**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.

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.

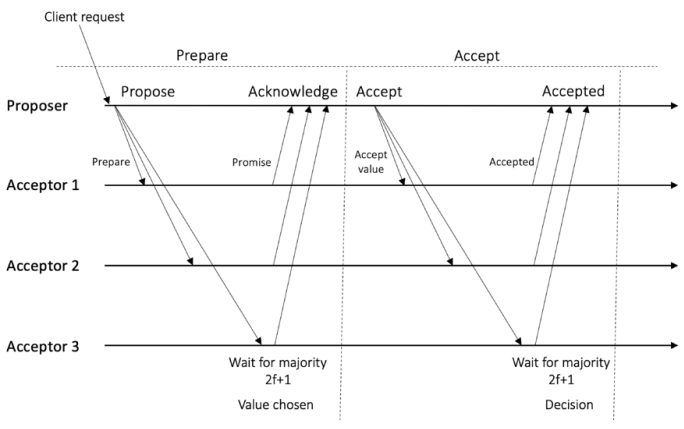
O artigo em que uma versão tolerante a falhas bizantinas do Paxos é descrita está disponível aqui:

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)

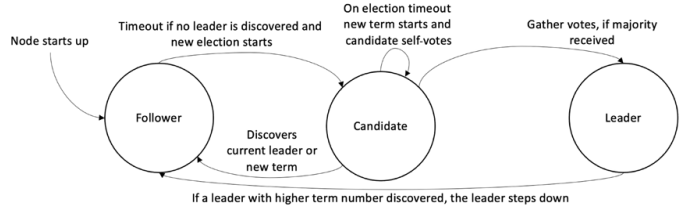
O protocolo Raft assegura:

* **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:

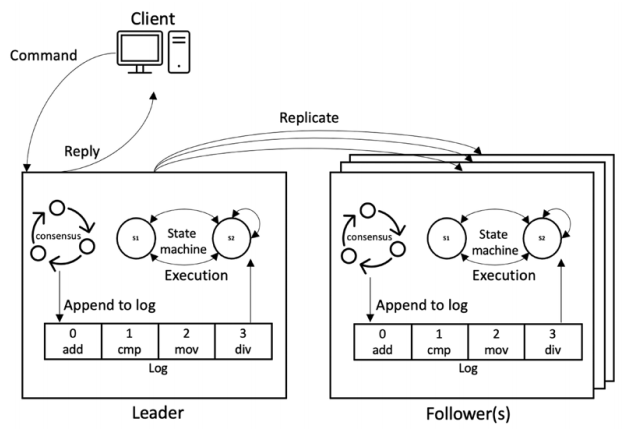
1. Primeiro, o nó inicia.
2. Depois disso, o processo de eleição de líder começa. Uma vez que um nó é eleito como líder, todas as mudanças passam por esse líder.
3. Cada alteração é registrada no log do nó.
4. A entrada de log permanece não comprometida até que seja replicada nos nós seguidores e o líder 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*).

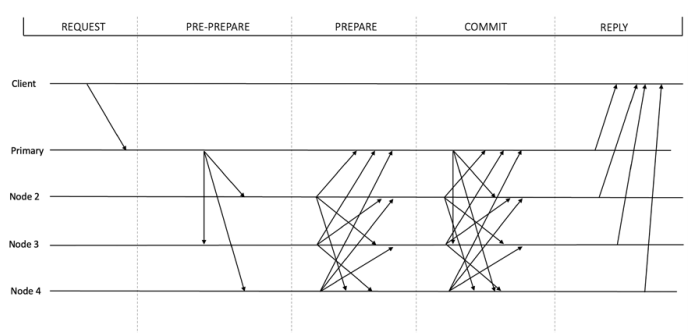
**Comprometimento** (*Commit*):  
Esta fase garante que as réplicas/nós honestos na rede concordem com a ordem total das requisições entre visões. Na fase *commit*, cada réplica envia uma mensagem de *commit* para todas as outras réplicas da rede. Assim como na fase *prepare*, as réplicas aguardam pelo 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.

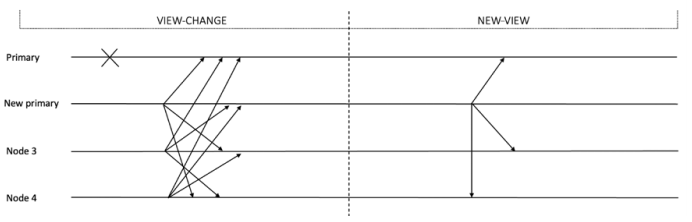
Quando essa mensagem chega ao novo primário, o primário espera por pelo menos **2f mensagens de VIEW-CHANGE** referentes à nova visão. Se receber ao menos 2f dessas mensagens, ele **difunde uma mensagem NEW-VIEW** para todas as réplicas e passa a operar novamente no modo normal.

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:

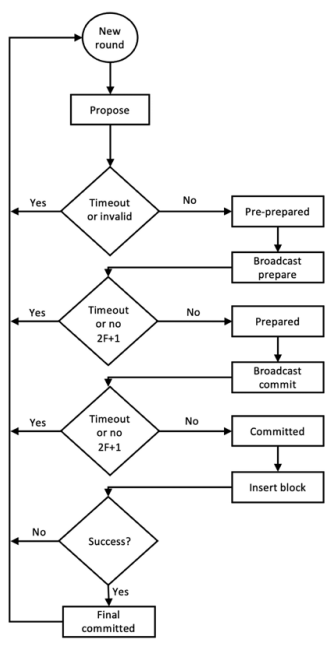
* No **IBFT**, a **participação dos validadores pode ser alterada dinamicamente**, e validadores podem ser votados para dentro ou para fora conforme necessário. Isso **contrasta com o PBFT original**, onde os nós validadores são estáticos.
* Existem **dois tipos de nós** em uma rede IBFT: **nós** e **validadores**. 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:

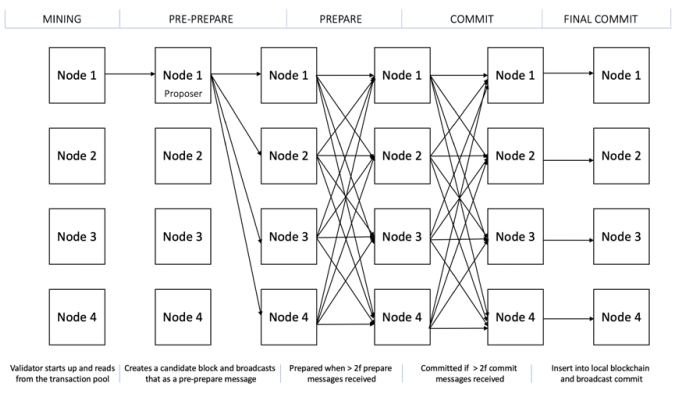
1. O protocolo começa com uma **nova rodada**. Nessa nova rodada, o proponente selecionado envia uma proposta (bloco) como uma mensagem *pre-prepare*.
2. Os nós que recebem essa mensagem *pre-prepare* a validam e a aceitam se ela for uma mensagem válida. Os nós então definem seu estado como **pre-preparado**.
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. É inspirado tanto pelos protocolos DLS quanto PBFT. O Tendermint também faz uso da abordagem SMR para fornecer replicação tolerante a falhas. Como vimos anteriormente, a replicação de máquina de estados é um mecanismo que permite sincronização entre réplicas/nós da rede.

