proprietário. Isso garante integridade forte (*strong consistency*) para ativos digitais, enquanto libera os servidores de jogo para lidarem com a lógica rápida e volátil das partidas [Antonopoulos and Wood 2018].

2.3. Arquitetura Híbrida: On-chain e Off-chain

Sistemas de jogos baseados em *blockchain* frequentemente enfrentam o desafio da escalabilidade e latência. Para mitigar isso, o projeto adota um padrão de arquitetura híbrida:

- **Off-chain (Alta Velocidade):** A comunicação em tempo real durante as batalhas (troca de movimentos, cálculo de dano) ocorre via conexões TCP diretas e APIs REST entre servidores Go, mantendo a baixa latência necessária para a experiência do usuário [Fielding 2000].
- **On-chain (Alta Segurança):** Eventos críticos, como a abertura de um pacote de cartas, o resultado final de uma partida e a troca de cartas entre jogadores, são submetidos à *blockchain* via transações assinadas.

Essa separação permite que o sistema escale horizontalmente o *matchmaking* (via *publish-subscribe* e orquestração de contêineres Docker), enquanto mantém um registro auditável e persistente dos ativos no livro-razão compartilhado [Xu et al. 2019].

2.4. Integração e Ferramentas

A interoperabilidade entre o ambiente de execução `Go` e a *Ethereum Virtual Machine* é viabilizada pelo uso de *bindings* gerados (via `abigen`) e pelo protocolo JSON-RPC. O cliente Go atua como um nó leve ou interage com um nó Geth (*Go-Ethereum*), assinando transações com chaves privadas geridas localmente. A orquestração de todo esse ambiente — incluindo nós de *blockchain* privados, servidores de aplicação e bancos de dados — continua sendo gerida via Docker Compose, essencial para garantir a reprodutibilidade dos testes de consenso e mineração em ambiente laboratorial.

A Tabela 1 atualiza os pilares teóricos do projeto, refletindo a substituição do anel de tokens pela tecnologia de registros distribuídos.

A **Figura 1** (Figura 1) ilustra a topologia consolidada: o anel de servidores de jogo conecta-se ao nó Ethereum, executa o token ring para mãos iniciais e submete transações ao contrato *PackRegistry*.

Tabela 1. Pilares teóricos atualizados para a arquitetura baseada em Blockchain.

Pilar	Referência Principal	Aplicação no Projeto
Blockchain Architecture	Wood [Wood 2014], Zheng *et al.* [Zheng et al. 2017]	Livro-razão imutável para persistência do estado global (estoque e inventário).
Smart Contracts	Szabo [Szabo 1997], Antonopoulos [Antonopoulos and Wood 2018]	Lógica autônoma para distribuição de pacotes e validação de trocas (`PackRegistry`).
Arquitetura Híbrida	Xu *et al.* [Xu et al. 2019]	Separação entre jogabilidade (Off-chain) e liquidação de ativos (On-chain).
RESTful APIs	Fielding [Fielding 2000]	Comunicação síncrona entre servidores para *matchmaking*.
Publish-Subscribe	Eugster *et al.* [Eugster et al. 2003]	Desacoplamento de mensagens de eventos para clientes conectados.
Concorrência em Go	The Go Authors [The Go Authors 2025]	Gerenciamento de múltiplas conexões TCP e *listeners* de eventos da *chain*.

