Algoritmos de Consenso Distribuído: Paxos e Raft

Introdução

Em sistemas distribuídos, garantir que múltiplos processos concordem sobre um valor comum — mesmo na presença de falhas — é fundamental para manter a **consistência** e a **disponibilidade** dos serviços. Esse problema é conhecido como **consenso distribuído**.

Dois dos algoritmos mais importantes para resolver o consenso distribuído são o **Paxos**, desenvolvido por Leslie Lamport, e o **Raft**, criado por Diego Ongaro e John Ousterhout. Neste texto, exploraremos a origem, o funcionamento, as vantagens e desvantagens desses algoritmos, e seu impacto na computação distribuída.

História e Contexto

Antes do Paxos

Antes do Paxos, os sistemas distribuídos enfrentavam o desafio de coordenar múltiplas máquinas para manter uma visão consistente dos dados, principalmente em ambientes com falhas de comunicação ou de máquinas.

O problema do consenso foi formalizado na década de 1980, com a publicação do famoso trabalho de Fischer, Lynch e Paterson (FLP) em 1985, que provou que é impossível atingir consenso de forma determinística em sistemas assíncronos se mesmo um nó pode falhar (FLP Impossibility Result). Isso fez com que a comunidade se voltasse para soluções que pudessem garantir consenso em ambientes parcialmente síncronos ou com probabilidades de falha.

Paxos (Leslie Lamport, 1998)

Em 1998, Leslie Lamport publicou o artigo “The Part-Time Parliament” [Lamport, 1998], onde apresentou o algoritmo **Paxos**, um método formal para alcançar consenso em sistemas distribuídos com falhas.

O nome “Paxos” vem de uma ilha grega, escolhida arbitrariamente por Lamport. O algoritmo foi inicialmente visto como complexo e difícil de entender, mas se tornou a base para muitos sistemas distribuídos modernos.

Paxos resolve o problema do consenso mesmo que alguns dos nós falhem ou que as mensagens sejam atrasadas, garantindo segurança (não há dois processos que decidem valores diferentes) e eventual progresso (desde que algumas condições sejam atendidas).

Raft (Ongaro & Ousterhout, 2014)

Em 2014, Diego Ongaro e John Ousterhout publicaram o artigo “In Search of an Understandable Consensus Algorithm” [Ongaro & Ousterhout, 2014]. Eles destacaram que, apesar da robustez do Paxos, sua complexidade dificulta a implementação e o ensino do algoritmo.

Raft foi criado com o objetivo de ser um algoritmo de consenso que mantém as mesmas garantias de Paxos, mas com design modular, mais fácil de entender, implementar e depurar. Raft foca em decompor o problema em subproblemas, como eleição de líder, replicação de log e segurança.

Por que esses algoritmos surgiram?

O consenso distribuído é um problema crucial porque, em sistemas distribuídos (como bancos de dados replicados, sistemas de arquivos distribuídos, ou plataformas de microsserviços), diferentes réplicas devem concordar em qual operação aplicar e em qual ordem.

Antes de Paxos e Raft, as soluções baseavam-se em:

- **Protocolos de coordenação centralizada:** Que dependiam de um servidor mestre único (single point of failure).
- **Soluções baseadas em bloqueios distribuídos:** Que podiam causar deadlocks ou perda de disponibilidade.
- **Algoritmos ad hoc**, muitas vezes difíceis de provar corretos.

Paxos trouxe uma solução formal e tolerante a falhas. Raft melhorou a aplicabilidade prática e a adoção industrial.

Funcionamento básico

Paxos

Paxos é baseado em três papéis principais: **proposers** (que propõem valores), **acceptors** (que aceitam propostas) e **learners** (que aprendem o valor decidido).

O protocolo ocorre em duas fases:

1. **Fase Prepare (prepare phase):** O proposer envia uma proposta com número único para os acceptors pedindo promessa para não aceitar propostas antigas.
2. **Fase Accept (accept phase):** Se os acceptors prometeram, o proposer envia a proposta para aceitação.

O consenso é alcançado quando a maioria dos acceptors aceita o mesmo valor.

Raft

Raft divide o consenso em três componentes principais:

- **Eleição de líder:** Os servidores elegem um líder responsável por gerenciar a replicação.
- **Replicação de log:** O líder aceita comandos do cliente, os adiciona ao seu log e replica para os seguidores.
- **Segurança:** Garantir que logs dos servidores converjam e que as decisões sejam consistentes.

Raft assume que o líder é o ponto central de coordenação, simplificando a lógica de consenso.