O IBFT foi implementado em vários blockchains. Exemplos de implementações incluem o **Quorum** e o **Celo**. Uma implementação do Quorum está disponível no seguinte link:  
<https://github.com/ConsenSys/quorum>

O IBFT, com algumas modificações, também foi implementado no blockchain Celo, disponível em:  
<https://github.com/celo-org/celo-blockchain>

Tradicionalmente, um mecanismo de consenso era usado para operar com um pequeno número de participantes, e, portanto, desempenho e escalabilidade não eram grandes preocupações. No entanto, com o advento do blockchain, há a necessidade de desenvolver algoritmos que possam funcionar em redes de área ampla e em ambientes assíncronos. A pesquisa nessas áreas de computação distribuída não é nova, e especialmente agora, devido ao aumento das criptomoedas e do blockchain, o interesse por esses tópicos de pesquisa cresceu significativamente nos últimos anos.

O protocolo Tendermint funciona executando rodadas. Em cada rodada, um líder é eleito, o qual propõe o próximo bloco. Observe também que no Tendermint, a mudança de rodada ou mudança de visão é parte da operação normal, ao contrário do PBFT, onde a mudança de visão ocorre apenas em caso de erros, ou seja, um líder suspeito de falha. O Tendermint funciona de forma semelhante ao PBFT, onde são necessárias três fases para se chegar a 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 chave no Tendermint é o design de um novo mecanismo de terminação. Ao contrário de outros protocolos semelhantes ao PBFT, o Tendermint desenvolveu um mecanismo mais direto, que é semelhante à operação normal no estilo PBFT. Em vez de ter dois subprotocolos para modo normal e modo de mudança de visão (recuperação em caso de erros), o Tendermint termina sem quaisquer custos adicionais de comunicação.

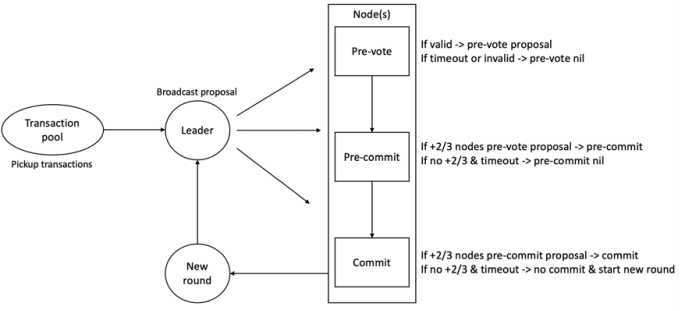
Como vimos anteriormente, na introdução, cada modelo de consenso é estudado, desenvolvido e executado sob um modelo de sistema com algumas suposições sobre o sistema. O Tendermint também é projetado com um modelo de sistema em mente. Agora definiremos e introduziremos cada elemento do modelo de sistema:

* **Processos**: Um processo é o participante fundamental do protocolo. Também é chamado de réplica (na literatura tradicional do PBFT), nó ou simplesmente processo. Os processos podem ser corretos ou honestos. Também podem ser falhos ou bizantinos. Cada processo possui algum poder de voto. Além disso, observe que os processos não estão necessariamente conectados diretamente; eles só precisam estar conectados de forma frouxa ou apenas com seu subconjunto imediato de processos/nós. Os processos têm um temporizador local que usam para medir o tempo limite.
* **Modelo de rede**: O modelo de rede é uma rede de processos que se comunicam usando mensagens. Em outras palavras, o modelo de rede é um conjunto de processos que se comunicam por troca de mensagens. Particularmente, o protocolo gossip é usado para comunicação entre processos. A suposição padrão de **n ≥ 3f + 1** BFT também é considerada. Isso significa que o protocolo opera corretamente desde que o número de nós na rede seja maior que 3f, onde f é o número de nós com falha. Isso implica que, no mínimo, deve haver quatro nós em uma rede para tolerar falhas bizantinas.
* **Suposições de temporização**: Sob o modelo de rede, o Tendermint assume uma rede parcialmente síncrona. Isso significa que há um limite desconhecido no atraso da comunicação, mas que se aplica apenas após um instante de tempo desconhecido chamado de **GST** (Global Stabilization Time). Na prática, isso significa que a rede se torna síncrona eventualmente, ou seja, após um ponto finito de tempo desconhecido.
* **Segurança e criptografia**: Assume-se que a criptografia de chave pública usada no sistema é segura e que a personificação ou falsificação de contas/identidades não é possível. As mensagens na rede são autenticadas e verificadas por meio de assinaturas digitais. O protocolo ignora quaisquer mensagens com assinatura digital inválida.
* **Replicação de máquina de estados (SMR)**: Para alcançar a replicação entre os nós, o mecanismo padrão de SMR é usado. Uma observação chave que é fundamental para o protocolo é que, em SMR, é garantido que todas as réplicas na rede recebam e processem a mesma sequência de requisições. Conforme observado no artigo do Tendermint, “acordo” e “ordem” são duas propriedades que garantem que todas as requisições sejam recebidas pelas réplicas, e a ordem garante que a sequência na qual as réplicas receberam as requisições seja a mesma. Ambos os requisitos garantem ordenação total no sistema. Além disso, o Tendermint assegura que as requisições sejam válidas e tenham sido propostas pelos clientes. Em outras palavras, apenas transações válidas são aceitas e executadas na rede.

Há três propriedades fundamentais de consenso que o Tendermint resolve. Como discutimos anteriormente no capítulo, geralmente, em um problema de consenso, há propriedades de **segurança** e **vivacidade** que precisam ser atendidas. Da mesma forma, no Tendermint, essas propriedades consistem em **acordo**, **terminação** e **validade**:

* **Acordo**: Nenhum dois processos corretos decidem sobre valores diferentes.
* **Terminação**: Todos os processos corretos eventualmente decidem um valor.
* **Validade**: Um valor decidido é válido, ou seja, satisfaz o predicado pré-definido denotado como valid().

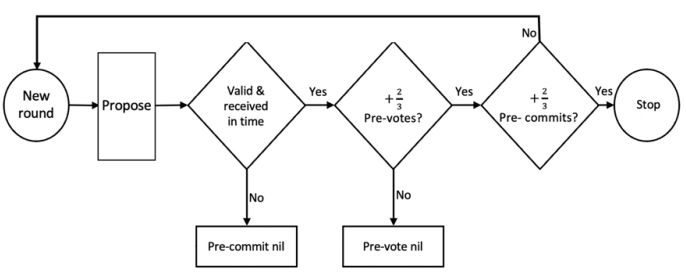
A transição de estado no Tendermint depende das mensagens recebidas e dos timeouts. Em outras palavras, o estado muda em resposta às mensagens recebidas por um processador ou no caso de timeouts. O mecanismo de timeout garante vivacidade e evita espera indefinida. Assume-se que, eventualmente, após um período de assincronia, haverá uma rodada ou período de comunicação durante o qual todos os processos podem se comunicar em tempo hábil, garantindo que os processos eventualmente decidam sobre um valor.