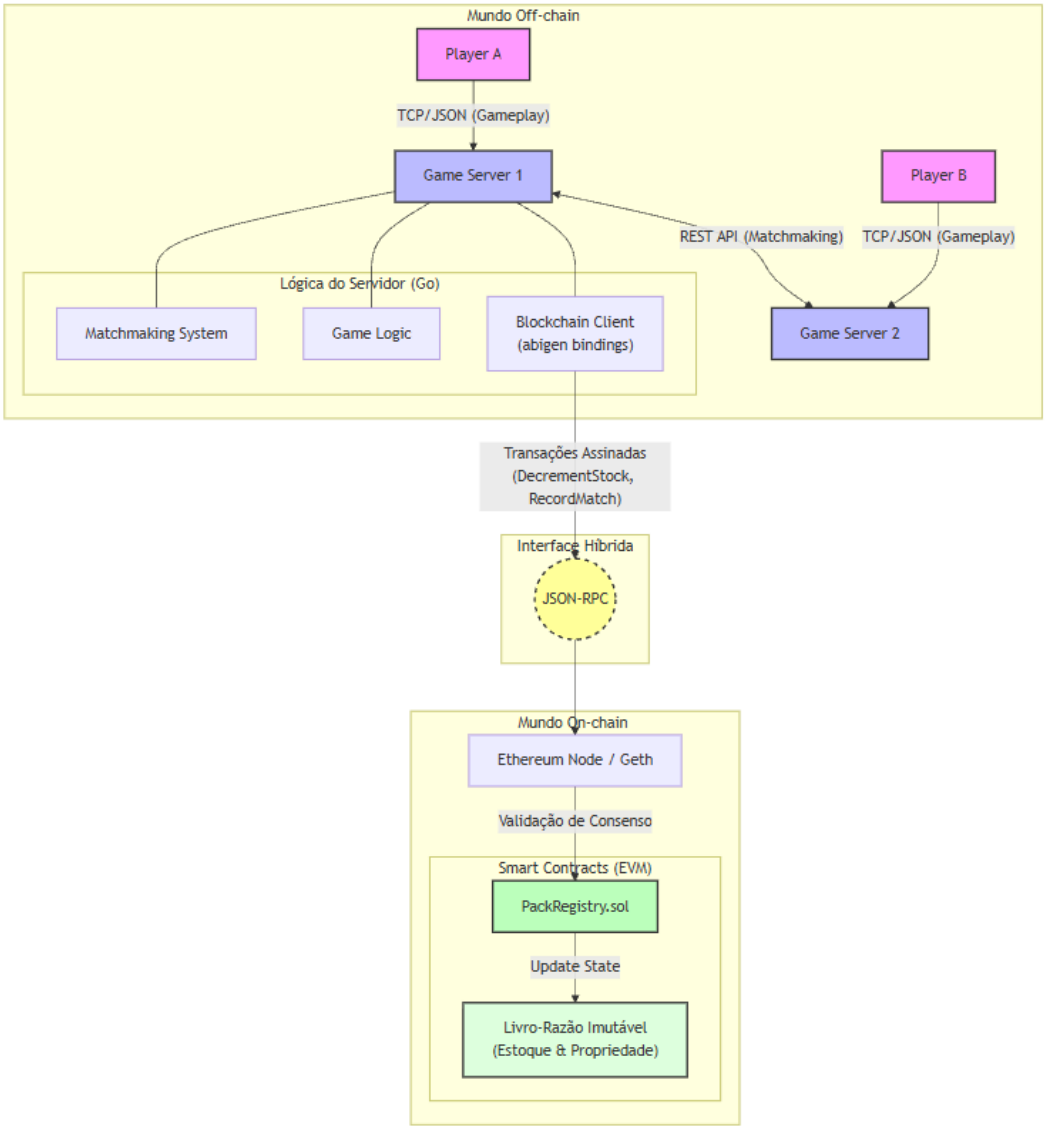

Figura 1. Topologia distribuída com token ring e integração ao contrato *PackRegistry*.

3. Metodologia, Implementação e Testes

A metodologia adotada para o Problema 3 concentrou-se em incorporar uma camada de blockchain ao *cluster* distribuído do *Attribute War*, garantindo escassez verificável de pacotes e rastreabilidade das cartas sem comprometer o desempenho das partidas. O processo abrangeu o provisionamento conjunto do anel de servidores e do nó Ethereum, o ciclo do token ring para mãos iniciais, a integração das APIs interservidores, o roteamento *publish/subscribe* para clientes TCP e a validação automatizada de cenários de concorrência e de liquidação *on-chain*.

Provisionamento do cluster e parâmetros operacionais

O ciclo inicia com a orquestração declarativa em `docker-compose.yml`: três serviços `server-n`, um `client` e o `blockchain-node` compartilham a rede `game-net`. Variáveis de ambiente de cada servidor (`LISTEN_ADDR`, `API_ADDR`, `ALL_SERVERS`, `BLOCKCHAIN_NODE_URL`, `CONTRACT_ADDRESS`, `ADMIN_PRIVATE_KEY`) definem portas, topologia do anel e credenciais de acesso ao contrato *PackRegistry*, preservando reprodutibilidade do ambiente [Docker, Inc. 2025]. No *bootstrap*, `server/main.go` determina sua posição no anel, configura o cliente Ethereum (`blockchain.NewClient`) com checagem de *chainId* e carrega o *State Manager*, o *Broker* e o serviço de matchmaking.

