Uma Biblioteca para Consenso Distribuído Baseada no Algoritmo de Blockchain Raft

Henrique Árabe Neres de Farias[1], **Calebe de Paula Bianchini**[1,2]

[1]Faculdade de Computação e Informática (FCI)
Universidade Presbiteriana Mackenzie – São Paulo, SP – Brasil

[2]CESAR – Centro de Estudos e Sistemas Avançados do Recife
Recife, PE – Brasil

henrique.farias@mackenzista.com.br, calebe.bianchini@mackenzie.br,

cpb@cesar.org.br

Abstract. *This paper presents a modular Go library that encapsulates the Raft consensus algorithm through well-defined interfaces (`StateMachine`, `Transport`, and `Storage`), allowing it to be reused across different distributed systems without modifying the library. Academic implementations are usually coupled to a single domain and production-grade libraries are large; the proposed library instead offers a compact code base suited to study and extension. It was validated through 31 automated tests and two proofs of concept in distinct domains: a distributed key-value store and a QuakeWorld-inspired game. Benchmarks showed throughput of up to 1,963 cmd/s under burst load and leader failure recovery in under 500 ms.*

Resumo. *Este trabalho apresenta uma biblioteca modular em Go que encapsula o algoritmo de consenso Raft por meio de interfaces bem definidas (`StateMachine`, `Transport` e `Storage`), permitindo reutilizá-lo em diferentes sistemas distribuídos sem alterar a biblioteca. As implementações acadêmicas costumam ser acopladas a um domínio e as bibliotecas de produção são de grande porte; a biblioteca proposta oferece uma base de código compacta, adequada a estudo e extensão. Ela foi validada por 31 testes automatizados e por duas provas de conceito em domínios distintos: um key-value store distribuído e uma integração com jogo inspirada no QuakeWorld. Benchmarks demonstraram throughput de até 1.963 cmd/s em rajada e recuperação de falha do líder em menos de 500ms.*

1. Introdução

Sistemas distribuídos demandam mecanismos de consenso para garantir que múltiplos nós mantenham o mesmo estado, mesmo diante de falhas de rede ou interrupções de servidores. O algoritmo Raft [Ongaro and Ousterhout 2014] é amplamente utilizado nesse contexto por oferecer uma abordagem mais compreensível que o Paxos [Lamport 1998], separando o consenso em três fases: eleição de líder, replicação de *logs* e segurança.

Apesar da popularidade do Raft, as implementações disponíveis dividem-se em dois grupos. As bibliotecas voltadas à produção, como a da HashiCorp

[HashiCorp 2024], a do etcd [etcd-io 2024] e a Dragonboat [Ni 2024], possuem bases de código extensas que dificultam seu uso como objeto de estudo e extensão. Já as implementações acadêmicas são compactas, porém acopladas aos seus domínios: [Pozzan and Vardanega 2022] aplicaram o Raft em jogos *multiplayer* utilizando tipos específicos (`GameLog`) e comunicação direta via `net/rpc`, e [Neto et al. 2023] estenderam essa proposta integrando o Raft à *game engine* Unity. Em ambos os casos, reutilizar a lógica do Raft em outro contexto exige modificar o código-fonte.

Este trabalho apresenta uma biblioteca modular em Go que encapsula o Raft por meio de interfaces genéricas, permitindo seu uso em diferentes aplicações distribuídas sem alteração do código da biblioteca. São objetivos específicos: (i) refatorar a lógica do Raft a partir de implementações de referência; (ii) definir interfaces que desacoplem o algoritmo da aplicação; (iii) implementar e validar os módulos do Raft (eleição, replicação, *snapshot*, mudança de configuração, persistência); e (iv) avaliar o desempenho por meio de *benchmarks* em cenários de operação normal e de falha.

2. Arquitetura da Biblioteca

A biblioteca, denominada `raft-lib`, foi organizada em quatro pacotes conectados por interfaces, conforme a Figura 1. O pacote `types` contém os tipos compartilhados (`ServerID`, `RaftLog`, `Config`) e a interface `StateMachine`; `transport` define a interface `Transport`, com implementação concreta em `net/rpc`; `storage` define a interface `Storage` para persistência; e `raft` contém a lógica central, dividida por responsabilidade (eleição, replicação, compactação, mudança de configuração, estado, coordenação e métricas).