****

**Figura 5.8: Visão geral de alto nível do Tendermint**

O Tendermint funciona em rodadas e cada rodada compreende fases: **proposta**, **pré-votação**, **pré-comprometimento** e **comprometimento**:

1. Cada rodada começa com um valor sendo proposto por um proponente. O proponente pode propor qualquer novo valor no início da primeira rodada para cada altura.
2. Após a primeira rodada, qualquer rodada subsequente terá o proponente, que propõe um novo valor apenas se não houver valor válido presente, isto é, null. Caso contrário, o valor de decisão possível é proposto, o qual já foi bloqueado em uma rodada anterior. A mensagem de proposta também inclui o valor de uma rodada válida, que denota a última rodada em que houve um valor válido atualizado.
3. A proposta é aceita por um processo correto apenas se:
   * O valor proposto for válido;
   * O processo não tiver bloqueado uma rodada;
   * Ou o processo tiver um valor bloqueado.
4. Se as condições anteriores forem atendidas, então o processo correto aceitará a proposta e enviará uma mensagem **PRE-VOTE**.
5. Se as condições anteriores **não** forem atendidas, então o processo enviará uma mensagem **PRE-VOTE** com um valor nil.
6. Além disso, há também um mecanismo de timeout associado à fase de proposta, que inicia o timeout se um processo não tiver enviado uma mensagem **PRE-VOTE** na rodada atual, ou se o temporizador expirar na fase de proposta.
7. Se um processo correto recebe uma mensagem de proposta com um valor e **2f + 1** mensagens de pré-votação, então ele envia a mensagem **PRE-COMMIT**.
8. Caso contrário, ele envia um **pré-comprometimento nil**.
9. Um mecanismo de timeout associado ao **pré-comprometimento** será inicializado se o temporizador expirar ou se o processo não tiver enviado uma mensagem **PRE-COMMIT** após receber uma mensagem de proposta e **2f + 1** mensagens **PRE-COMMIT**.
10. Um processo correto decide sobre um valor se tiver recebido a mensagem de proposta em alguma rodada e **2f + 1** mensagens **PRE-COMMIT** para o ID do valor proposto.
11. Também há um mecanismo de timeout associado a esta etapa, que garante que o processador não espere indefinidamente para receber **2f + 1** mensagens. Se o temporizador expirar antes que o processador possa decidir, o processador inicia a próxima rodada.
12. Quando um processador finalmente decide, ele aciona a próxima instância de consenso para a próxima proposta de bloco, e todo o processo de proposta, pré-votação e pré-comprometimento começa novamente.

****

**Figura 5.9: Fluxograma do Tendermint**

**Tipos de mensagens**

Há três tipos de mensagens no Tendermint:

* **Proposta**: Como o nome sugere, essa é usada pelo líder da rodada atual para propor um valor ou bloco.
* **Pré-votação (Pre-vote)**: Essa mensagem é usada para votar em um valor proposto.
* **Pré-comprometimento (Pre-commit)**: Essa mensagem também é usada para votar em um valor proposto.

Essas mensagens podem ser consideradas, em certa medida, equivalentes às mensagens **PRE-PREPARE**, **PREPARE** e **COMMIT** do PBFT.

Todas as mensagens mencionadas anteriormente também têm um mecanismo de timeout correspondente, que garante que os processos não acabem esperando indefinidamente por alguma condição ser atendida. Se um processador não puder decidir em um tempo esperado, ele irá expirar e disparar uma mudança de rodada.

Cada tipo de mensagem tem um timeout associado. Assim, há três timeouts no Tendermint, correspondendo a cada tipo de mensagem:

* **Timeout-propose**
* **Timeout-prevote**
* **Timeout-precommit**

Observe que, no Tendermint, apenas a mensagem de proposta carrega o valor original, e as outras duas mensagens, **pre-vote** e **pre-commit**, operam em um identificador de valor, representando a proposta original.

Esses mecanismos de timeout impedem que o algoritmo fique aguardando indefinidamente para que uma condição seja atendida. Eles também garantem que os processos avancem pelas rodadas. Um mecanismo inteligente para aumentar o timeout a cada nova rodada garante que, após atingir o **GST**, eventualmente, a comunicação entre os processos corretos se torne confiável e uma decisão possa ser alcançada.

**Variáveis de estado**

Todos os processos no Tendermint mantêm um conjunto de variáveis que auxiliam na execução do algoritmo. Cada uma dessas variáveis armazena valores críticos, os quais garantem a execução correta do algoritmo.

Essas variáveis são listadas e discutidas a seguir:

* **Step**: A variável *step* contém informações sobre o estado atual do algoritmo, ou seja, o estado atual da máquina de estados do Tendermint na rodada atual.
* **lockedValue**: A variável *lockedValue* armazena o valor mais recente (com respeito a um número de rodada) para o qual uma mensagem *pre-commit* foi enviada.
* **lockedRound**: A variável *lockedRound* contém informações sobre a última rodada na qual o processo enviou uma mensagem **PRE-COMMIT** diferente de nil. Esta é a rodada em que um possível valor de decisão foi bloqueado. Isso significa que, se uma mensagem de proposta e as correspondentes **2f + 1** mensagens foram recebidas para um valor em uma rodada, então, devido ao fato de que **2f + 1** pré-votações já foram recebidas para esse valor, esse é um possível valor de decisão.
* **validValue**: O papel da variável *validValue* é armazenar o valor de decisão possível mais recente.
* **validRound**: A variável *validRound* representa a última rodada na qual *validValue* foi atualizado.

Além das variáveis anteriores, um processo também armazena a instância atual de consenso (chamada de **altura** no Tendermint) e o número da rodada atual. Essas variáveis são anexadas a cada mensagem. Um processo também armazena um array de decisões. O Tendermint assume uma sequência de instâncias de consenso, uma para cada altura.

Agora vamos explorar o novo mecanismo de terminação. Para esse propósito, há duas variáveis, a saber, **validValue** e **validRound**, que são usadas pela mensagem de proposta. Ambas as variáveis são atualizadas por um processo correto quando o processo recebe uma mensagem de proposta válida e **2f + 1** mensagens **PRE-VOTE** subsequentes/correspondentes.

Esse processo funciona utilizando o protocolo gossip, que garante que, se um processo correto bloqueou um valor em uma rodada, todos os processos corretos então atualizarão suas variáveis *validValue* e *validRound* com os valores bloqueados até o final da rodada durante a qual foram bloqueados. A ideia principal é que, uma vez que esses valores tenham sido bloqueados por um processador correto, eles serão propagados para outros nós dentro da mesma rodada, e cada processador saberá o valor bloqueado e a rodada, ou seja, os valores válidos.

**lockedValue**, **lockedRound**, **validValue** e **validRound** são redefinidos para os valores iniciais toda vez que uma decisão é alcançada.

Agora, quando a próxima proposta é feita, os valores bloqueados serão selecionados pelo proponente, e eles já terão sido bloqueados como resultado da proposta válida e das correspondentes **2f + 1** mensagens **PRE-VOTE**. Dessa forma, pode-se garantir que o valor sobre o qual os processos eventualmente decidirão é aceitável conforme as condições de validade descritas anteriormente.

Os algoritmos discutidos até agora são variantes dos algoritmos tradicionais tolerantes a falhas bizantinas. Agora, vamos introduzir os protocolos que são especificamente desenvolvidos para protocolos de blockchain.

**Consenso Nakamoto**

O consenso Nakamoto, ou *Proof of Work* (PoW), foi introduzido pela primeira vez com o Bitcoin em 2009. Desde então, resistiu ao teste do tempo e é a rede blockchain com funcionamento mais longo. Ao contrário da crença comum, o mecanismo PoW não é um algoritmo de consenso, mas sim um mecanismo de defesa contra ataques Sybil. O consenso é alcançado aplicando a regra de escolha do fork para escolher a cadeia mais longa/mais pesada para finalizar a cadeia canônica. Nesse sentido, pode-se dizer que o PoW é um algoritmo de facilitação de consenso que facilita o consenso e mitiga ataques Sybil, mantendo assim a consistência do blockchain e a segurança da rede.