Ciclo do token ring, ledger e distribuição de cartas

O estoque usado nas partidas permanece governado por um token que circula em anel, oferecendo exclusão mútua e ordem total para retirada de cartas iniciais. O `MatchmakingService.Run()` bloqueia até receber o token, processa filas locais e distribui mãos com `DrawCards()` para `CreateLocalMatchWithCards()` ou `CreateDistributedMatchAsHostWithCards()`. O líder regenera o token a partir de `cards.json` quando o *watchdog* expira, evitando esgotamento. Em paralelo, a escassez global de pacotes é delegada ao contrato *PackRegistry*, que rejeita decréscimos indevidos e modela a propriedade como estado imutável [Wood 2014, Szabo 1997].

Matchmaking distribuído e API interservidores

A cooperação entre servidores segue o estilo REST com mensagens JSON [Fielding 2000]. Quando apenas um jogador está na fila local, o host percorre `ALL_SERVERS` via `GET /api/find-opponent`; em caso de sucesso, envia `POST /api/request-match` com as cartas do convidado retiradas do token. Jogadas remotas usam `POST /matches/{id}/action`, aplicando `match.PlayCard()` em ambos os lados; o protocolo impede eco de mensagens ao distinguir anfitrião e convidado. A passagem do token entre nós (`/api/receive-token`) garante que apenas um servidor manipule o *pool* de cartas de partida por vez, enquanto o *ledger* on-chain persiste ativos.

Canal cliente-servidor e roteamento publish-subscribe

Clientes TCP autenticam-se, assinam os tópicos `player.<id>` no `pubsub.Broker` e recebem `MATCH_FOUND`, `STATE` e `ROUND_RESULT`. O padrão *publish/subscribe* desacopla produtores de consumidores, reduzindo bloqueios entre *goroutines* [Eugster et al. 2003]. A abertura de pacotes

OPEN_PACK aciona blockchain.DecrementStock(), espera o recibo (WaitForTransactionReceipt), gera cartas base via PackSystem, anexa UUIDs, registra a posse com AssignCards() e retorna PACK_OPENED com estoque atualizado. A troca /trade chama TransferCard() e notifica remetente e destinatário; a consulta /collection utiliza GetUserCards() para exibir NFTs já registrados. O PING/PONG segue periódico para medir latência fim a fim.

Automatização de testes e monitoramento

A validação automatizada usa go test em dois eixos. O tests/packs_test.go modela concorrência de abertura de pacotes com 20 *goroutines*, aferindo atomicidade, ausência de duplicatas e exaustão do estoque, com logs de auditoria. O tests/stress_cluster_test.go compila o servidor, sobe três instâncias, injeta centenas de clientes simulados e verifica a proporção entre PACK_OPENED e OUT_OF_STOCK, mesmo sob falha intencional de um nó. Logs estruturados ([BLOCKCHAIN], [MATCHMAKING], [MATCH]) e o RTT medido por PING/PONG auxiliam o diagnóstico. A Tabela 2 resume os casos principais e as evidências esperadas.

Fluxo consolidado

O fluxo consolidado segue os passos:

1. Inicialização: cada servidor descobre sua posição no anel, cria clientes Ethereum e valida acesso ao contrato.
2. Regeneração do token: o líder embaralha o *pool* de cartas e injeta o token para iniciar o ciclo.
3. Matchmaking: preferência por partidas locais; na falta de oponente, consulta remota e cria partida distribuída com cartas fornecidas pelo token.
4. Liquidação de ativos: pedidos OPEN_PACK e /trade emitem transações, aguardam recibos e notificam clientes.
5. Observabilidade: PING/PONG e logs estruturados expõem latência, consumo de estoque e hashes de transação.
6. Repassa token: após processar filas e partidas, o nó serializa o token e envia ao próximo servidor, reiniciando o ciclo.