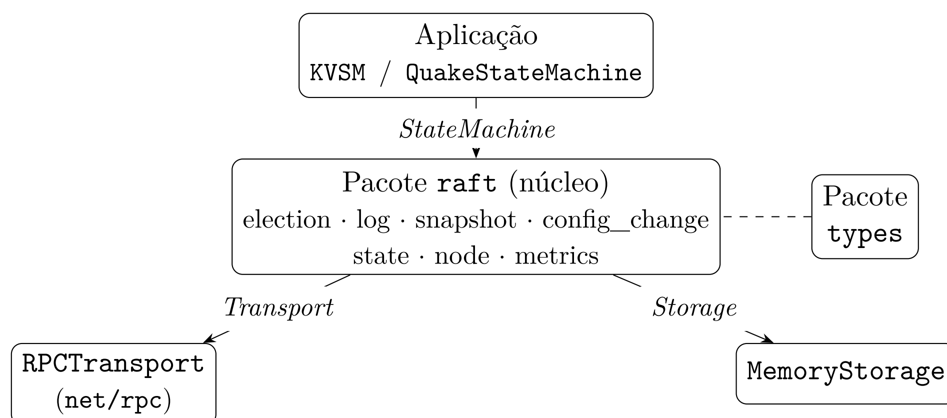

Figura 1. Arquitetura da `raft-lib`: o núcleo comunica-se com a aplicação e a infraestrutura apenas por interfaces.

A interface `StateMachine` exige apenas três métodos (`Apply`, `Snapshot` e `Restore`), o que permite conectar qualquer aplicação ao consenso sem conhecer os detalhes internos do algoritmo e substituir o transporte ou a persistência sem alterar o *core*. O estado de cada nó é protegido por *mutex*, a comunicação entre *goroutines* usa *channels* e as métricas usam contadores atômicos (`sync/atomic`). Em relação ao código de referência [Pozzan and Vardanega 2022], que utiliza o tipo `GameLog` e gerencia conexões diretamente, a biblioteca generaliza o `RaftLog` com um campo

`Data []byte`, abstrai a comunicação por interfaces e corrige o defeito de *snapshot* no qual *logs* duplicados surgiam na interação entre a compactação e atrasos de rede.

3. Implementação e Validação

A implementação foi incremental, com cada módulo testado antes de avançar ao próximo. A eleição usa *goroutines* paralelas com *timeouts* configuráveis; a deduplicação de comandos usa os campos `ClientID` e `SeqNum` no `RaftLog`; a compactação por *snapshots* é disparada quando o *array* de *logs* atinge a capacidade máxima; e a mudança de configuração adota o método simplificado de [Ongaro 2014]. A persistência salva `currentTerm`, `votedFor` e o *log* antes de responder a qualquer RPC, e o transporte concreto usa `net/rpc` com *timeout* de conexão reduzido de 300s para 5s em relação ao código de referência.

A validação foi feita por 21 testes automatizados da `raft-lib`, cobrindo eleição, replicação, *snapshots*, mudança de configuração, integração TCP, persistência, *crash* do líder e métricas, todos passando inclusive com o *flag* `-race`, o que confirma as cinco propriedades de segurança do Raft. A modularidade foi comprovada por duas provas de conceito: um *key-value store*, cuja máquina de estados (`KVSM`) implementa a `StateMachine` sobre um `map[string]string` com operações `SET`, `GET` e `DEL` via JSON; e uma aplicação inspirada no QuakeWorld, cuja `QuakeStateMachine` implementa a mesma interface, com *bridge* UDP e 10 testes adicionais. Ao todo são 31 testes, e ambas usam a biblioteca sem alteração do código-fonte.