A ideia-chave por trás do PoW como solução para o problema dos generais bizantinos é que todos os generais honestos (mineradores no mundo do Bitcoin) alcançam acordo sobre o mesmo estado (valor de decisão). Desde que participantes honestos controlem a maioria da rede, o PoW resolve o problema dos generais bizantinos. Observe que esta é uma solução **probabilística**, e não determinística.

É bastante fácil obter múltiplas identidades e tentar influenciar a rede. No entanto, no Bitcoin, devido aos requisitos de poder de hashing, esse ataque é mitigado.

O Tendermint é desenvolvido em Go. Ele pode ser implementado em vários cenários. O código-fonte está disponível em:  
<https://github.com/tendermint/tendermint>  
Também é lançado como **Tendermint Core**, que é um middleware de programação independente de linguagem que pega uma máquina de transição de estado e a replica em várias máquinas. O Tendermint Core está disponível aqui:  
<https://tendermint.com/core/>

Um ataque **Sybil** é um tipo de ataque que visa obter influência majoritária na rede para controlá-la. Uma vez que uma rede esteja sob o controle de um adversário, qualquer atividade maliciosa pode ocorrer. Um ataque Sybil é geralmente conduzido por um nó que gera e usa múltiplas identidades na rede. Se houver identidades suficientes mantidas por uma entidade, então essa entidade pode influenciar a rede ao distorcer decisões baseadas na maioria. A maioria, nesse caso, está nas mãos do adversário.

A postagem original de Satoshi Nakamoto pode ser encontrada aqui:  
<https://satoshi.nakamotoinstitute.org/emails/cryptography/11/>

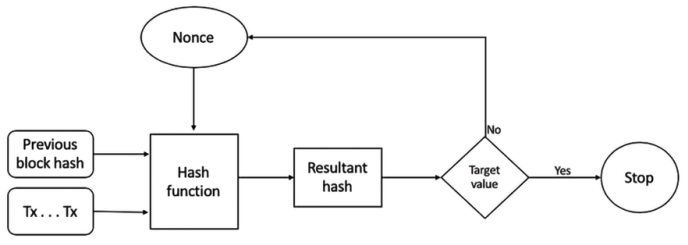
Em resumo, o PoW funciona da seguinte forma:

* O PoW faz uso de quebra-cabeças de hash.
* Um nó que propõe um bloco precisa encontrar um *nonce* tal que:  
  **H(nonce || hash anterior || Tx || … || Tx) < valor limiar**.

Esse valor limiar ou valor alvo é baseado na dificuldade de mineração da rede, que é ajustado ao adicionar ou remover zeros à esquerda do valor hash alvo. Mais zeros significam maior dificuldade, e menos zeros significam menor dificuldade para resolver. Se houver mais mineradores na rede, então a dificuldade aumenta, caso contrário, ela diminui à medida que o número de mineradores diminui.

O processo pode ser resumido da seguinte forma:

1. Novas transações são transmitidas para a rede.
2. Cada nó coleta as transações em um bloco candidato.
3. Mineradores propõem novos blocos.
4. Mineradores concatenam e aplicam hash com o cabeçalho do bloco anterior.
5. O hash resultante é comparado com o valor alvo da rede (dificuldade da rede).
6. Se o hash resultante for menor que o valor limiar (valor alvo), então o PoW é resolvido;  
   caso contrário, o *nonce* é incrementado e o nó tenta novamente.  
   Esse processo continua até que se encontre um hash menor que o valor limiar.

****

**Figura 5.10: PoW**

**Nakamoto versus consenso tradicional**

O consenso Nakamoto foi o primeiro de seu tipo. Ele resolveu o problema de consenso, que anteriormente era resolvido por protocolos pré-Bitcoin, como o PBFT. No entanto, podemos mapear as propriedades do consenso PoW para as propriedades tradicionais do consenso bizantino.

Nos algoritmos de consenso tradicionais, temos as propriedades de **acordo**, **validade** e **vivacidade**, que podem ser mapeadas para as propriedades específicas do Nakamoto: **prefixo comum**, **qualidade da cadeia** e **crescimento da cadeia**, respectivamente:

* **Propriedade do prefixo comum** significa que o blockchain mantido por nós honestos compartilhará o mesmo grande prefixo comum. Se isso não ocorrer, então a propriedade de acordo do protocolo não pode ser garantida, ou seja, os processadores não conseguirão decidir e concordar sobre o mesmo valor.
* **Propriedade da qualidade da cadeia** significa que o blockchain contém um certo nível mínimo de blocos corretos criados por nós honestos (mineradores). Se a qualidade da cadeia for comprometida, então a propriedade de validade do protocolo não pode ser garantida. Isso significa que existe a possibilidade de que um valor seja decidido que não foi proposto por um processo correto, resultando em violações de segurança.
* **Propriedade do crescimento da cadeia** significa simplesmente que novos blocos corretos são continuamente adicionados ao blockchain. Se o crescimento da cadeia for impactado, então a propriedade de vivacidade do protocolo não pode ser garantida. Isso significa que o sistema pode entrar em deadlock ou falhar ao decidir um valor.

**Variantes de PoW**

Existem duas principais variantes dos algoritmos de PoW, com base no tipo de hardware utilizado para seu processamento:

* **PoW dependente de CPU (CPU-bound PoW)** refere-se a um tipo de PoW onde o processamento necessário para encontrar a solução do quebra-cabeça criptográfico é diretamente proporcional à velocidade de cálculo da CPU ou de hardware como os ASICs. Como os ASICs dominaram o PoW do Bitcoin e oferecem certa vantagem indevida aos mineradores que podem pagar por esses dispositivos, este PoW dependente de CPU é visto como uma tendência à centralização. Além disso, *mining pools* com poder de hashing extraordinário podem inclinar o equilíbrio de poder para si. Por isso, algoritmos PoW dependentes de memória foram introduzidos, os quais são resistentes a ASICs e são baseados em um design orientado à memória em vez de CPU.
* **PoW dependente de memória (Memory-bound PoW)** depende da RAM do sistema para fornecer PoW. Aqui, o desempenho é limitado pela velocidade de acesso à memória ou pelo tamanho da memória. Essa dependência da memória também torna esses algoritmos PoW resistentes a ASICs. O **Equihash** é um dos algoritmos de PoW dependentes de memória mais proeminentes.

O PoW consome quantidades enormes de energia; como tal, vários alternativas têm sido sugeridas por pesquisadores. Uma das primeiras alternativas propostas é o **PoS**.

As propriedades de qualidade da cadeia e prefixo comum do PoW são introduzidas e discutidas no seguinte artigo:

Garay, J., Kiayias, A. e Leonardos, N., 2015, abril. *The bitcoin backbone protocol: Analysis and applications*. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (pp. 281-310). Springer, Berlin, Heidelberg.  
<https://eprint.iacr.org/2014/765.pdf>

**Alternativas ao PoW**

O PoW possui várias desvantagens, e a maior de todas é o consumo de energia — a eletricidade total consumida por mineradores de Bitcoin é maior do que a de muitos países! Isso é enorme, e toda essa energia é, de certa forma, desperdiçada; nenhum propósito útil é servido além da mineração. Ambientalistas levantaram preocupações reais sobre essa situação. Além do consumo de eletricidade, a pegada de carbono também é muito alta e só tende a crescer.