Essa metodologia alinha a arquitetura distribuída com garantias fortes de posse e de escassez providas pelo *ledger*, preservando a reprodutibilidade e a validação automatizada requeridas para o Problema 3.

4. Resultados e Discussões

Esta seção apresenta as evidências coletadas nas campanhas de testes descritas na Metodologia, cobrindo integração com o contrato *PackRegistry*, consistência do token ring nas mãos iniciais, operação das APIs REST para partidas distribuídas, latência observada e resiliência a falhas. Todos os experimentos foram executados no ambiente orquestrado por Docker Compose, garantindo condições controladas e reproduzíveis [Docker, Inc. 2025].

Liquidação on-chain e controle de estoque

O fluxo OPEN_PACK foi exercitado em dezenas de requisições simultâneas. Cada pedido resultou em (i) chamada decrementStock no contrato, (ii) espera pelo recibo via

Tabela 2. Casos de teste e evidências de validação do cluster com blockchain.

ID	Escopo	Procedimento	Evidência de sucesso
T1	Conexão blockchain	Inicializar `docker-compose up`	Logs `[BLOCKCHAIN]` exibem *chainId* e estoque inicial lido do contrato.
T2	Token ring	Receber e repassar token entre os três servidores	Mensagens `Token recebido/passo` alternam entre nós sem bloqueio.
T3	Partida distribuída	Conectar clientes em servidores distintos e acionar `FIND_MATCH`	Sequência `find-opponent` → `request-match` → `MATCH_FOUND` simétrica; mãos idênticas em ambos os lados.
T4	Abertura de pacote *on-chain*	Enviar `OPEN_PACK` e aguardar confirmação	Decremento de estoque confirmado, UUIDs atribuídos e retorno `PACK_OPENED`.
T5	Troca de cartas	Executar `/trade` entre dois jogadores	`TransferCard` aceita, recibo confirmado e mensagem de recebimento para o destinatário.
T6	Tolerância a falhas	Matar um servidor durante o teste de estresse	Demais nós continuam a responder; proporção `PACK_OPENED`/`OUT_OF_STOCK` coerente com estoque global.

WaitForTransactionReceipt, (iii) geração de cartas base e anexação de UUIDs, (iv) chamada assignCards e (v) entrega da mensagem PACK_OPENED. Os logs [BLOCKCHAIN] mostraram hashes distintos para débito e atribuição, e [HANDLER] reportou o estoque lido em GetStock() após cada confirmação. Nenhum pacote foi entregue sem recibo minerado, e não houve respostas OUT_OF_STOCK enquanto o contrato mantinha saldo positivo. A Tabela 3 resume os indicadores principais.

Tabela 3. Indicadores do teste de abertura de pacotes com liquidação on-chain.

Indicador	Valor observado	Efeito
Requisições concorrentes	20 goroutines	Exercitou confirmações de transação
Estoque inicial (contrato)	10 pacotes	Recurso escasso controlado pela EVM
Pacotes liberados	10 (100% do estoque)	Nenhum débito sem recibo minerado
Falhas controladas	10 OUT_OF_STOCK	Sem erros inesperados de RPC
Entradas de auditoria	10 registros de assignCards	Rastreabilidade de dono por UUID
Estoque final (contrato)	0	Sem saldo negativo

Nos testes de integração do cluster, cada servidor consumiu cartas iniciais somente quando detinha o token, como evidenciado por mensagens [MATCHMAKING] Pegou 10 cartas do token para a partida seguidas de [MATCHMAKING] A passar o token .... A reintrodução automática (refillPool_unsafe) foi acionada quando o pool caiu abaixo do limiar, emitindo [TOKEN] Pool reabastecido. Além disso, o registro recordMatch foi disparado ao final das partidas com vencedores definidos, gravando IDs no *ledger*.

Partidas distribuídas e sincronização de estado

Jogadores conectados a server-1 e server-2 foram pareados via find-opponent/request-match. O lote de dez cartas retirado do token foi particionado em mãos simétricas para anfitrião e convidado, confirmado por logs [MATCH] em ambos os nós. Cada jogada remota percorreu POST /matches/{id}/action; forwardPlayIfNeeded() evitou eco e manteve determinismo. Os resultados de rodada (ROUND_RESULT) exibiram danos e HP idênticos, e MATCH_END foi entregue simultaneamente aos dois clientes.

