**Capítulo 5  
Algoritmos de Consenso**

Consenso é um problema fundamental em sistemas distribuídos. Desde a década de 1970, esse problema tem sido pesquisado no contexto de sistemas distribuídos, mas um interesse renovado surgiu no desenvolvimento de algoritmos de consenso distribuído que sejam adequados para redes blockchain. Neste capítulo, exploraremos as técnicas subjacentes por trás dos algoritmos de consenso distribuído, seu funcionamento interno e novos algoritmos que foram desenvolvidos especificamente para redes blockchain.

Além disso, apresentaremos vários algoritmos bem conhecidos em um cenário tradicional de sistemas distribuídos que também podem ser implementados em redes blockchain com algumas modificações, como Paxos, Raft e PBFT. Também exploraremos outros mecanismos que foram introduzidos especificamente para redes blockchain, como prova de trabalho (PoW), prova de participação (PoS) e versões modificadas de consenso tradicional, como Istanbul Byzantine Fault Tolerant (IBFT), que é uma versão “blockchainizada” do algoritmo Prático de Tolerância a Falhas Bizantinas (PBFT). Ao longo do capítulo, cobriremos os seguintes tópicos:

* Introdução ao consenso
* Análise e design
* Classificação
* Algoritmos
* Escolhendo um algoritmo

Antes de mergulharmos em algoritmos específicos, primeiro precisamos compreender alguns conceitos fundamentais e uma visão geral do problema de consenso.

**Introdução ao consenso**

O problema de consenso distribuído tem sido amplamente estudado na pesquisa de sistemas distribuídos desde o final da década de 1970. Os sistemas distribuídos são classificados em duas categorias principais: troca de mensagens e memória compartilhada. No contexto de blockchain, estamos preocupados com o tipo de sistema distribuído baseado em troca de mensagens, onde os participantes da rede se comunicam entre si por meio do envio de mensagens. Consenso é o processo que permite que todos os processos em uma rede concordem sobre algum valor específico na presença de falhas.

Como vimos no Capítulo 1, *Blockchain 101*, existem diferentes tipos de redes blockchain. Em particular, foram discutidos dois tipos: com permissão e públicas (sem permissão). O problema de consenso também pode ser classificado com base nesses dois paradigmas. Por exemplo, o Bitcoin é uma blockchain pública. Ele executa PoW, também chamado de consenso de Nakamoto. Em contraste, muitas blockchains com permissão tendem a executar variantes do consenso distribuído tradicional ou clássico. Um exemplo importante é o IBFT, que é uma versão em blockchain do PBFT. Outros exemplos incluem Tendermint, Casper FFG e muitas variantes do PBFT.

Uma área comum de pesquisa é converter mecanismos de consenso distribuído tradicionais (clássicos) em suas variantes em blockchain. Outra área de interesse é analisar protocolos de consenso existentes e novos.

**Tolerância a falhas**

Um requisito fundamental em um mecanismo de consenso é a tolerância a falhas em uma rede, e ele deve continuar funcionando mesmo na presença de falhas. Isso naturalmente significa que deve haver algum limite para o número de falhas que uma rede pode suportar, já que nenhuma rede pode operar corretamente se muitos de seus nós falharem. Com base nesse requisito de tolerância a falhas, os algoritmos de consenso também são chamados de algoritmos tolerantes a falhas, e há dois tipos:

* **Tolerância a falhas por travamento (Crash Fault-Tolerance - CFT)**, que cobre apenas falhas por travamento ou, em outras palavras, falhas benignas.
* **Tolerância a falhas bizantinas (Byzantine Fault-Tolerance - BFT)**, que lida com o tipo de falhas que são arbitrárias e podem até ser maliciosas.

A replicação é uma abordagem padrão para melhorar a tolerância a falhas e a disponibilidade de uma rede. A replicação resulta em uma cópia sincronizada de dados em todos os nós de uma rede. Isso significa que, mesmo que alguns nós se tornem defeituosos, o sistema/rede como um todo permanece disponível devido aos dados estarem disponíveis em múltiplos nós. Existem dois principais tipos de técnicas de replicação:

* **Replicação ativa**, que é um tipo onde cada réplica se torna uma cópia da réplica da máquina de estados original.
* **Replicação passiva**, que é um tipo onde há apenas uma única cópia da máquina de estados no sistema mantida pelo nó primário, e os demais nós/réplicas apenas mantêm o estado.

No contexto de mecanismos de consenso tolerantes a falhas, a replicação desempenha um papel vital ao introduzir resiliência no sistema.

A **replicação de máquina de estados** (state machine replication – SMR) é uma técnica de fato usada para fornecer serviços de replicação determinística a fim de alcançar tolerância a falhas em um sistema distribuído. Em um nível abstrato, uma máquina de estados é um modelo matemático usado para descrever uma máquina que pode estar em diferentes estados, mas ocupa um estado de cada vez. Uma máquina de estados armazena um estado do sistema e o transiciona para o próximo estado como resultado da entrada recebida. Como resultado da transição de estado, uma saída é produzida juntamente com um estado atualizado. A ideia fundamental por trás da SMR pode ser resumida da seguinte forma:

1. Todos os servidores sempre começam com o mesmo estado inicial.
2. Todos os servidores recebem requisições de forma totalmente ordenada (sequenciadas conforme geradas pelos clientes).
3. Todos os servidores produzem a mesma saída determinística para a mesma entrada.

A replicação de máquina de estados é implementada sob o paradigma primário/cópia de segurança (primary/backup), onde um nó primário é responsável por receber e difundir as requisições dos clientes. Esse mecanismo de difusão é chamado de **difusão de ordem total** (total order broadcast) ou **difusão atômica** (atomic broadcast), que garante que os nós de backup ou réplica recebam e executem as mesmas requisições na mesma sequência que o primário.

Consequentemente, isso significa que todas as réplicas eventualmente terão o mesmo estado que o primário, resultando assim na obtenção de consenso. Em outras palavras, isso significa que **difusão de ordem total** e **consenso distribuído** são problemas equivalentes; se você resolve um, o outro também está resolvido.

Agora que entendemos os fundamentos da replicação e da tolerância a falhas, é importante entender que a tolerância a falhas funciona até certo limite. Por exemplo, se uma rede tiver uma grande maioria de nós constantemente falhando e links de comunicação comprometidos, não é difícil entender que esse tipo de rede pode não ser tão tolerante a falhas quanto gostaríamos. Em outras palavras, mesmo na presença de medidas de tolerância a falhas, se houver falta de recursos na rede, ela ainda pode não ser capaz de fornecer o nível necessário de tolerância a falhas. Em alguns cenários, pode ser impossível fornecer os serviços exigidos devido à falta de recursos em um sistema. Em computação distribuída, esses cenários impossíveis são pesquisados e relatados como **resultados de impossibilidade**.

**Impossibilidade de FLP**

Resultados de impossibilidade fornecem uma compreensão sobre se um problema é solucionável e os recursos mínimos necessários para isso. Se o problema for insolúvel, então esses resultados fornecem uma compreensão clara de que uma tarefa específica não pode ser realizada e que nenhuma pesquisa adicional é necessária. Sob outro ângulo, podemos dizer que resultados de impossibilidade (às vezes chamados de resultados de insolvência) mostram que certos problemas não são computáveis sob recursos insuficientes. Resultados de impossibilidade revelam aspectos profundos da computação distribuída e nos permitem entender por que certos problemas são difíceis de resolver e sob quais condições um problema anteriormente insolúvel pode ser resolvido.

Os problemas que não são solucionáveis sob quaisquer condições são conhecidos como resultados de insolvência. Este resultado é conhecido como o **resultado de impossibilidade de FLP**, que é um resultado fundamental de insolvência que afirma que em um ambiente assíncrono, o consenso determinístico é impossível, mesmo que apenas um processo esteja com falha.

Para contornar a impossibilidade de FLP, várias técnicas foram introduzidas:

* **Detectores de falhas** podem ser vistos como oráculos associados a processadores para detectar falhas. Na prática, isso geralmente é implementado como um mecanismo de tempo limite (timeout).