Vantagens e Desvantagens

Aspecto	Paxos	Raft
Complexidade	Alta, difícil de entender e implementar	Projetado para ser mais simples e modular
Performance	Similar, pode ser otimizado	Similar, com potencial vantagem prática
Tolerância a falhas	Alta, tolera falhas de nós e redes	Igual, com mecanismo claro de eleição de líder
Adoção industrial	Usado em sistemas como Chubby (Google), Zookeeper	Usado em etcd, Consul, RethinkDB e outros
Documentação e ensino	Pouco didático, base acadêmica rigorosa	Focado no aprendizado e implementação fácil

Claro! Aqui está um texto explicando onde os algoritmos de consenso como **Paxos** e **Raft** são aplicados no dia a dia, e qual o impacto prático deles no mundo real.

Aplicações Práticas dos Algoritmos de Consenso (Paxos e Raft) e seu Impacto no Dia a Dia

Os algoritmos de consenso distribuído, como Paxos e Raft, são fundamentais para garantir a **confiabilidade**, **disponibilidade** e **consistência** de sistemas que usamos cotidianamente, mesmo que isso não seja visível diretamente para o usuário final.

Onde são aplicados?

1. **Sistemas de Arquivos Distribuídos**

Sistemas de arquivos modernos e distribuídos, como o **Google File System (GFS)** e o **Hadoop Distributed File System (HDFS)**, precisam garantir que várias réplicas de dados estejam sempre sincronizadas para evitar perda ou corrupção.

O Paxos é utilizado no **Google Chubby**, um serviço de bloqueio distribuído que coordena o acesso a recursos compartilhados e serve como base para sistemas de arquivos distribuídos.

2. **Bancos de Dados Distribuídos**

Bancos de dados modernos, especialmente aqueles que funcionam em clusters distribuídos, precisam garantir que todas as réplicas concordem sobre a ordem das operações para manter a integridade dos dados.

Exemplos incluem:

- **etcd**: Um banco de dados chave-valor usado para armazenar configurações e coordenar clusters, baseado no Raft.
- **Consul**: Um sistema de descoberta e configuração para microserviços que usa Raft para garantir consistência.
- **Apache Cassandra** e **Google Spanner** utilizam variações do consenso para replicação segura.

3. **Sistemas de Mensageria e Filas de Mensagens**

Serviços que gerenciam filas e mensagens distribuídas, como **Apache Kafka** e **RabbitMQ**, precisam garantir a ordem e a durabilidade das mensagens, mesmo que algumas máquinas falhem.

Alguns desses sistemas utilizam protocolos inspirados em Paxos ou Raft para coordenar líderes e replicar logs.

4. **Orquestração de Containers e Serviços em Nuvem**

Plataformas como **Kubernetes** usam etcd (baseado em Raft) para armazenar o estado do cluster e garantir que múltiplos nós estejam sincronizados sobre o estado atual dos serviços e pods, garantindo que o orquestrador tome decisões corretas sobre escalabilidade e falhas.

5. **Serviços de Alta Disponibilidade**

Muitas aplicações que exigem uptime elevado, como sistemas bancários, plataformas de comércio eletrônico, e serviços de telecomunicações, dependem desses algoritmos para que um grupo de servidores distribuídos atue como uma única entidade confiável, mesmo em caso de falhas parciais.

Impacto no Dia a Dia

Embora os usuários finais raramente percebam, os algoritmos de consenso têm um impacto direto em sua experiência:

- **Disponibilidade contínua:** Ao permitir que sistemas distribuídos continuem operando mesmo com falhas em alguns nós, garantem que serviços essenciais (bancos, redes sociais, plataformas de streaming) estejam sempre disponíveis.
- **Consistência dos dados:** Evitam situações onde diferentes servidores apresentam dados divergentes, garantindo que uma conta bancária, por exemplo, não fique inconsistente em caso de falhas.
- **Escalabilidade:** Permitem que sistemas cresçam horizontalmente (com mais servidores) sem perder a integridade das operações.
- **Recuperação rápida:** Com mecanismos de eleição de líder e replicação de logs, esses algoritmos possibilitam a rápida recuperação e continuidade do serviço após falhas.

Os algoritmos de consenso, como Paxos e Raft, são invisíveis, mas essenciais para a infraestrutura tecnológica que sustenta nosso cotidiano digital. Eles garantem que, ao enviar um e-mail, fazer uma transação bancária, usar um aplicativo na nuvem ou assistir a um vídeo online, os sistemas distribuídos por trás dessas ações funcionem de forma confiável, consistente e resiliente.

Seu impacto vai além da teoria, sendo a base para a confiabilidade de praticamente toda aplicação distribuída moderna — um pilar invisível que sustenta a era da computação em nuvem e dos serviços online.

Beleza! Vou trazer exemplos práticos de empresas e serviços que usam **Paxos** e **Raft** no dia a dia, e também um pouco mais de detalhes técnicos sobre como esses algoritmos atuam nesses contextos reais.