Observabilidade de latência e vazão

O cliente CLI mediu o RTT com PING/PONG periódicos. Durante partidas locais e distribuídas, os valores permaneceram em poucos milissegundos, coerentes com rede local e sem perdas de PING. Os logs [MATCH] e [MATCHMAKING] indicaram vazão sustentada de mensagens sem filas internas, mesmo durante confirmação de transações, pois estas são aguardadas em *goroutine* separada.

Tolerância a falhas e reabastecimento

Ao interromper deliberadamente um dos três servidores durante tests/stress_cluster_test.go, os demais nós continuaram a responder: pacotes foram abertos até o estoque on-chain zerar, e as proporções

PACK_OPENED/OUT_OF_STOCK corresponderam ao saldo do contrato. O *watchdog* de token promoveu o próximo nó vivo e reconfigurou o anel, evitando travamentos. O pipeline completo abrange leitura do `cards.json`, rota do token, chamadas *on-chain* de estoque e transferências, e auditoria de partidas.

Em síntese, o sistema apresentou:

- **Escassez verificável:** nenhum pacote foi emitido sem confirmação on-chain, e o estoque final coincidiu com o registrado no contrato.
- **Consistência interservidores:** partidas distribuídas mantiveram HP, mãos e resultados idênticos entre anfitrião e convidado.
- **Observabilidade:** logs estruturados e RTT expuseram latência e hashes de transação para diagnóstico.

Como ponto de atenção, o reabastecimento do token para mãos iniciais ainda depende do fallback local `refillHands()` em partidas longas. Embora não tenha causado inconsistências, evoluir esse mecanismo para consumos exclusivamente mediados pelo token reduzirá o risco de divergências futuras.

5. Conclusão

O terceiro ciclo do *Attribute War* comprovou que a incorporação da blockchain ao cluster distribuído tornou verificáveis a escassez de pacotes e a posse de cartas, sem degradar a jogabilidade em tempo real. O token ring manteve a coerência das mãos iniciais, enquanto o contrato *PackRegistry* arbitrou débitos e transferências, eliminando duplicações e ancorando auditoria de partidas. A API REST interservidores e o barramento *publish/-subscribe* preservaram sincronismo entre anfitrião e convidado; testes automatizados e logs estruturados evidenciaram partidas distribuídas consistentes, confirmações on-chain e latência controlada.

Os objetivos foram alcançados: pacotes só foram liberados com recibo minerado, `recordMatch` registrou vencedores no *ledger*, e o token circulou mesmo sob falhas transitórias de nós. Persistem limitações: o reabastecimento do token para partidas longas ainda depende do `refillHands()`, o que deixa parte do ciclo de cartas fora do controle estrito do token; além disso, falta telemetria externa (histogramas de RTT, contadores de transações e de partidas) para fechar o ciclo de observabilidade.

Próximos passos incluem estender o token para fornecer cartas adicionais em rodadas prolongadas, expor métricas de desempenho e de blockchain em painéis agregados e ampliar testes de caos (atraso deliberado na passagem do token, falhas de RPC ou de mineração). Essas evoluções consolidariam o ecossistema distribuído, elevando segurança e confiabilidade para operação em ambientes multi-região.

Referências

Antonopoulos, A. M. and Wood, G. (2018). *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Sebastopol, CA.

Docker, Inc. (2025). Compose file reference. Docker. Disponível em: `https://docs.docker.com/reference/compose-file/`. Acesso em: 16 set. 2025.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131.

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

Szabo, N. (1997). Smart contracts: Formalizing and securing relationships on public networks. *First Monday*, 2(9).

The Go Authors (2025). A tour of go: Concurrency. The Go Project. Disponível em: `https://go.dev/tour/concurrency`. Acesso em: 16 set. 2025.

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper.

Xu, X., Weber, I., and Staples, M. (2019). *Architecture for Blockchain Applications*. Springer, Cham.

Zheng, Z., Xie, S., Dai, H.-N., Chen, X., and Wang, H. (2017). An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 557–564. IEEE.