**FLP** é nomeado com base nos nomes dos autores, Fischer, Lynch e Patterson. Seu resultado foi apresentado em seu artigo:

Fischer, M.J., Lynch, N.A., e Paterson, M.S., 1982. *Impossibility of distributed consensus with one faulty process* (No. MIT/LCS/TR-282). Massachusetts Institute of Technology, Laboratory for Computer Science.

O artigo está disponível em: <https://apps.dtic.mil/dtic/tr/fulltext/u2/a132503.pdf>

* **Algoritmos randomizados** foram introduzidos para fornecer uma garantia probabilística de terminação. A ideia central por trás dos protocolos randomizados é que os processadores em tais protocolos podem fazer uma escolha aleatória de valor de decisão se o processador não receber o quórum necessário de mensagens confiáveis. Eventualmente, com uma probabilidade muito alta, toda a rede convergirá para um valor de decisão.
* **Suposições de sincronismo**, onde suposições adicionais de sincronização e temporização são feitas para garantir que o algoritmo de consenso tenha tempo adequado para ser executado de modo que possa progredir e terminar.

Agora que entendemos um resultado fundamental de impossibilidade, vejamos outro resultado relevante que destaca a insolubilidade do consenso devido à falta de recursos: o chamado resultado de limite inferior. Podemos pensar em limite inferior como o número mínimo de recursos, por exemplo, o número de processadores ou links de comunicação necessários para resolver um problema. Os resultados mais comuns e fundamentais são os seguintes, onde *f* representa o número de nós com falhas:

* No caso de CFT, são necessários pelo menos **2f + 1** nós para alcançar consenso.
* No caso de BFT, são necessários pelo menos **3f + 1** nós para alcançar consenso.

Agora cobrimos os fundamentos da teoria de consenso distribuído. A seguir, vamos nos aprofundar um pouco mais na análise e no design dos algoritmos de consenso.

**Análise e design**

Para analisar e entender um algoritmo de consenso, precisamos definir um modelo sob o qual nosso algoritmo será executado. Esse modelo fornece algumas suposições sobre o ambiente operacional do algoritmo e fornece uma maneira intuitiva de estudar e raciocinar sobre várias propriedades do algoritmo.

Nas seções seguintes, descreveremos um modelo que é útil para descrever e analisar mecanismos de consenso.

**Modelo**

Sistemas de computação distribuída representam diferentes entidades no sistema sob um modelo computacional. Esse modelo computacional é uma maneira útil de descrever o sistema sob certas suposições de sistema. Um modelo computacional representa processos, condições de rede, suposições de temporização e como todas essas entidades interagem e trabalham juntas. Vamos agora examinar esse modelo em detalhes e apresentar todos os objetos um por um.

**Processos**  
Os processos se comunicam entre si trocando mensagens. Portanto, esses sistemas são chamados de sistemas distribuídos baseados em troca de mensagens. Existe outra classe, chamada de memória compartilhada, que não discutiremos aqui, pois todos os sistemas blockchain são sistemas de troca de mensagens.

**Suposições de temporização**

Há também algumas suposições de temporização que são feitas ao projetar algoritmos de consenso:

* **Sincronismo**: em sistemas síncronos, existe um limite superior conhecido para os atrasos de comunicação e de processamento. Algoritmos síncronos são projetados para serem executados em redes síncronas. Em um nível fundamental, em um sistema síncrono, uma mensagem enviada por um processador para outro é recebida pelo destinatário na mesma rodada de comunicação em que foi enviada.
* **Assincronismo**: em sistemas assíncronos, não há limite superior para os atrasos de comunicação e de processamento. Em outras palavras, é impossível definir um limite superior para os atrasos de comunicação e de processamento em sistemas assíncronos. Algoritmos assíncronos são projetados para rodar em redes assíncronas sem qualquer suposição de temporização. Esses sistemas são caracterizados pela imprevisibilidade dos atrasos de transferência de mensagens (comunicação) e dos atrasos de processamento. Esse cenário é comum em sistemas distribuídos de larga escala geograficamente dispersos e em sistemas onde a carga de entrada é imprevisível.
* **Sincronismo parcial**: neste modelo, há um limite superior para os atrasos de comunicação e de processamento, no entanto, esse limite superior não é conhecido pelos processadores. Um sistema eventualmente síncrono é um tipo de sincronismo parcial, o que significa que o sistema se torna síncrono após um instante de tempo chamado tempo de estabilização global (GST – Global Stabilization Time). O GST não é conhecido pelos processadores. Em geral, o sincronismo parcial captura o fato de que, geralmente, os sistemas são síncronos, mas há períodos assíncronos arbitrários, porém limitados. Além disso, o sistema em algum momento se torna síncrono por tempo suficiente para que os processadores possam decidir (chegar a um acordo) e terminar durante esse período.

Agora que compreendemos os fundamentos da teoria de consenso distribuído, vejamos duas classes principais de algoritmos de consenso. Essa categorização surgiu após a invenção do Bitcoin. Antes do Bitcoin, havia uma longa história de pesquisa em protocolos de consenso distribuído.

**Classificação**

Os algoritmos de consenso podem ser classificados em duas categorias amplas:

* **Tradicional**: consenso baseado em votação. O consenso tradicional baseado em votação tem sido pesquisado em sistemas distribuídos por muitas décadas. Muitos resultados fundamentais e uma grande quantidade de trabalhos inovadores já foram produzidos nesse campo. Algoritmos como Paxos e PBFT são exemplos principais desse tipo de algoritmo. O consenso tradicional também pode ser chamado de consenso distribuído tolerante a falhas.
* **Baseado em sorteio**: consenso de Nakamoto e pós-Nakamoto. O consenso baseado em sorteio ou do tipo Nakamoto foi introduzido pela primeira vez com o Bitcoin. Essa classe também pode ser chamada simplesmente de consenso em blockchain.

Observe que eles são distinguidos pelo período em que foram inventados; os protocolos tradicionais foram desenvolvidos antes da introdução do Bitcoin, e os protocolos baseados em sorteio foram introduzidos com o Bitcoin.

Deseja que eu continue com a próxima parte da tradução?

Os requisitos fundamentais dos algoritmos de consenso se resumem às condições de **Segurança** e **Vivacidade**.  
O requisito de **Segurança** geralmente significa que nada de ruim acontece, e geralmente se baseia em alguns requisitos de segurança dos algoritmos, como acordo, validade e integridade. Normalmente, há três propriedades dentro dessa classe de requisitos, que estão listadas a seguir:

* **Acordo (Agreement)**: A propriedade de acordo exige que dois processos não decidam sobre valores diferentes.
* **Validade (Validity)**: A validade afirma que, se um processo decidiu um valor, esse valor deve ter sido proposto por um processo. Em outras palavras, o valor decidido sempre é proposto por um processo honesto e não foi criado do nada.
* **Integridade (Integrity)**: Um processo deve decidir apenas uma vez.

O requisito de **Vivacidade** geralmente significa que algo bom eventualmente acontece. Isso significa que o protocolo pode progredir mesmo que as condições da rede não sejam ideais. Normalmente, há um único requisito:

* **Terminação (Termination)**: Esta propriedade de vivacidade afirma que cada nó honesto deve eventualmente decidir um valor.

Com isso, cobrimos a classificação e os requisitos dos algoritmos de consenso. Na próxima seção, apresentaremos vários algoritmos de consenso que podemos avaliar usando os modelos de consenso abordados nesta seção.

**Algoritmos**

Nesta seção, discutiremos os principais algoritmos em detalhes. Veremos os dois principais tipos de algoritmos tolerantes a falhas, que são classificados com base no nível de tolerância a falhas que fornecem: **CFT** ou **BFT**.

**Algoritmos CFT**

Começaremos examinando alguns algoritmos que resolvem o problema de consenso com tolerância a falhas por travamento. Um dos algoritmos mais fundamentais nesse espaço é o **Paxos**.