Foi proposto que os quebra-cabeças do PoW possam ser projetados de forma a servir a dois propósitos. Primeiro, seu objetivo principal está nos mecanismos de consenso e, segundo, eles servem para realizar algum cálculo científico útil. Dessa forma, os esquemas poderiam ser usados não apenas para mineração, mas também para ajudar a resolver outros problemas científicos.

Outra desvantagem do esquema PoW, especialmente o usado no Bitcoin (*Double SHA-256*), é que, desde a introdução dos ASICs, o poder está se concentrando nas mãos dos mineradores ou pools de mineração que podem operar fazendas de ASICs em larga escala. Essa mudança de poder desafia a filosofia central da descentralização do Bitcoin. Algumas alternativas foram propostas, como os **quebra-cabeças resistentes a ASICs**, que são projetados de tal forma que a construção de ASICs para resolvê-los é inviável e não oferece ganho de desempenho significativo sobre o hardware comum. Uma técnica comum para isso é aplicar uma classe de problemas computacionalmente difíceis chamados de **quebra-cabeças computacionais com uso intensivo de memória** (*memory-hard*).

Essa técnica foi inicialmente usada no **Litecoin** e **Tenebrix**, onde a função hash **Scrypt** foi usada como um esquema PoW resistente a ASIC. Embora esse esquema tenha sido inicialmente divulgado como resistente a ASICs, ASICs para Scrypt tornaram-se disponíveis alguns anos depois, refutando a alegação original do Litecoin. Isso aconteceu porque, mesmo que o Scrypt seja um mecanismo intensivo em memória, inicialmente pensava-se que construir ASICs com grandes memórias fosse difícil por limitações técnicas e de custo. Isso já não é mais o caso, pois o hardware de memória está se tornando cada vez mais barato. Além disso, com a capacidade de produzir circuitos em escala nanométrica, é possível construir ASICs que executam o algoritmo Scrypt.

Outra abordagem para resistência a ASICs é aquela em que várias funções hash precisam ser calculadas para fornecer o PoW. Isso também é chamado de **esquema de hashing encadeado** (*chained hashing*). A justificativa por trás dessa ideia é que projetar várias funções hash em um ASIC não é muito viável. O exemplo mais comum é a função **X11 memory-hard**, implementada no Dash. O X11 compreende 11 algoritmos concorrentes do SHA-3, onde um algoritmo fornece a saída de hash calculada para o próximo algoritmo, até que todos os 11 algoritmos sejam usados em sequência. Esses algoritmos incluem: **BLAKE**, **BMW**, **Groestl**, **JH**, **Keccak**, **Skein**, **Luffa**, **CubeHash**, **SHAvite**, **SIMD** e **ECHO**. Essa abordagem inicialmente forneceu alguma resistência ao desenvolvimento de ASICs, mas atualmente mineradores ASIC estão disponíveis comercialmente e suportam a mineração de X11 e esquemas similares.

Talvez outra abordagem seja projetar quebra-cabeças **auto-mutantes** que mudam de forma inteligente ou aleatoriamente o esquema PoW ou seus requisitos como uma função do tempo. Essa estratégia tornaria quase impossível sua implementação em ASICs, pois exigiria múltiplos ASICs projetados para cada função. Além disso, esquemas que mudam aleatoriamente seriam praticamente impossíveis de lidar em ASICs. No momento, não está claro como isso pode ser alcançado na prática.

Para dados atualizados sobre a pegada de carbono do Bitcoin, acesse:  
<https://digiconomist.net/bitcoin-energy-consumption>

**Prova de Armazenamento**

Também conhecida como **proof of retrievability** (prova de recuperabilidade), esta é outro tipo de prova de trabalho útil que exige o armazenamento de uma grande quantidade de dados. Introduzida pela Microsoft Research, essa abordagem oferece um benefício útil ao distribuir o armazenamento de dados arquivados. Os mineradores são obrigados a armazenar um subconjunto selecionado pseudoaleatoriamente de dados grandes para poderem realizar a mineração.

**Prova de Participação (Proof of Stake - PoS)**

A prova de participação (PoS) também é chamada de mineração virtual. Esta é outra forma de quebra-cabeça de mineração que foi proposta como alternativa aos esquemas tradicionais de PoW. Foi proposta pela primeira vez no Peercoin, em agosto de 2012, e agora blockchains proeminentes como **EOS**, **NxT**, **Steem**, **Ethereum 2.0**, **Polkadot** e **Tezos** estão usando variantes dos algoritmos PoS. O Ethereum, com sua atualização **Serenity**, fará em breve a transição para um mecanismo de consenso baseado em PoS.

Nesse esquema, a ideia é que os usuários devem demonstrar a posse de uma certa quantidade de moeda (tokens), provando assim que possuem participação (stake) na moeda. A forma mais simples dessa participação é aquela em que a mineração se torna comparativamente mais fácil para os usuários que detêm maiores quantidades da moeda digital. Os benefícios desse esquema são dois: primeiro, adquirir grandes quantidades de moeda digital é relativamente difícil comparado à compra de dispositivos ASIC de alto desempenho; segundo, resulta em economia de recursos computacionais.

Várias formas de *stake* têm sido propostas:

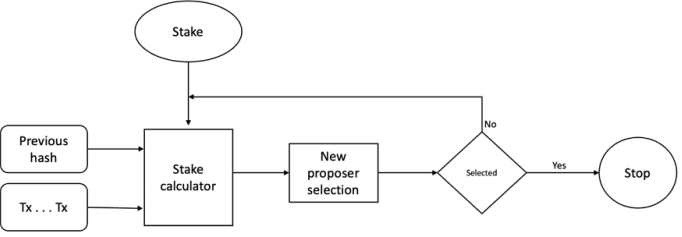
* **Prova de tempo de posse (Proof of Coinage)**: A idade de uma moeda é o tempo desde que as moedas foram usadas ou mantidas pela última vez. Essa é uma abordagem diferente da forma usual de PoS, em que a mineração se torna mais fácil para usuários com maior participação na altcoin. Na abordagem baseada em idade da moeda, a idade é redefinida toda vez que um bloco é minerado. O minerador é recompensado por manter e não gastar moedas por um certo tempo. Esse mecanismo foi implementado no **Peercoin**, combinado com PoW de maneira criativa. A dificuldade dos quebra-cabeças de mineração (PoW) é inversamente proporcional à idade da moeda (*coinage*), o que significa que, se os mineradores consumirem parte da idade da moeda usando transações de *stake*, então os requisitos de PoW são aliviados.
* **Prova de Depósito (Proof of Deposit)**: Esse é um tipo de PoS. A ideia central desse esquema é que os blocos recém-criados pelos mineradores se tornam inutilizáveis (não podem ser gastos) por um certo período. Mais precisamente, as moedas ficam bloqueadas por um número fixo de blocos durante a operação de mineração. O esquema funciona permitindo que os mineradores realizem a mineração ao custo de congelar uma certa quantidade de moedas por um tempo.
* **Prova de Queima (Proof of Burn)**: Como uma despesa alternativa ao poder computacional, esse método destrói uma certa quantidade de Bitcoins para obter altcoins equivalentes. Isso é comumente usado no início de novos projetos de moedas como uma forma de fornecer uma distribuição inicial justa. Pode ser considerado um esquema alternativo de mineração, onde o valor das novas moedas vem do fato de que anteriormente um certo número de moedas foi destruído.