Exemplos Práticos e Detalhes Técnicos do Uso de Paxos e Raft

Exemplos Reais de Uso

1. **Google — Paxos no Chubby**

O **Chubby** é um serviço de coordenação distribuída interno do Google que usa o algoritmo Paxos para fornecer:

- **Serviço de bloqueio distribuído:** Evita que múltiplos processos acessem simultaneamente recursos compartilhados, como arquivos ou bancos de dados.
- **Armazenamento de configurações e eleição de líderes:** Serve para que componentes do Google mantenham consistência de estado e eleição de líderes para serviços críticos, como o Bigtable.

Impacto prático: Milhares de serviços Google dependem do Chubby para funcionar corretamente, incluindo o Bigtable e o Google File System.

2. **CoreOS / Kubernetes — Raft no etcd**

O **etcd** é um banco de dados chave-valor leve usado para armazenar o estado do cluster Kubernetes. Ele implementa o algoritmo Raft para garantir que todas as réplicas estejam sincronizadas e que haja um líder eleito para coordenar mudanças.

- **Eleição de líder:** O líder Raft aceita requisições de escrita e replica o log para seguidores.
- **Replicação de log:** Cada mudança no estado do cluster é registrada no log replicado de forma segura.
- **Failover automático:** Se o líder falhar, um novo é eleito rapidamente, minimizando downtime.

Impacto prático: O funcionamento confiável do Kubernetes — que gerencia milhões de containers em produção — depende diretamente do consenso garantido por Raft no etcd.

3. **HashiCorp Consul**

O **Consul** usa Raft para manter a consistência do seu banco de dados distribuído que armazena informações de configuração e serviços.

- Permite a descoberta dinâmica de serviços.
- Coordena registros e status com forte consistência.

Detalhes Técnicos do Funcionamento em Contextos Reais

Paxos em Chubby

- **Papéis fixos:** Acceptors, proposers e learners são implementados como servidores e clientes do serviço.
- **Majority quorum:** Para aceitar um valor, é preciso consenso da maioria dos acceptors.
- **Persistência:** Os estados dos acceptors são gravados em disco para tolerar falhas.

Essa implementação lida com falhas reais (nós caindo, mensagens perdidas), e é projetada para oferecer **alta disponibilidade** mesmo em condições adversas.

Raft em etcd e Consul

- **Eleição de líder periódica:** Cada nó tem um timeout para começar eleição se não receber batimentos (heartbeats).
- **Logs replicados:** Cada operação (ex: atualizar configuração) é adicionada ao log do líder e replicada para seguidores, que aplicam as mudanças em ordem.
- **Commit seguro:** Um comando só é considerado aplicado após confirmação da maioria.
- **Estado consistente:** Mesmo em caso de falhas, o protocolo garante que os logs nunca divergem em comandos já confirmados.

Por que essa engenharia importa?

Em sistemas distribuídos reais, os desafios são enormes:

- **Falhas e particionamento de rede** podem ocorrer a qualquer momento.
- **Replicação e sincronização de dados** precisam ser rápidas para evitar inconsistência.
- **Escalabilidade e flexibilidade** são necessárias para suportar cargas variáveis e múltiplos usuários.

Paxos e Raft fornecem o rigor formal e prático para lidar com esses desafios, permitindo a construção de sistemas robustos e confiáveis que usamos todos os dias.

Implementação simples

- [1] **Python** — orientada a objetos e fácil de expandir.
- [2] **C** — focando no núcleo do algoritmo para desempenho e controle.

Resumo da Lógica do Raft Simplificado

Vamos implementar um sistema com:

- Estados de **Follower**, **Candidate** e **Leader**

- **Eleição de líder** com timeout aleatório
- **Heartbeat** do líder para manter controle

⚠ *Nota:* Para simplificação, o sistema:

- Usa comunicação via chamadas locais (simulação de RPC)
- Não implementa logs replicados ainda (apenas eleição de líder)