**Paxos**

Paxos foi desenvolvido por Leslie Lamport. É o algoritmo de consenso distribuído mais fundamental, permitindo consenso sobre um valor sob comunicações não confiáveis. Em outras palavras, o Paxos é usado para construir um sistema confiável que funcione corretamente, mesmo na presença de falhas. Muitos outros protocolos surgiram desde então a partir do Paxos básico, como o Multi-Paxos, o Fast Paxos e o Cheap Paxos.

Paxos foi proposto pela primeira vez em 1989 e, mais tarde, mais formalmente, em 1998, no seguinte artigo:

Lamport, L., 1998. “The part-time parliament”, ACM Transactions on Computer Systems (TOCS), 16(2): pp. 133-169.  
O artigo está disponível aqui: <https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf>

Paxos funciona sob um modelo de rede assíncrona e suporta apenas o tratamento de falhas benignas. Este **não** é um protocolo tolerante a falhas bizantinas. No entanto, posteriormente, foi desenvolvida uma variante do Paxos que fornece BFT.

O Paxos faz uso de **2f + 1 processos** para garantir tolerância a falhas em uma rede onde os processos podem falhar por travamento, ou seja, apresentar falhas benignas. Falha benigna significa perda de mensagem ou parada de um processo. Em outras palavras, o Paxos pode tolerar uma falha por travamento em uma rede de três nós.

O Paxos é um protocolo de duas fases. A primeira fase é chamada de fase de preparação (*prepare phase*), e a fase seguinte é chamada de fase de aceitação (*accept phase*). O Paxos tem um proponente (*proposer*) e aceitadores (*acceptors*) como participantes, onde o proponente são as réplicas ou nós que propõem os valores, e os aceitadores são os nós que aceitam os valores.

O protocolo Paxos assume uma rede de troca de mensagens assíncrona com menos de 50% de falhas por travamento. Como de costume, as propriedades críticas do algoritmo de consenso Paxos são segurança e vivacidade. Sob segurança, temos:

* **Acordo** (*Agreement*), que especifica que dois valores diferentes não são acordados.
* **Validade** (*Validity*), que significa que apenas os valores propostos são decididos.

Sob vivacidade, temos:

* **Terminação** (*Termination*), que significa que, eventualmente, o protocolo é capaz de decidir e encerrar.

Os processos podem assumir diferentes papéis, listados a seguir:

* **Proponentes** são os líderes eleitos que podem propor um novo valor a ser decidido.
* **Aceitadores** participam do protocolo como um meio de fornecer uma decisão por maioria.
* **Aprendizes** (*learners*) são nós que apenas observam o processo de decisão e o valor.

Um único processo em uma rede Paxos pode assumir todos os três papéis.

A ideia principal por trás do Paxos é que o nó proponente propõe um valor, que é considerado final apenas se a maioria dos nós aceitadores o aceitar. Os nós aprendizes também aprendem essa decisão final.

Paxos pode ser visto como um protocolo bastante semelhante ao protocolo de confirmação em duas fases (*two-phase commit*). A confirmação em duas fases (2PC) é um protocolo padrão de compromisso atômico para garantir que transações sejam confirmadas em bancos de dados distribuídos apenas se todos os participantes concordarem em confirmar. Em contraste com a confirmação em duas fases, o Paxos introduziu a ordenação (sequenciamento para alcançar ordem total) das propostas e aceitação por maioria das propostas em vez de esperar que todos os nós concordem (para permitir progresso mesmo que alguns nós falhem). Ambas as melhorias contribuem para garantir a segurança e a vivacidade do algoritmo Paxos.

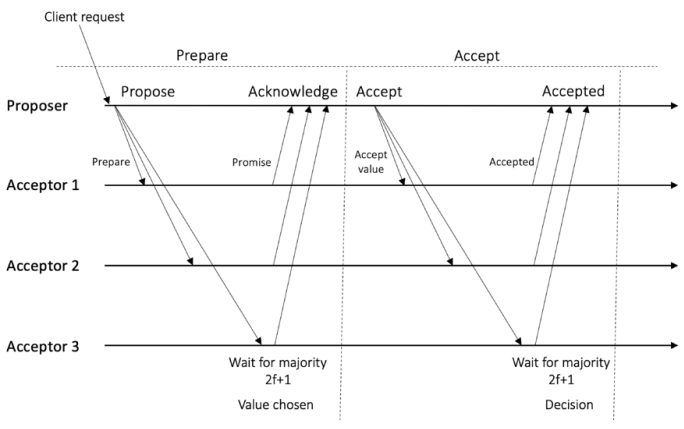
O artigo em que uma versão tolerante a falhas bizantinas do Paxos é descrita está disponível aqui:

Lamport, L., 2011, setembro. *Byzantizing Paxos by refinement*. International Symposium on Distributed Computing (pp. 211–224). Springer, Berlin, Heidelberg.  
Link para o artigo: <http://lamport.azurewebsites.net/pubs/web-byzpaxos.pdf>

Agora descreveremos como o protocolo Paxos funciona, passo a passo:

1. O proponente propõe um valor enviando uma mensagem de difusão, <prepare(n)>, para todos os aceitadores.
2. Os aceitadores respondem com uma mensagem de reconhecimento se a proposta *n* for a mais alta à qual o aceitador respondeu até o momento. A mensagem de reconhecimento <ack(n, v, s)> consiste em três variáveis, onde *n* é o número da proposta, *v* é o valor da proposta de número mais alto que o aceitador aceitou até então, e *s* é o número de sequência da proposta mais alta aceita até o momento. É nesse ponto que os aceitadores concordam em assumir o compromisso com o valor proposto. O proponente agora aguarda o recebimento de mensagens de reconhecimento da maioria dos aceitadores indicando o valor escolhido.
3. Se a maioria for recebida, o proponente envia a mensagem de “aceitação” <accept(n, v)> para os aceitadores.
4. Se a maioria dos aceitadores aceitar o valor proposto (agora na forma da mensagem “accept”), então ele é decidido: ou seja, o acordo é alcançado.
5. Finalmente, na fase de aprendizagem (*learning phase*), os aceitadores difundem a mensagem “accepted” <accepted(n, v)> para o proponente. Essa fase é necessária para disseminar qual proposta foi finalmente aceita. O proponente então informa todos os outros aprendizes sobre o valor decidido. Alternativamente, os aprendizes podem aprender o valor decidido por meio de uma mensagem contendo o valor aceito (valor da decisão) enviado em multicast pelos aceitadores.

Podemos visualizar esse processo no seguinte diagrama:



**Figura 5.1: Como o Paxos funciona**

Uma pergunta natural surge sobre como o Paxos garante suas garantias de segurança e vivacidade. Paxos, em sua essência, é bastante simples, mas ainda assim atinge todas essas propriedades de forma eficiente. As provas reais da correção do Paxos são bastante profundas e não são o foco deste capítulo. No entanto, a justificativa por trás de cada propriedade é apresentada da seguinte forma:

* **Acordo** é garantido ao impor que apenas uma proposta pode ganhar votos da maioria dos aceitadores.
* **Validade** é garantida ao impor que apenas propostas genuínas sejam decididas. Em outras palavras, nenhum valor é comprometido a menos que tenha sido proposto primeiro em uma mensagem de proposta.
* **Vivacidade**, ou **terminação**, é garantida ao assegurar que, em algum momento durante a execução do protocolo, eventualmente haverá um período durante o qual haverá apenas um proponente sem falhas.

Embora o algoritmo Paxos seja bastante simples em sua essência, ele é considerado difícil de entender, e muitos artigos acadêmicos foram escritos para explicá-lo. Esse pequeno problema, no entanto, não impediu que ele fosse implementado em muitas redes de produção, como o *Spanner* do Google, já que provou ser o protocolo mais eficiente para resolver o problema de consenso. Ainda assim, houve tentativas de criar algoritmos alternativos mais fáceis de entender. **Raft** é uma dessas tentativas de criar um algoritmo CFT fácil de compreender.

**Raft**

O protocolo Raft é um mecanismo de consenso com tolerância a falhas por travamento (CFT) desenvolvido por Diego Ongaro e John Ousterhout na Universidade de Stanford. No Raft, assume-se sempre que o líder é honesto.