A participação (*stake*) representa o número de moedas (dinheiro) no protocolo de consenso que foram empenhadas por um participante do blockchain. A ideia-chave é que, se alguém tem participação no sistema, então essa pessoa não tentará sabotá-lo. A chance de propor o próximo bloco é diretamente proporcional ao valor empenhado pelo participante.

Nos sistemas PoS, não há o conceito de mineração como no consenso Nakamoto tradicional. No entanto, o processo relacionado a ganhar receita às vezes é chamado de **mineração virtual**. Um minerador PoS é chamado de **validador**, **minter** ou **stakeholder**.

O direito de se tornar o próximo proponente geralmente é atribuído de forma aleatória. Os proponentes são recompensados com taxas de transação ou recompensas por bloco. Similar ao PoW, o controle da maioria da rede — na forma de controle de uma grande parte da participação — é necessário para atacar e controlar a rede.

Os mecanismos PoS geralmente selecionam um stakeholder e concedem os direitos apropriados com base em seus ativos empenhados. O cálculo da participação é específico para cada aplicação, mas geralmente é baseado no saldo, valor do depósito ou votação entre validadores. Uma vez que a participação é calculada e um stakeholder é selecionado para propor um bloco, o bloco proposto por esse proponente é prontamente aceito. A probabilidade de seleção aumenta com uma participação maior. Em outras palavras, quanto maior a participação, maiores as chances de ganhar o direito de propor o próximo bloco.

****

**Figura 5.11: PoS**

No diagrama anterior, uma função de cálculo de participação é usada para calcular a quantia de fundos empenhados, e com base nisso, um novo proponente é selecionado. Se por algum motivo a seleção do proponente falhar, então a função de cálculo da participação e seleção de novo proponente é executada novamente para escolher um novo proponente de bloco (líder).

O **PoS baseado em cadeia (chain-based PoS)** é muito semelhante ao PoW. A única mudança em relação ao mecanismo PoW é o método de geração de blocos. Um bloco é gerado em duas etapas seguindo um protocolo simples:

* As transações são retiradas do pool de memória e um bloco candidato é criado.
* Um relógio é configurado com um intervalo de tique constante e, a cada tique, verifica-se se o hash do cabeçalho do bloco concatenado com o tempo do relógio é menor que o produto do valor alvo e o valor da participação (*stake*).

Esse processo pode ser representado por uma fórmula simples:

**Hash(B || tempo do relógio) < alvo × valor da participação**

O valor da participação depende de como o algoritmo é projetado. Em alguns sistemas, ele é diretamente proporcional à quantidade de *stake*, e em outros, é baseado no tempo durante o qual o *stake* foi mantido pelo participante (também chamado de **coinage**). O **alvo** é a dificuldade de mineração por unidade do valor de participação.

Esse mecanismo ainda usa quebra-cabeças de hash, como no PoW. Mas, em vez de competir para resolver o quebra-cabeça de hash consumindo grandes quantidades de eletricidade e hardware especializado, o quebra-cabeça de hash no PoS é resolvido em intervalos regulares com base no tique do relógio. Um quebra-cabeça de hash se torna mais fácil de resolver se o valor da participação do minerador for alto.

No **PoS baseado em comitê (committee-based PoS)**, um grupo de stakeholders é escolhido aleatoriamente, geralmente por meio de uma **função aleatória verificável (VRF)**. Essa VRF, uma vez invocada, produz um conjunto aleatório de stakeholders com base em suas participações e no estado atual do blockchain. O grupo escolhido de stakeholders torna-se responsável por propor blocos em ordem sequencial.

O **PoS Delegado (Delegated PoS – DPoS)** é muito semelhante ao PoS baseado em comitê, mas com uma diferença crucial. Em vez de usar uma função aleatória para derivar o grupo de stakeholders, o grupo é escolhido por delegação de participação. O grupo selecionado é composto por um número fixo de mineradores que criam blocos em um esquema rotativo. Os delegados são escolhidos por votação dos usuários da rede. Os votos são proporcionais à quantidade de *stake* que os participantes possuem na rede. Essa técnica é usada em **Lisk**, **Cosmos** e **EOS**. O DPoS **não é descentralizado**, pois um pequeno número de usuários conhecidos é responsável por propor e gerar blocos.

**Proof of Activity (PoA)**

Esse esquema é um híbrido entre PoW e PoS. Nele, os blocos são inicialmente produzidos usando PoW, mas depois cada bloco atribui aleatoriamente três stakeholders que devem assiná-lo digitalmente. A validade dos blocos subsequentes depende da assinatura bem-sucedida dos blocos anteriormente escolhidos aleatoriamente.

No entanto, há um possível problema conhecido como o **problema do nada em jogo (nothing at stake)**, no qual seria trivial criar um fork do blockchain. Isso é possível porque, no PoW, são necessários recursos computacionais apropriados para minerar, enquanto no PoS, não há tal exigência; como resultado, um invasor pode tentar minerar em várias cadeias ao mesmo tempo usando a mesma moeda.

O **Peercoin** foi o primeiro blockchain a implementar PoS.  
**Nxt** (<https://www.jelurida.com/nxt>) e **Peercoin** (<https://www.peercoin.net>) são dois exemplos de blockchains onde o PoS baseado em cadeia é implementado.

Esse mecanismo é usado no mecanismo de consenso **Ouroboros PoS**, que é utilizado no **Cardano**. Mais detalhes estão disponíveis em:  
<https://www.cardano.org/en/ouroboros/>

As VRFs foram introduzidas no Capítulo 4, Criptografia Assimétrica. Mais informações sobre VRF podem ser encontradas em:  
<https://datatracker.ietf.org/doc/html/draft-goldbe-vrf-01>

**Quebra-cabeças não terceirizáveis (Non-outsourceable puzzles)**

A motivação principal por trás desse tipo de quebra-cabeça é desenvolver resistência contra o surgimento de **pools de mineração**. Como discutido anteriormente, os pools de mineração oferecem recompensas a todos os participantes proporcionalmente ao poder computacional que utilizam. No entanto, nesse modelo, o operador do pool de mineração é uma autoridade central à qual todas as recompensas são enviadas e que pode impor regras específicas. Além disso, nesse modelo, todos os mineradores apenas confiam uns nos outros porque estão trabalhando com um objetivo comum, na esperança de que o gerente do pool receba a recompensa. **Quebra-cabeças não terceirizáveis** são um esquema que permite que mineradores reivindiquem recompensas para si mesmos; consequentemente, a formação de pools se torna improvável devido à desconfiança inerente entre mineradores anônimos.

**HotStuff**

**HotStuff** é a classe mais recente de protocolo BFT, com várias otimizações. Existem várias mudanças no HotStuff que o tornam um protocolo diferente e, de certa forma, melhor do que o PBFT tradicional. O HotStuff foi introduzido pela **VMware Research** em 2018. Posteriormente, foi apresentado no *Symposium on Principles of Distributed Computing*.

As três principais otimizações do HotStuff estão listadas abaixo:

* A **mudança de visão linear** resulta em menor complexidade de comunicação. Isso é alcançado por meio do algoritmo, no qual, após o GST ser atingido, um líder correto designado enviará apenas **O(n)** autenticadores (seja uma assinatura parcial ou uma assinatura) para alcançar o consenso. No pior cenário, onde líderes falham sucessivamente, o custo de comunicação é **O(n²)** — quadrático. Em termos simples, complexidade quadrática significa que o desempenho do algoritmo é proporcional ao quadrado do tamanho da entrada.
* A **responsividade otimista** garante que qualquer líder correto, após o GST ser alcançado, precise apenas das primeiras **n−f** respostas para garantir progresso.
* A **qualidade da cadeia** garante equidade e vivacidade no sistema ao permitir rotação rápida e frequente de líderes.