---

[1] Implementação em Python (Raft simplificado)

```
import random
import threading
import time

FOLLOWER = "Follower"
CANDIDATE = "Candidate"
LEADER = "Leader"

class Node:
    def __init__(self, id, cluster):
        self.id = id
        self.cluster = cluster
        self.state = FOLLOWER
        self.votes = 0
        self.term = 0
        self.leader_id = None
        self.timeout = random.uniform(1.5, 3.0)
        self.reset_timer()

    def reset_timer(self):
        self.timer = threading.Timer(self.timeout, self.start_election)
        self.timer.start()

    def start_election(self):
        self.state = CANDIDATE
        self.term += 1
        self.votes = 1  # vote for self
        print(f"[{self.id}] becomes candidate for term {self.term}")
        for peer in self.cluster:
            if peer.id != self.id:
                peer.request_vote(self)

        self.check_votes()

    def request_vote(self, candidate):
        if candidate.term > self.term:
            self.term = candidate.term
            self.state = FOLLOWER
            candidate.receive_vote()
            print(f"[{self.id}] votes for [{candidate.id}]")
```

```
        else:
            print(f"[{self.id}] rejects vote for [{candidate.id}]")

    def receive_vote(self):
        self.votes += 1
        self.check_votes()

    def check_votes(self):
        majority = len(self.cluster) // 2 + 1
        if self.votes >= majority and self.state == CANDIDATE:
            self.become_leader()

    def become_leader(self):
        self.state = LEADER
        self.leader_id = self.id
        print(f"[{self.id}] becomes LEADER for term {self.term}")
        self.timer.cancel()
        self.send_heartbeats()

    def send_heartbeats(self):
        def heartbeat():
            if self.state == LEADER:
                for peer in self.cluster:
                    if peer.id != self.id:
                        print(f"[{self.id}] sends heartbeat to
[{peer.id}]")
                threading.Timer(1.0, heartbeat).start()

        heartbeat()

def simulate_cluster(n=3):
    cluster = []
    for i in range(n):
        cluster.append(Node(i, None))
    for node in cluster:
        node.cluster = cluster
    return cluster

if __name__ == "__main__":
    simulate_cluster()
    time.sleep(10)
```

---

[2] ⚙️ Implementação em C (núcleo da eleição do líder)

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
```

```c
typedef enum {FOLLOWER, CANDIDATE, LEADER} State;

typedef struct Node {
    int id;
    State state;
    int term;
    int votes;
    int total_nodes;
} Node;

void start_election(Node *node, Node *cluster) {
    node->state = CANDIDATE;
    node->term++;
    node->votes = 1;

    printf("[Node %d] is candidate for term %d\n", node->id, node->term);

    for (int i = 0; i < node->total_nodes; i++) {
        if (i == node->id) continue;
        if (node->term > cluster[i].term) {
            cluster[i].term = node->term;
            cluster[i].state = FOLLOWER;
            node->votes++;
            printf("[Node %d] votes for [Node %d]\n", i, node->id);
        }
    }

    if (node->votes > node->total_nodes / 2) {
        node->state = LEADER;
        printf("[Node %d] becomes LEADER\n", node->id);
    }
}

int main() {
    const int N = 3;
    Node cluster[N];

    for (int i = 0; i < N; i++) {
        cluster[i].id = i;
        cluster[i].state = FOLLOWER;
        cluster[i].term = 0;
        cluster[i].votes = 0;
        cluster[i].total_nodes = N;
    }

    srand(time(NULL));
    int random_node = rand() % N;
    sleep(1);
    start_election(&cluster[random_node], cluster);

    return 0;
}
```

🧪 O que você pode fazer a partir daqui?

- Adicionar **replicação de logs**
- Suportar falhas e reinícios
- Usar sockets para simular RPC real
- Visualizar o cluster com logs ou gráfico

Citações de Autores

- Leslie Lamport (1998) define Paxos como "o primeiro protocolo correto para consenso distribuído que é assíncrono e tolera falhas de processo".
- Ongaro e Ousterhout (2014) escreveram que "a complexidade do Paxos tem impedido a ampla adoção, e Raft foi projetado para ser compreensível, com uma abordagem prática para replicação de logs".

Conclusão

Paxos e Raft são algoritmos essenciais para garantir consenso em sistemas distribuídos com falhas. Paxos trouxe uma base teórica e formal fundamental, mas sua complexidade dificultou sua adoção prática. Raft, por sua vez, conseguiu democratizar o consenso distribuído, facilitando o entendimento, implementação e manutenção do protocolo, o que explica sua ampla adoção em projetos de código aberto e em empresas.

Entender ambos os algoritmos é crucial para cientistas da computação que trabalham com sistemas distribuídos, bases de dados, e aplicações que exigem alta disponibilidade e consistência.

Referências

- Lamport, Leslie. "The Part-Time Parliament." *ACM Transactions on Computer Systems* 16, no. 2 (1998): 133–169. [https://doi.org/10.1145/279227.279229]
- Ongaro, Diego, and John Ousterhout. "In Search of an Understandable Consensus Algorithm." *USENIX Annual Technical Conference* (2014). [https://raft.github.io/raft.pdf]
- Fischer, Michael J., Nancy A. Lynch, and Michael S. Paterson. "Impossibility of distributed consensus with one faulty process." *Journal of the ACM (JACM)* 32, no. 2 (1985): 374-382.