Em nível conceitual, trata-se de um **log replicado para uma máquina de estados replicada (Replicated State Machine – RSM)**, onde um líder único é eleito a cada “termo” (divisão de tempo), cujo log é replicado para todos os nós seguidores (*follower nodes*).

Raft é composto por três subproblemas:

* **Eleição de líder** (uma nova eleição de líder caso o atual falhe)
* **Replicação de log** (sincronização do log entre líder e seguidores)
* **Segurança** (nenhuma entrada de log conflitante (índice) entre servidores)

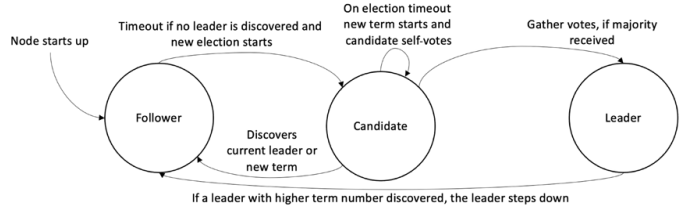
O protocolo Raft assegura:

* **Segurança da eleição** (apenas um vencedor em cada termo de eleição)
* **Apenas adição ao log do líder**
* **Correspondência de logs**
* **Completude do líder**
* **Vivacidade** (algum candidato eventualmente vence)
* **Segurança da máquina de estados**

Cada servidor no Raft pode estar em um dos três estados: **seguidor**, **líder** ou **candidato**. Em seu nível fundamental, o protocolo é bastante simples e pode ser descrito pela sequência seguinte:

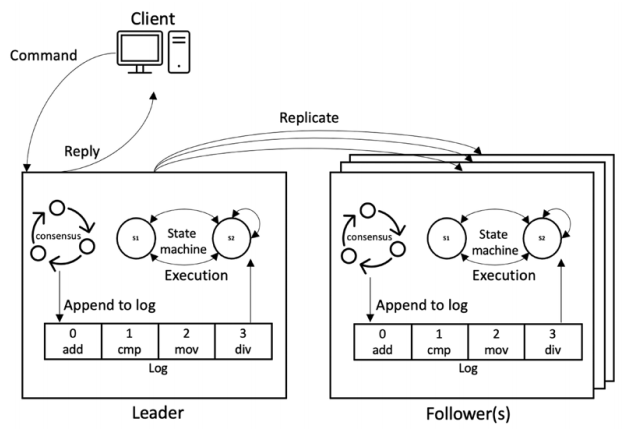
1. Primeiro, o nó inicia.
2. Depois disso, o processo de eleição de líder começa. Uma vez que um nó é eleito como líder, todas as mudanças passam por esse líder.
3. Cada alteração é registrada no log do nó.
4. A entrada de log permanece não comprometida até que seja replicada nos nós seguidores e o líder receba confirmações de gravação da maioria dos nós; então, ela é confirmada localmente.
5. O líder notifica os seguidores a respeito da entrada confirmada.
6. Uma vez que esse processo termina, o acordo é alcançado.

A transição de estado do algoritmo Raft pode ser visualizada no seguinte diagrama:



**Figura 5.2: Transição de estados do Raft**

Vimos anteriormente que os dados são eventualmente replicados por todos os nós em um mecanismo de consenso. No Raft, o log (dados) é eventualmente replicado por todos os nós. A lógica de replicação pode ser visualizada no seguinte diagrama. O objetivo da replicação de logs é sincronizar os nós entre si:



**Figura 5.3: Mecanismo de replicação de log**

A replicação de log é um mecanismo simples. Como mostrado no diagrama anterior, o líder é responsável pela replicação do log. Uma vez que o líder possui uma nova entrada em seu log, ele envia as requisições de replicação para os nós seguidores. Quando o líder recebe confirmações suficientes dos nós seguidores indicando que o pedido de replicação foi aceito e processado, o líder confirma essa entrada em sua máquina de estados local. Nesse estágio, a entrada é considerada como **confirmada**.

Com isso, nossa discussão sobre algoritmos **CFT** está concluída. Agora apresentaremos os algoritmos **tolerantes a falhas bizantinas (BFT)**, que têm sido objeto de pesquisa por muitos anos em computação distribuída.

**Algoritmos BFT**

Nesta seção, apresentaremos os mecanismos que foram desenvolvidos para resolver o problema dos generais bizantinos (consenso na presença de falhas).

**Tolerância a Falhas Bizantinas Prática (PBFT)**

O **PBFT** (*Practical Byzantine Fault Tolerance*) foi desenvolvido em 1999 por Miguel Castro e Barbara Liskov. PBFT, como o nome sugere, é um protocolo desenvolvido para fornecer consenso na presença de falhas bizantinas. Esse algoritmo demonstrou, pela primeira vez, que PBFT era possível.

O PBFT é composto por três subprotocolos:

* **Subprotocolo de operação normal** refere-se a um esquema que é executado quando tudo está funcionando normalmente e não há erros no sistema.
* **Mudança de visão (view change)** é um subprotocolo que é executado quando um nó líder defeituoso é detectado no sistema.
* **Checkpointing** é outro subprotocolo, que é usado para descartar dados antigos do sistema.

O protocolo é executado em rodadas nas quais, em cada rodada, um líder eleito, ou nó primário, lida com a comunicação com o cliente. Em cada rodada, o protocolo progride por três fases ou etapas. Essas fases são: **pré-preparação** (*pre-prepare*), **preparação** (*prepare*) e **comprometimento** (*commit*).

Os participantes no protocolo PBFT são chamados de **réplicas**, onde uma das réplicas se torna o **primário** como líder em cada rodada, e os demais nós atuam como **cópias de segurança** (*backups*). Cada réplica mantém um estado local composto por três elementos principais: um estado de serviço, um log de mensagens e um número representando a visão atual da réplica.

O PBFT baseia-se no protocolo SMR (replicação de máquina de estados), apresentado anteriormente. Aqui, cada nó mantém um log local, e os logs são mantidos sincronizados entre si por meio do protocolo de consenso: ou seja, o PBFT.

Como vimos anteriormente, para tolerar falhas bizantinas, o número mínimo de nós exigido é **n = 3f + 1**, onde *n* é o número de nós e *f* é o número de nós com falhas. O PBFT garante a tolerância a falhas bizantinas desde que o número de nós no sistema obedeça a **n ≥ 3f + 1**.

Agora discutiremos as fases mencionadas acima, uma a uma, começando com a **pré-preparação**.

**Pré-preparação** (*Pre-prepare*): o principal objetivo desta fase é atribuir um número de sequência único à requisição. Podemos pensar nela como um **ordenador**. Esta é a primeira fase no protocolo, onde o nó primário (líder) recebe (aceita) uma requisição do cliente. O nó primário atribui um número de sequência à requisição. Em seguida, ele envia a mensagem *pre-prepare* com a requisição para todas as réplicas de backup.

Quando a mensagem *pre-prepare* é recebida pelas réplicas de backup, elas verificam vários aspectos para garantir a validade da mensagem:

1. Primeiro, se a assinatura digital é válida.
2. Em seguida, se o número da visão atual é válido.
3. Depois, se o número de sequência da mensagem da requisição da operação é válido.
4. Finalmente, se o hash (ou *digest*) da mensagem da requisição da operação é válido.

Se todos esses elementos forem válidos, então a réplica de backup aceita a mensagem. Após aceitar a mensagem, ela atualiza seu estado local e avança para a fase de **preparação**.

**Preparação** (*Prepare*):  
Esta fase garante que as réplicas/nós honestos na rede concordem com a ordem total das requisições dentro de uma visão. Observe que as fases de *pre-prepare* e *prepare* juntas fornecem a ordenação total das mensagens. Uma mensagem *prepare* é enviada por cada réplica de backup a todas as outras réplicas no sistema. Cada réplica de backup espera receber pelo menos **2f + 1 mensagens de preparação** (*prepare*) de outras réplicas. Elas também verificam se a mensagem *prepare* contém o mesmo número de visão, número de sequência e valores do hash da mensagem. Se todos esses testes forem aprovados, então a réplica atualiza seu estado local e avança para a fase de **comprometimento** (*commit*).