Outra inovação do HotStuff é a **separação dos mecanismos de segurança e vivacidade**. A separação de responsabilidades permite melhor modularidade, arquitetura mais limpa e controle sobre o desenvolvimento dessas funcionalidades de forma independente. A **segurança** é garantida por meio de regras de votação e de comprometimento para os nós participantes da rede. A **vivacidade**, por outro lado, é responsabilidade de um módulo separado chamado **Pacemaker**, que garante a eleição de um novo líder correto e único.

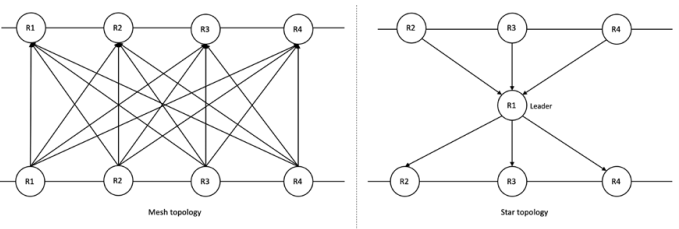
A referência do artigo é a seguinte:

Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G. e Abraham, I., 2019, julho. *HotStuff: BFT consensus with linearity and responsiveness*. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (pp. 347–356). ACM.

Em comparação com o PBFT tradicional, o **HotStuff** introduziu várias mudanças, que resultam em desempenho aprimorado. Primeiramente, os protocolos no estilo PBFT funcionam utilizando uma **topologia de comunicação em malha**, onde cada mensagem precisa ser transmitida para os demais nós da rede. No **HotStuff**, a comunicação foi alterada para a **topologia em estrela**, o que significa que os nós não se comunicam diretamente entre si, mas todas as mensagens de consenso são coletadas por um líder e, então, retransmitidas para os outros nós. Isso resulta imediatamente em uma **complexidade de comunicação reduzida**.

Uma pergunta surge aqui: o que acontece se o líder estiver corrompido ou comprometido? Esse problema é resolvido pelas mesmas regras de tolerância a falhas bizantinas, nas quais, se um líder propuser um bloco malicioso, ele será rejeitado pelos demais validadores honestos e um novo líder será escolhido. Esse cenário pode desacelerar a rede por um tempo limitado (até que um novo líder honesto seja eleito), mas eventualmente (desde que a maioria da rede seja honesta), um líder honesto será escolhido, que proporá um bloco válido. Além disso, para maior proteção, geralmente o papel de líder é **rotacionado frequentemente** (geralmente, a cada bloco) entre os validadores, o que pode neutralizar qualquer ataque malicioso direcionado à rede. Essa propriedade assegura **equidade**, o que contribui para alcançar a qualidade da cadeia mencionada anteriormente.

Entretanto, observe que há um problema nesse esquema: se o líder estiver **sobrecarregado demais**, então o processamento pode se tornar lento, impactando toda a rede. As topologias **em malha** e **em estrela** podem ser visualizadas no diagrama a seguir:

****

**Figura 5.12: Topologia em estrela**

Uma segunda mudança refere-se aos dois subprotocolos principais do PBFT: **modo normal** e **modo de mudança de visão**. Nesse modelo, o modo de mudança de visão é acionado quando um líder é suspeito de falha. Essa abordagem realmente oferece uma garantia de vivacidade ao PBFT, mas **aumenta significativamente a complexidade de comunicação**. O **HotStuff** resolve isso **mesclando o processo de mudança de visão com o modo normal**. Isso significa que os nós podem alternar para uma nova visão diretamente, sem esperar que um número mínimo de mensagens de mudança de visão seja recebido por outros nós. No PBFT, os nós aguardam **2f + 1** mensagens antes que a mudança de visão possa ocorrer, mas no **HotStuff**, essa mudança pode acontecer diretamente sem exigir um novo subprotocolo. Em vez disso, a verificação do limiar de mensagens para mudar de visão passa a fazer parte da visão normal.

Quando comparado com o **Tendermint**, o HotStuff apresenta melhorias em dois aspectos. Primeiro, ele pode ser **encadeado** (*chained*), o que significa que pode ser **pipelined**, onde um único certificado de quórum pode ser usado em diferentes fases ao mesmo tempo, resultando em melhor desempenho. Segundo, ele é **otimisticamente responsivo**, como mencionado anteriormente.

Assim como o PBFT, o **HotStuff** também resolve o problema de **Replicação de Máquina de Estados (SMR)**. Agora, descreveremos como esse protocolo funciona. Como vimos anteriormente, os algoritmos de consenso são descritos e operam dentro de um modelo de sistema.

O sistema é baseado na suposição BFT padrão de que **n = 3f + 1** nós existem no sistema, onde **f** é o número de nós com falha e **N** é o número total de nós na rede. Os nós se comunicam entre si por meio de **passagem de mensagens ponto a ponto**, utilizando **links de comunicação confiáveis e autenticados**. Assume-se que a rede é **parcialmente síncrona**.

O HotStuff faz uso de **assinaturas de limiar (threshold signatures)**, onde uma única **chave pública** é usada por todos os nós, mas cada réplica utiliza uma **chave privada única**. O uso de assinaturas de limiar resulta na **redução da complexidade de comunicação** do protocolo. Além disso, funções de hash criptográficas são usadas para fornecer identificadores únicos para mensagens.

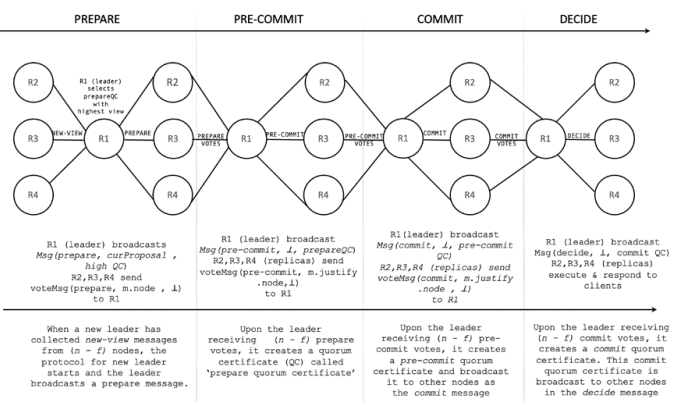
O HotStuff opera em fases, a saber:

* **Fase de preparação (Prepare)**: Assim que um novo líder coleta mensagens **NEW-VIEW** de **n − f** nós, o protocolo para o novo líder começa. O líder coleta e processa essas mensagens para determinar o ramo mais recente no qual o certificado de quórum mais alto de mensagens de preparação foi formado.
* **Pré-comprometimento (Pre-commit)**: Assim que um líder recebe **n − f** votos de preparação, ele cria um certificado de quórum chamado “**certificado de preparação**” (*prepare quorum certificate*). Esse certificado é então transmitido aos outros nós como uma mensagem **PRE-COMMIT**. Quando uma réplica recebe a mensagem **PRE-COMMIT**, ela responde com um voto de pré-comprometimento. O certificado de quórum indica que o limiar necessário de nós confirmou a requisição.
* **Comprometimento (Commit)**: Quando o líder recebe **n − f** votos de pré-comprometimento, ele cria um certificado de quórum de **PRE-COMMIT** e o transmite aos demais nós como uma mensagem **COMMIT**. Quando as réplicas recebem essa mensagem **COMMIT**, elas respondem com seus votos de comprometimento. Neste ponto, as réplicas bloqueiam o certificado de quórum de **PRE-COMMIT** para garantir a segurança do algoritmo mesmo se ocorrer mudança de visão.
* **Decisão (Decide)**: Quando o líder recebe **n − f** votos de comprometimento, ele cria um certificado de quórum **COMMIT**. Esse certificado é transmitido aos demais nós como a mensagem **DECIDE**. Quando as réplicas recebem essa mensagem **DECIDE**, elas executam a requisição, pois essa mensagem contém um valor/certificado já comprometido. Uma vez que a transição de estado ocorre como resultado do processamento da mensagem **DECIDE** por uma réplica, uma nova visão é iniciada.