4. Análise dos Resultados

Para avaliar quantitativamente a biblioteca, um programa de *benchmark* executa *clusters* de três nós com `RPCTransport` TCP real. Em carga constante, com cada cliente enviando comandos a cada 50ms (20 Hz), observou-se *throughput* de 98,6 cmd/s com 5 clientes e 196,7 cmd/s com 10, ambos com latência média próxima de 51ms (p99 de 51,8 e 51,9ms). Essa latência é dominada pelo intervalo de *heartbeat* (50ms), que define quando o líder envia novos *AppendEntries* RPCs; o processamento Raft em si leva cerca de 1ms. O *throughput* acompanha a carga oferecida, indicando que o consenso não constitui gargalo.

No cenário de rajada, 100 comandos simultâneos foram processados em aproximadamente 51ms, com pico de 1.963 cmd/s, pois o mecanismo de *batching* agrega os comandos pendentes em um único *AppendEntries* RPC. No cenário de falha, o líder foi encerrado abruptamente durante a operação; a recuperação levou 486ms, incluindo o *timeout* de eleição (entre 300 e 500ms) e a eleição propriamente dita. Após a reeleição, o *cluster* voltou a operar sem perda de dados, com a consistência do estado verificada entre os nós e latência p99 transitória de 101,6ms estabilizando nos ciclos seguintes. Esses valores são comparáveis aos reportados por [Pozzan and Vardanega 2022] em condições similares.

5. Considerações Finais

Este trabalho apresentou uma biblioteca modular em Go que encapsula o algoritmo Raft por meio de interfaces genéricas (`StateMachine`, `Transport` e `Storage`), reutilizável em diferentes aplicações distribuídas sem alterar o código da

biblioteca[1]. O foco em uma base compacta, sem dependências externas e derivada das implementações de referência, a distingue das bibliotecas voltadas à produção e a torna adequada para estudo e extensão. A modularidade foi comprovada por duas provas de conceito em domínios distintos, um *key-value store* e uma integração com jogo inspirada no QuakeWorld, que usam a mesma biblioteca sem nenhuma alteração. A biblioteca mantém as propriedades de segurança do Raft, verificadas por 31 testes automatizados, corrige o defeito de *snapshot* documentado por [Pozzan and Vardanega 2022] e apresenta desempenho comparável ao da implementação de referência.

Como limitações, a persistência é apenas em memória, não há transporte alternativo ao `net/rpc` e a avaliação ficou restrita a cenários abaixo do ponto de saturação. Como trabalhos futuros, pretende-se realizar testes de saturação e de escalabilidade (medindo CPU, memória, número de nós e falhas simultâneas), comparar quantitativamente a biblioteca com soluções como etcd/raft, HashiCorp Raft e Dragonboat, implementar transporte em gRPC e persistência em disco, e explorar tolerância a falhas bizantinas.

Agradecimentos

Os autores agradecem o apoio da MackCloud, Laboratório Multidisciplinar de Computação Científica e Nuvem[2]; e do projeto SPRACE – Processo nº 2018/25225-9 da FAPESP. Este trabalho foi financiado em parte pelo Fundo Mackenzie de Pesquisa e Inovação (MackPesquisa) – Projetos nº 231009 e 251005.

Referências

etcd-io (2024). raft: Raft library for maintaining a replicated state machine. `https://github.com/etcd-io/raft`.

HashiCorp (2024). raft: Golang implementation of the Raft consensus protocol. `https://github.com/hashicorp/raft`.

Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169.

Neto, P. U. et al. (2023). Raft Unity game. `https://github.com/PedroUnello/raft-unity-game`.

Ni, L. (2024). Dragonboat: A high performance multi-group Raft library in Go. `https://github.com/lni/dragonboat`.

Ongaro, D. (2014). *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference*, pages 305–320, Philadelphia, PA, USA. USENIX Association.

Pozzan, G. and Vardanega, T. (2022). Rafting multiplayer video games. *Software: Practice and Experience*, 52(4):1065–1091.

[1] O código está disponível em `https://github.com/HenriqueArabe/raft-lib` e a integração com o QuakeWorld em `https://github.com/HenriqueArabe/quake-raft`.

[2] `https://mackcloud.mackenzie.br`