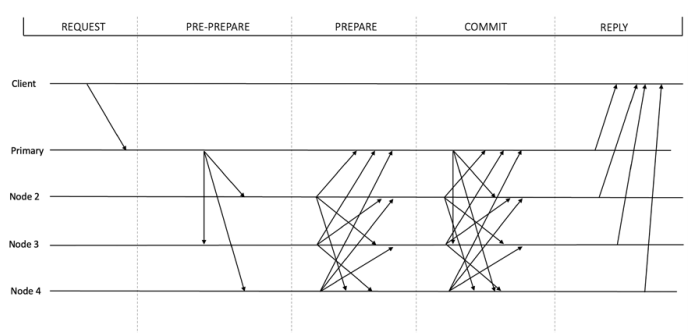
**Comprometimento** (*Commit*):  
Esta fase garante que as réplicas/nós honestos na rede concordem com a ordem total das requisições entre visões. Na fase *commit*, cada réplica envia uma mensagem de *commit* para todas as outras réplicas da rede. Assim como na fase *prepare*, as réplicas aguardam pelo recebimento de **2f + 1 mensagens de *commit*** de outras réplicas. As réplicas também verificam o número da visão, número de sequência e valores do hash da mensagem. Se forem válidos para **2f + 1 mensagens de commit** recebidas de outras réplicas, então a réplica executa a requisição, produz um resultado e finalmente atualiza seu estado para refletir o comprometimento. Se já houver mensagens em fila, a réplica executará essas requisições primeiro antes de processar os últimos números de sequência. Por fim, a réplica envia o resultado ao cliente em uma mensagem de resposta (*reply*). O cliente aceita o resultado apenas após receber **2f + 1 mensagens de resposta contendo o mesmo resultado**.

**Em resumo, o protocolo funciona da seguinte forma:**

1. Um cliente envia uma requisição para invocar uma operação de serviço ao nó primário.
2. O primário realiza *multicast* da requisição para os *backups*.
3. As réplicas executam a requisição e enviam uma resposta ao cliente.
4. O cliente aguarda respostas de diferentes réplicas com o mesmo resultado; este é o resultado da operação.

O principal objetivo dessas fases é alcançar o consenso, onde cada fase é responsável por uma parte crítica do mecanismo de consenso que, após passar por todas elas, resulta na obtenção de um acordo final.

A visão normal do protocolo PBFT pode ser visualizada da seguinte forma:



**Figura 5.4: Protocolo PBFT**

Na execução do protocolo, a integridade das mensagens e das operações do protocolo deve ser mantida para fornecer um nível adequado de segurança e garantia. Isso é mantido por meio do uso de **assinaturas digitais**. Além disso, **certificados** são usados para assegurar a maioria adequada de participantes (nós).

**Certificados**

Os **certificados** nos protocolos PBFT são usados para demonstrar que pelo menos **2f + 1 nós** armazenaram as informações necessárias. Por exemplo, se um nó coletou **2f + 1 mensagens de preparação**, então, ao combiná-las com a respectiva mensagem de *pre-prepare* contendo a mesma visão, sequência e requisição, obtém-se um certificado, chamado de **certificado de preparação** (*prepared certificate*). De modo semelhante, uma coleção de **2f + 1 mensagens de commit** é chamada de **certificado de comprometimento** (*commit certificate*).

Há também várias **variáveis** que o protocolo PBFT mantém para executar o algoritmo. Essas variáveis e seus significados estão listados a seguir:

| **Variável de estado** | **Explicação** |
| --- | --- |
| **v** | Número da visão |
| **m** | Última mensagem de requisição |
| **n** | Número de sequência da mensagem |
| **h** | Hash da mensagem |
| **i** | Número de índice |
| **C** | Conjunto de todos os checkpoints |

**Não confunda esses certificados com certificados digitais** comumente usados em infraestrutura de chave pública (PKI), ou em websites e infraestruturas de TI para proteger ativos como servidores.

| **Variável de estado** | **Explicação** |
| --- | --- |
| **P** | Conjunto de todas as mensagens *pre-prepare* e *prepare* correspondentes |
| **O** | Conjunto de mensagens *pre-prepare* sem as mensagens de requisição correspondentes |

Agora podemos examinar os tipos de mensagens e seus formatos, o que se torna bastante fácil de entender se fizermos referência à tabela de variáveis acima.

**Tipos de mensagens**

O protocolo PBFT funciona por meio da troca de várias mensagens. Uma lista desses tipos de mensagem é apresentada a seguir com seus formatos e direções:

| **Mensagem** | **De** | **Para** | **Formato** | **Assinada por** |
| --- | --- | --- | --- | --- |
| **Request** | Cliente | Primário | <REQUEST, m> | Cliente |
| **Pre-prepare** | Primário | Backups | <PRE-PREPARE, v, n, h> | Cliente |
| **Prepare** | Réplica | Réplicas | <PREPARE, v, n, h, i> | Réplica |
| **Commit** | Réplica | Réplicas | <COMMIT, v, n, h, i> | Réplica |
| **Reply** | Réplica | Cliente | <REPLY, r, i> | Réplica |
| **View change** | Réplica | Réplicas | <VIEWCHANGE, v+1, n, C, P, i> | Réplica |
| **New view** | Réplica primária | Réplicas | <NEWVIEW, v+1, v, O> | Réplica |
| **Checkpoint** | Réplica | Réplicas | <CHECKPOINT, n, h, i> | Réplica |

Note que todas essas mensagens são assinadas.  
Vamos examinar alguns tipos específicos de mensagens trocadas durante o protocolo PBFT.

**Mudança de visão (*View change*)**

A **mudança de visão** ocorre quando um primário é suspeito de estar com falha. Essa fase é necessária para garantir o progresso do protocolo. Com o subprotocolo de *view change*, um novo primário é selecionado, o qual então inicia novamente a operação no modo normal. O novo primário é selecionado em esquema de **rotação (round-robin)**.

Quando uma réplica de backup recebe uma requisição, ela tenta executá-la após validar a mensagem, mas, por algum motivo, se não conseguir executá-la por um tempo, a réplica entra em **timeout** e inicia o subprotocolo de *view change*.

No protocolo de mudança de visão, a réplica para de aceitar mensagens relacionadas à visão atual e atualiza seu estado para **VIEW-CHANGE**. As únicas mensagens que ela pode receber nesse estado são mensagens de **CHECKPOINT**, **VIEW-CHANGE** e **NEW-VIEW**. Depois disso, ela envia uma mensagem *VIEW-CHANGE* com o próximo número de visão para todas as réplicas.

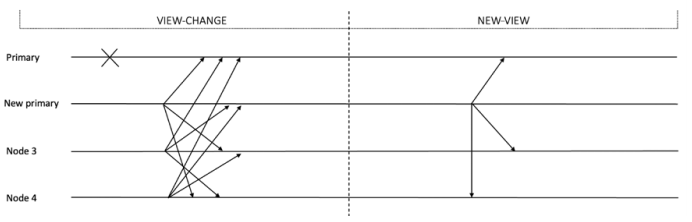
Quando essa mensagem chega ao novo primário, o primário espera por pelo menos **2f mensagens de VIEW-CHANGE** referentes à nova visão. Se receber ao menos 2f dessas mensagens, ele **difunde uma mensagem NEW-VIEW** para todas as réplicas e passa a operar novamente no modo normal.

Quando outras réplicas recebem uma mensagem *NEW-VIEW*, elas atualizam seus estados locais de acordo e iniciam novamente a operação no modo normal.

O algoritmo para o subprotocolo de mudança de visão é mostrado a seguir:

1. Parar de aceitar mensagens *pre-prepare*, *prepare* e *commit* da visão atual.
2. Criar um conjunto de todos os certificados preparados até o momento.
3. Difundir uma mensagem *VIEW-CHANGE* com o próximo número de visão e o conjunto de todos os certificados preparados para todas as réplicas.