Estudamos as **assinaturas de limiar** no Capítulo 4, *Criptografia Assimétrica*. Você pode consultar os detalhes lá, se necessário.

Um termo comum que veremos na próxima seção é **certificado de quórum (quorum certificate – QC)**.  
Trata-se de uma estrutura de dados que representa uma **coleção de assinaturas produzidas por N − F réplicas** para demonstrar que o número mínimo de mensagens foi atingido. Em outras palavras, é simplesmente um conjunto de votos de **n − f** nós.

Este algoritmo pode ser visualizado no seguinte diagrama:



**Figura 5.13: Protocolo HotStuff**

O **HotStuff**, com algumas variações, também é usado no **DiemBFT**, que é um banco de dados distribuído projetado para oferecer suporte a uma moeda global proposta pelo Facebook.

O **HotStuff** garante **vivacidade (progresso)** usando o **Pacemaker**, que garante progresso após o GST dentro de um intervalo de tempo limitado. Este componente possui dois elementos:

* Existem **intervalos de tempo** durante os quais todas as réplicas permanecem em uma determinada altura por períodos suficientemente longos. Essa propriedade pode ser alcançada aumentando progressivamente o tempo até que o progresso ocorra (ou seja, até que uma decisão seja tomada).
* Um **líder único e correto** é eleito para aquela altura. Novos líderes podem ser eleitos de forma determinística usando um esquema de rotação de líderes ou funções pseudoaleatórias.

A **segurança** no **HotStuff** é garantida por regras de votação e de comprometimento relevantes. A **vivacidade** e a **segurança** são mecanismos **separados**, o que permite **desenvolvimento independente**, **modularidade** e **separação de responsabilidades**.

O **HotStuff** é um protocolo simples, porém poderoso, que fornece propriedades de **linearidade** e **responsividade**. Ele permite **consenso sem latência adicional**, na velocidade real da rede.

Mais detalhes sobre o protocolo **Diem** estão disponíveis aqui:  
<https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf>

**Conclusão do HotStuff**

Além disso, trata-se de um protocolo que apresenta **complexidade de comunicação linear**, reduzindo assim o custo de comunicação quando comparado a protocolos no estilo PBFT. É também um **framework** no qual outros protocolos, como **DLS**, **PBFT** e **Tendermint**, podem ser expressos.

**Escolhendo um algoritmo**

Escolher um algoritmo de consenso depende de diversos fatores. Não apenas é algo dependente do caso de uso, como também pode exigir **compromissos** para criar um sistema que atenda a todos os requisitos **sem comprometer** as propriedades fundamentais de **segurança** e **vivacidade**. Vamos discutir alguns dos principais fatores que podem influenciar a escolha de um algoritmo de consenso. Observe que esses fatores são **diferentes das propriedades fundamentais** de segurança e vivacidade discutidas anteriormente, que são requisitos obrigatórios para qualquer mecanismo de consenso. Os fatores a seguir são **específicos do caso de uso** e afetam a escolha do algoritmo. Eles incluem: **finalidade**, **velocidade**, **desempenho** e **escalabilidade**.

**Finalidade (Finality)**

**Finalidade** refere-se ao conceito de que, uma vez que uma transação foi concluída, ela não pode ser revertida. Em outras palavras, se uma transação foi confirmada no blockchain, ela **não será revogada ou desfeita**. Esse recurso é especialmente importante em redes financeiras, onde, uma vez que uma transação foi realizada, o consumidor pode ter **confiança** de que ela é **irrevogável e definitiva**.

Existem dois tipos de finalidade: **probabilística** e **determinística**.

* A **finalidade probabilística**, como o nome indica, oferece uma garantia de finalidade baseada em probabilidade. A certeza de que uma transação, uma vez confirmada no blockchain, não será revertida **cresce com o tempo**, em vez de ser imediata. Por exemplo, no consenso Nakamoto, a probabilidade de que uma transação não seja revertida **aumenta** à medida que **mais blocos são adicionados após o bloco da transação**. À medida que a cadeia cresce, o bloco contendo a transação fica mais profundo, o que **aumenta a segurança** de que ela não será revertida. Embora esse método tenha funcionado e resistido ao tempo por muitos anos, ele é **bastante lento**. Por exemplo, na rede Bitcoin, os usuários geralmente **precisam esperar seis blocos** (cerca de **uma hora**) para obter um **alto nível de confiança** na finalidade da transação. Esse tipo de finalidade pode ser aceitável em blockchains públicos, mas **não é adequado** para **transações financeiras em blockchains consorciados**, onde é necessária **finalidade imediata**.
* A **finalidade determinística** ou **finalidade imediata** fornece uma **garantia absoluta** de que uma transação é final assim que for confirmada em um bloco. **Não há forks nem reversões** que possam desfazer uma transação. O nível de confiança na finalidade da transação é de **100%** assim que o bloco que a contém é finalizado. Esse tipo de finalidade é fornecido por **algoritmos tolerantes a falhas**, como o **PBFT**.

**Velocidade, desempenho e escalabilidade**

O **desempenho** é um fator significativo que impacta a escolha dos algoritmos de consenso. As cadeias baseadas em **PoW são mais lentas** do que as baseadas em **BFT**. Se o desempenho for um requisito crucial, então é aconselhável usar algoritmos baseados em **votação** para **blockchains permissionados**, como o PBFT, que fornecerá **melhor desempenho** em termos de **processamento mais rápido de transações**.

Contudo, há uma ressalva que precisa ser considerada: blockchains do tipo PBFT **não escalam bem**, embora ofereçam melhor desempenho. Por outro lado, cadeias do tipo PoW **escalam melhor**, mas são **mais lentas** e **não atendem aos requisitos de desempenho em nível corporativo**.

**Resumo**

Neste capítulo, explicamos alguns dos protocolos mais proeminentes em blockchain e consenso de sistemas distribuídos tradicionais. Cobrimos vários algoritmos, incluindo **PoW**, **proof of stake**, **protocolos BFT tradicionais** e os protocolos mais recentes, como o **HotStuff**. Não cobrimos algoritmos mais novos como **BABE**, **GRANDPA**, **proof of history** e **Casper**, que serão discutidos mais adiante neste livro.

O consenso distribuído é uma área de pesquisa muito interessante, e pesquisadores acadêmicos e da indústria participam ativamente desse campo empolgante. Também é uma área **profunda e vasta**; portanto, é impossível cobrir todos os protocolos e todas as dimensões dessa disciplina envolvente em um único capítulo. No entanto, os protocolos e ideias apresentados aqui oferecem uma **base sólida para pesquisa adicional** e uma **exploração mais profunda**.