O subprotocolo de mudança de visão pode ser visualizado no seguinte diagrama:



**Figura 5.5: Subprotocolo de mudança de visão**

O subprotocolo de mudança de visão é um mecanismo para alcançar **vivacidade**. Três técnicas inteligentes são usadas nesse subprotocolo para garantir que, eventualmente, haverá um momento em que a operação solicitada será executada:

1. Uma réplica que difundiu a mensagem *VIEW-CHANGE* espera por **2f + 1 mensagens VIEW-CHANGE** e então inicia seu temporizador. Se o temporizador expirar antes que o nó receba uma mensagem *NEW-VIEW* para a nova visão, o nó iniciará a mudança de visão para a próxima sequência, mas aumentará seu valor de tempo limite. Isso também ocorre se a réplica expirar antes de executar a nova requisição única na nova visão.
2. Assim que a réplica recebe **f + 1 mensagens VIEW-CHANGE** para um número de visão maior que sua visão atual, ela enviará a mensagem *VIEW-CHANGE* para a **menor visão que conhece** no conjunto, para que a próxima mudança de visão não ocorra tarde demais. Isso acontece mesmo se o temporizador ainda não tiver expirado; ela enviará a mudança de visão para a menor visão conhecida.
3. Como a mudança de visão só ocorrerá se pelo menos **f + 1 réplicas** tiverem enviado a mensagem *VIEW-CHANGE*, esse mecanismo garante que um primário defeituoso **não possa impedir indefinidamente o progresso** solicitando sucessivas mudanças de visão.

**Checkpointing**

**Checkpointing** é outro subprotocolo crucial. Ele é usado para **descartar mensagens antigas** no log de todas as réplicas. Com isso, as réplicas concordam com um **checkpoint estável**, que fornece um **instantâneo do estado global** em um determinado momento. Esse é um processo periódico realizado por cada réplica após executar a requisição e marcá-la como um checkpoint em seu log.

Uma variável chamada **low watermark** (em terminologia PBFT) é usada para registrar o número de sequência do último checkpoint estável. Esse checkpoint é então difundido para os outros nós. Assim que uma réplica tem pelo menos **2f + 1 mensagens de checkpoint**, ela salva essas mensagens como prova de um checkpoint estável. Em seguida, **descarta todas as mensagens anteriores de *pre-prepare*, *prepare* e *commit*** de seus logs.

PBFT é de fato um protocolo revolucionário que abriu um novo campo de pesquisa em protocolos PBFT. O PBFT original possui muitas **forças**, mas também algumas **limitações**. Discutimos a maioria das forças e limitações mais frequentemente citadas na tabela a seguir:

| **Forças** | **Limitações** |
| --- | --- |
| PBFT fornece **finalidade imediata e determinística** para transações. Isso contrasta com o PoW, onde várias confirmações são necessárias para finalizar uma transação com alta probabilidade. | A **escalabilidade de nós é bastante baixa**. Geralmente, redes pequenas com poucos nós (dezenas) são adequadas para uso com PBFT devido à sua alta complexidade de comunicação. Por isso, é mais apropriado para redes consorciadas do que blockchains públicas. |
| PBFT é **energeticamente eficiente** em comparação ao PoW, que consome uma enorme quantidade de eletricidade. | Ataques **Sybil** podem ser realizados em uma rede PBFT, onde uma única entidade pode controlar muitas identidades para influenciar a votação e, subsequentemente, a decisão. No entanto, a correção é trivial, e de fato, esse ataque não é muito prático em redes consorciadas, pois todas as identidades são conhecidas na rede. Esse problema pode ser resolvido simplesmente aumentando o número de nós na rede. |

No modelo tradicional cliente-servidor, o PBFT funciona bem; no entanto, no caso do blockchain, implementar diretamente o PBFT em seu estado original pode não funcionar corretamente. Isso ocorre porque o design original do PBFT **não foi desenvolvido para blockchain**. Essa constatação levou ao desenvolvimento do **IBFT** e à implementação do PBFT no **Hyperledger Sawtooth**, **Hyperledger Fabric** e outras blockchains. Em todos esses cenários, algumas **modificações** foram feitas no protocolo central para garantir que ele seja compatível com o ambiente blockchain.

**Tolerância a Falhas Bizantinas de Istambul (IBFT)**

O **IBFT** foi desenvolvido pela AMIS Technologies como uma **variante do PBFT** adequada para redes blockchain. Ele foi apresentado na **EIP 650** para a blockchain do Ethereum.

Vamos primeiro discutir as principais **diferenças** entre os protocolos PBFT e IBFT. Elas são as seguintes:

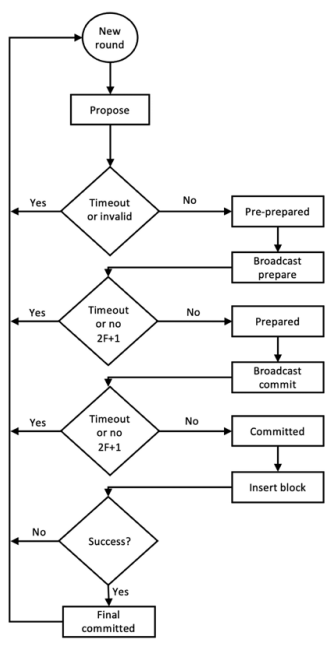
* No **IBFT**, a **participação dos validadores pode ser alterada dinamicamente**, e validadores podem ser votados para dentro ou para fora conforme necessário. Isso **contrasta com o PBFT original**, onde os nós validadores são estáticos.
* Existem **dois tipos de nós** em uma rede IBFT: **nós** e **validadores**. Os nós se sincronizam com o blockchain **sem participar do processo de consenso IBFT**.
* O IBFT baseia-se em uma estrutura **mais simples de mensagens de mudança de visão (mudança de rodada)** em comparação com o PBFT.
* **Ao contrário do PBFT**, no IBFT **não há um conceito concreto de checkpoints**. No entanto, **cada bloco pode ser considerado como um indicador do progresso até o momento** (a altura da cadeia – *chain height*).
* **Não há coleta de lixo** (*garbage collection*) no IBFT.

Agora que cobrimos os principais pontos de comparação entre os dois algoritmos BFT, examinaremos como o protocolo IBFT funciona e suas várias fases.

O IBFT assume um **modelo de rede** sob o qual deve operar. Esse modelo é composto por:

* Pelo menos **3f + 1 processos** (a suposição padrão de BFT)
* Uma rede de troca de mensagens **parcialmente síncrona**
* **Criptografia segura**

O protocolo é executado em **rodadas**. Ele possui **três fases**: *pre-prepare*, *prepare* e *commit*. Em cada rodada, geralmente um novo líder é eleito com base em um mecanismo de **rotação (round-robin)**. O fluxograma a seguir mostra como o protocolo IBFT funciona:



**Figura 5.6: Fluxograma do IBFT**

Vamos discutir esse processo passo a passo, conforme segue:

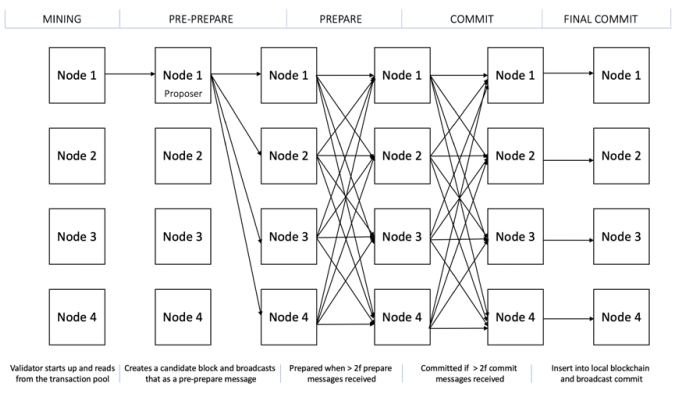
1. O protocolo começa com uma **nova rodada**. Nessa nova rodada, o proponente selecionado envia uma proposta (bloco) como uma mensagem *pre-prepare*.
2. Os nós que recebem essa mensagem *pre-prepare* a validam e a aceitam se ela for uma mensagem válida. Os nós então definem seu estado como **pre-preparado**.
3. Neste ponto, se ocorrer um **timeout**, ou se a proposta for considerada **inválida** pelos nós, eles iniciarão uma **mudança de rodada** (*round change*). O processo normal então começa novamente com um proponente propondo um bloco.
4. Os nós então difundem a mensagem *prepare* e aguardam por **2f + 1 mensagens *prepare*** vindas de outros nós. Se os nós não receberem 2f + 1 mensagens a tempo, eles entram em timeout, e o processo de mudança de rodada começa. Os nós definem seu estado como **preparado** após receberem as 2f + 1 mensagens de outros nós.
5. Finalmente, os nós enviam uma mensagem *commit* e aguardam **2f + 1 mensagens *commit*** de outros nós. Se elas forem recebidas, o estado é definido como **comprometido** (*committed*); caso contrário, ocorre timeout, e o processo de mudança de rodada é iniciado.
6. Uma vez **comprometido**, tenta-se a inserção do bloco. Se for bem-sucedida, o protocolo avança para o estado final de comprometido, e eventualmente uma nova rodada começa. Se a inserção falhar por algum motivo, o processo de mudança de rodada é acionado. Novamente, os nós aguardam por **2f + 1 mensagens de mudança de rodada**, e se o limiar for atingido, então a mudança de rodada ocorre.

**Estados de consenso**

O **IBFT** é um algoritmo de **replicação de máquina de estados (SMR)**. Cada validador mantém uma **réplica da máquina de estados** para alcançar consenso sobre o bloco, isto é, **chegar a um acordo**. Esses estados estão listados a seguir, com suas respectivas explicações:

* **Nova rodada (New round)**: Neste estado, uma nova rodada do mecanismo de consenso começa, e o proponente selecionado envia uma nova proposta de bloco para os outros validadores. Nesse estado, todos os outros validadores aguardam a mensagem *PRE-PREPARE*.
* **Pré-preparado (Pre-prepared)**: Um validador transita para este estado quando recebeu uma mensagem *PRE-PREPARE* e difunde uma mensagem *PREPARE* para os demais validadores. O validador então aguarda **2f + 1 mensagens *PREPARE* ou *COMMIT***.
* **Preparado (Prepared)**: Este estado é atingido por um validador quando ele recebeu **2f + 1 mensagens *PREPARE*** e difundiu a mensagem *COMMIT*. O validador então aguarda **2f + 1 mensagens *COMMIT*** de outros validadores.
* **Comprometido (Committed)**: Este estado indica que um validador recebeu **2f + 1 mensagens *COMMIT***. O validador, neste estágio, pode inserir o bloco proposto na blockchain.
* **Comprometido final (Final committed)**: Este estado é alcançado por um validador quando o novo bloco comprometido foi inserido com sucesso na blockchain. Neste estado, o validador também está pronto para a próxima rodada de consenso.
* **Mudança de rodada (Round change)**: Este estado indica que os validadores estão aguardando **2f + 1 mensagens de mudança de rodada** (*round change*) para o novo número de rodada proposto.

O protocolo IBFT pode ser visualizado em um diagrama, semelhante ao PBFT, conforme segue:



**Figura 5.7: IBFT**

Um mecanismo adicional que torna o IBFT bastante atrativo é seu **mecanismo de gerenciamento de validadores**. Por meio deste mecanismo, validadores podem ser **adicionados ou removidos** por meio de **votação entre os membros da rede**. Esta é uma funcionalidade bastante útil e fornece o nível certo de flexibilidade quando se trata de gerenciar validadores de forma eficiente, ao invés de adicioná-los ou removê-los manualmente do conjunto de validadores.

**Tendermint**

**Tendermint** é outra variante do PBFT. Ele é inspirado tanto nos protocolos **DLS** quanto no **PBFT**. O Tendermint também faz uso da abordagem de **replicação de máquina de estados (SMR)** para fornecer replicação tolerante a falhas. Como vimos antes, a replicação de máquina de estados é um mecanismo que permite a sincronização entre réplicas/nós da rede.

O IBFT foi implementado em diversas blockchains. Exemplos de implementação incluem **Quorum** e **Celo**.  
Uma implementação do Quorum está disponível no seguinte link: <https://github.com/ConsenSys/quorum>  
O IBFT, com algumas modificações, também foi implementado na blockchain **Celo**, disponível em: <https://github.com/celo-org/celo-blockchain>

Tradicionalmente, um mecanismo de consenso era utilizado com um pequeno número de participantes, e, portanto, desempenho e escalabilidade **não eram grandes preocupações**. No entanto, com o surgimento do blockchain, há a necessidade de desenvolver algoritmos que possam funcionar em **redes de larga escala** e em **ambientes assíncronos**. Pesquisas nessas áreas de computação distribuída não são novas, e especialmente agora, devido ao crescimento das criptomoedas e do blockchain, o interesse nesses temas aumentou significativamente nos últimos anos.

O protocolo Tendermint funciona executando **rodadas**. Em cada rodada, um líder é eleito, que propõe o próximo bloco.  
Observe também que no Tendermint, o processo de **mudança de rodada (ou visão)** **faz parte da operação normal**, ao contrário do PBFT, onde a mudança de visão ocorre apenas em caso de erros, ou seja, quando há suspeita de que o líder esteja com falha.

O Tendermint funciona de forma semelhante ao PBFT, onde são necessárias **três fases** para alcançar uma decisão. Uma vez que uma rodada é concluída, uma nova rodada começa com três fases e termina quando uma decisão é alcançada.

Uma inovação fundamental no Tendermint é o design de um novo **mecanismo de terminação**. Ao contrário de outros protocolos estilo PBFT, o Tendermint desenvolveu um mecanismo mais direto, semelhante à operação normal do PBFT.  
Em vez de possuir dois subprotocolos separados (modo normal e modo de recuperação para mudança de visão), o Tendermint **termina sem custo adicional de comunicação**.

Vimos anteriormente, na introdução, que cada modelo de consenso é estudado, desenvolvido e executado sob um **modelo de sistema**, com algumas suposições a respeito desse sistema. O Tendermint também é projetado com um modelo de sistema em mente. Agora definiremos e apresentaremos cada elemento desse modelo de sistema:

* **Processos**: Um processo é o participante fundamental do protocolo. Ele também é chamado de **réplica** (na literatura tradicional do PBFT), **nó**, ou simplesmente processo. Os processos podem ser **corretos (honestos)** ou **com falhas (bizantinos)**. Cada processo possui um **poder de voto**.  
  Também é importante notar que os processos **não estão necessariamente conectados diretamente**; eles precisam apenas estar conectados de forma **solta**, ou apenas com um subconjunto imediato de nós.  
  Os processos possuem um **temporizador local**, que usam para medir o tempo limite (timeout).
* **Modelo de rede**: O modelo de rede é um conjunto de processos que se comunicam trocando mensagens. Particularmente, o protocolo de **fofoca (gossip protocol)** é usado para a comunicação entre processos.  
  Assume-se a regra padrão de **n ≥ 3f + 1** (para tolerância a falhas bizantinas). Isso significa que o protocolo funciona corretamente enquanto o número de nós com falhas *f* for menor que um terço dos nós honestos. Em outras palavras, são necessários no mínimo **quatro nós** na rede para tolerar falhas bizantinas.

O protocolo **DLS** é um mecanismo de consenso nomeado com base em seus autores (**Dwork, Lynch e Stockmeyer**).  
Esse protocolo consiste em duas rodadas iniciais. Esse processo de rodadas com troca apropriada de mensagens garante que o **acordo seja eventualmente alcançado** sobre um valor proposto ou padrão.  
Enquanto o número de nós no sistema for maior que 3f, esse protocolo alcança consenso.

**Temporização**

A temporização no Tendermint segue o modelo de **sincronismo parcial**. Isso significa que o protocolo **não depende de limites estritos de tempo** em termos de comunicação ou execução. Em vez disso, assume-se que o sistema será eventualmente síncrono, ou seja, **haverá um ponto no tempo em que todas as mensagens serão entregues com um limite de tempo finito**.

Esse modelo é prático para sistemas reais, nos quais pode haver períodos de comportamento assíncrono, mas espera-se que, após algum tempo, o sistema retorne a um estado de comportamento previsível e estável.

**Segurança**

O Tendermint garante **segurança sob qualquer condição de rede** (mesmo assíncrona), desde que o número de nós maliciosos não ultrapasse **f < n/3**, sendo *n* o número total de validadores.  
Mais especificamente, ele garante:

* **Segurança**: se um nó decide um bloco, **nenhum outro bloco com a mesma altura pode ser decidido por outro nó**.
* **Validade**: qualquer bloco decidido **precisa ser proposto por um validador válido**.
* **Integridade**: um validador **só decide uma vez por altura**.
* **Terminação**: eventualmente, todos os validadores **honestos** decidem por um bloco para cada altura da blockchain.

**Criptografia**

O protocolo requer o uso de um **sistema de assinatura digital seguro**. Cada validador possui uma **chave pública conhecida** por todos os outros validadores, de modo que qualquer mensagem enviada por ele possa ser **verificada criptograficamente**.

Além disso, o Tendermint utiliza **hashes criptográficos** para garantir a integridade das mensagens trocadas e dos blocos propostos.

Agora que discutimos as suposições fundamentais do sistema, apresentaremos o **funcionamento interno do protocolo**. Como mencionado anteriormente, o protocolo opera em **rodadas**, e cada rodada contém **três etapas principais**:

1. **Propose**
2. **Prevote**
3. **Precommit**

**Funcionamento interno**

Conforme mencionado, cada **rodada** no protocolo Tendermint é composta por três etapas principais:

**1. Proposta (*Propose*)**

Nesta etapa, o **proponente da rodada** (também chamado de **líder** ou **proposer**) propõe um novo bloco, que deve ser aceito pelos demais validadores.

* O proponente selecionado envia uma mensagem do tipo <PROPOSAL, h, r, B, L> para todos os validadores, onde:
  + *h* é a **altura da blockchain**,
  + *r* é o **número da rodada**,
  + *B* é o **bloco proposto**,
  + *L* é o **bloqueio** (*lock*) anterior, se houver.
* Os validadores recebem essa proposta e decidem se ela é válida.  
  A proposta só é aceita se for válida e se vier de um proponente legítimo para aquela rodada.

Se não houver proposta válida recebida dentro do tempo limite, a etapa termina sem bloco proposto.

**2. Pré-voto (*Prevote*)**

Nesta etapa, todos os validadores **transmitem um voto sobre o bloco proposto**.  
O objetivo é **verificar a validade da proposta** e preparar o sistema para o comprometimento, caso haja consenso suficiente.

* Cada validador envia uma mensagem do tipo <PREVOTE, h, r, id(B)>, onde *id(B)* é o identificador (hash) do bloco proposto, ou um valor nulo se nenhuma proposta válida foi recebida.
* Se um validador já tiver **um bloqueio anterior** (lock) sobre um bloco *B*, ele deve votar novamente por *B*, mesmo que outro bloco tenha sido proposto.

Após o envio dos pré-votos, cada validador **aguarda os votos de todos os outros validadores**.  
Se houver pelo menos **2f + 1 votos para o mesmo bloco**, o validador **bloqueia esse bloco** (*lock on B*) e passa para a próxima etapa.

**3. Pré-comprometimento (*Precommit*)**

Agora, os validadores que **bloquearam um bloco B** enviam um voto de **precommit** para esse mesmo bloco.

* A mensagem enviada é do tipo <PRECOMMIT, h, r, id(B)>.
* Se um validador **não recebeu 2f + 1 votos prevote para o mesmo bloco**, ele envia um *PRECOMMIT* com valor **nulo**.
* Quando um validador recebe **2f + 1 mensagens PRECOMMIT com o mesmo id(B)**, ele **decide esse bloco como finalizado**, ou seja, o bloco é **aceito como parte da blockchain**.

Se a decisão for tomada, todos os validadores **avançam para a próxima altura da cadeia** (h := h + 1).  
Se não houver consenso suficiente em uma rodada (por exemplo, se não for alcançado o limite de 2f + 1), inicia-se uma **nova rodada (r := r + 1)** com um **novo proponente**.

O Tendermint também emprega um **sistema de timeout** para evitar bloqueios:

* Se um tempo limite expira em qualquer etapa (Propose, Prevote ou Precommit), os validadores **iniciam uma nova rodada**, garantindo **vivacidade** mesmo em condições de rede adversas.

Esse mecanismo de tempo limite, combinado com o **modelo de sincronismo parcial**, garante que, **eventualmente**, todas as transações válidas sejam incluídas na blockchain.

**Comparação entre algoritmos BFT e CFT**

A seguir, apresentamos uma **comparação de alto nível** entre os principais algoritmos de consenso que foram discutidos neste capítulo. Esta tabela serve para dar uma visão geral das diferenças entre os algoritmos **CFT (Crash Fault Tolerance)** e **BFT (Byzantine Fault Tolerance)**:

| **Algoritmo** | **Tipo** | **Modelo de rede** | **Tolerância a falhas** | **Rodadas** | **Complexidade de comunicação** | **Finalidade** |
| --- | --- | --- | --- | --- | --- | --- |
| Paxos | CFT | Assíncrono | f < n/2 | Várias | O(n²) | Probabilística |
| Raft | CFT | Assíncrono | f < n/2 | 1 | O(n) | Determinística |
| PBFT | BFT | Parcialmente síncrono | f < n/3 | 1 | O(n²) | Determinística |
| IBFT | BFT | Parcialmente síncrono | f < n/3 | 1 | O(n²) | Determinística |
| Tendermint | BFT | Parcialmente síncrono | f < n/3 | Múltiplas | O(n²) | Determinística |

Aqui, *n* é o número de nós, e *f* é o número de falhas toleradas.  
*Finalidade determinística* significa que, uma vez que um bloco é confirmado, ele **não pode ser revertido**.  
*Finalidade probabilística* significa que a probabilidade de um bloco ser revertido diminui com o tempo (como no caso do PoW).

**Escolhendo um algoritmo**

Agora que analisamos os principais algoritmos de consenso, surge a pergunta: **qual algoritmo devemos escolher**?

A resposta depende de **vários fatores**, incluindo:

* Se a rede é **permissiva (permissioned)** ou **sem permissão (permissionless)**
* Se a aplicação exige **finalidade rápida e determinística**
* Se o sistema precisa tolerar **falhas bizantinas** ou apenas falhas por travamento
* A **escala da rede** (pequena, média ou grande)
* A **tolerância à latência e ao throughput**
* O **consumo de energia** (como no caso de PoW)

Aqui está um resumo das recomendações:

* Para **redes permissionadas** com poucos nós e necessidade de **finalidade rápida e segurança forte**, **PBFT**, **IBFT** ou **Tendermint** são adequados.
* Para **redes públicas permissionless**, os algoritmos **PoW** (Proof of Work) ou **PoS** (Proof of Stake) ainda são os mais utilizados, devido à sua robustez e tolerância a participantes anônimos.
* Para aplicações com **baixa latência** e **alta disponibilidade**, o **Raft** é uma escolha popular.
* Para aplicações com **modelo de segurança simples**, o **Paxos** ainda é relevante.

**Resumo**

Neste capítulo, discutimos o papel central dos **algoritmos de consenso** em sistemas distribuídos e blockchains.  
Exploramos tanto os algoritmos **tradicionais** (como Paxos e Raft), quanto os algoritmos **bizantinos** (como PBFT, IBFT e Tendermint), que são mais apropriados para redes onde **participantes podem agir maliciosamente**.

Compreender os **modelos, propriedades, forças e limitações** desses algoritmos é essencial para projetar e avaliar sistemas distribuídos confiáveis e seguros, especialmente em aplicações **blockchain**.