PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAFT IGUALITÁRIO — ERAFT

ERICK PINTOR

Trabalho de Conclusão IV apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Fernando Luís Dotti

Porto Alegre
2023

RAFT IGUALITÁRIO — ERAFT

RESUMO

O problema do consenso é um dos desafios fundamentais da área de sistemas distribuídos. Nele, processos independentes devem concordar sobre um determinado valor, proposto entre eles, e que todos adotam. Com uma literatura rica que data do início dos anos 90, o problema do consenso tem sido explorado por diversos ângulos, nos quais sua complexidade é notável. O algoritmo Raft tenta prover uma solução de fácil implementação contando com crescente adoção na indústria. Devido à alta demanda por sistemas distribuídos em redes de longa distância, algoritmos de consenso generalizado e igualitários foram propostos para aumentar a vazão do consenso. Neste trabalho de conclusão de curso introduzimos um novo algoritmo chamado Raft Igualitário, ou Egalitarian Raft (ERaft), que se baseia em Raft para construir um algoritmo de consenso igualitário de fácil implementação.

Palavras-Chave: sistemas distribuídos, consenso, algoritmos de consenso, consenso igualitário, consenso sem líder.

LISTA DE FIGURAS

2.1 Troca de mensagens em uma instância do algoritmo Paxos. 12
2.2 Diagrama de transição de estados do algoritmo Raft. 13
2.3 Troca de mensagens de consenso igualitário com conflito. 16
3.1 Modelo de replicação do *log*. 19
3.2 Diagrama de estados de processos no algoritmo ERaft. 19
3.3 Diagrama de transição de fases de um comando em ERaft. 20
3.4 Exemplo de ciclos de dependências. 20

LISTA DE ALGORITMOS

3.1 Algoritmo que define o estado inicial de um processo *p*. 22
3.2 Algoritmo que adiciona um novo comando ao *log* do líder. 22
3.3 Algorítimo acionado para a um evento do tipo *timeout*. 23
3.4 Algoritmo que envia comandos do líder para os seguidores. 23
3.5 Algoritmo utilizado pelo seguidor ao receber um comando do líder. 24
3.6 Algoritmo utilizado pelo líder ao receber uma resposta do seguidor. 24
3.7 Algoritmo utilizado pelo líder avalia o progresso do consenso. 25
3.8 Algoritmo utilizado para consolidar logs coletados durante uma eleição. . . . 27

SUMÁRIO

1 INTRODUÇÃO **6**

2 CONTEXTUALIZAÇÃO **9**

2.1 CONSENSO CLÁSSICO 10

2.1.1 ALGORITMO PAXOS 10

2.1.2 ALGORITMO RAFT 12

2.2 CONSENSO IGUALITÁRIO 14

3 RAFT IGUALITÁRIO **17**

3.1 MOTIVAÇÃO 17

3.2 MODELO DO SISTEMA 17

3.3 ALGORITMO 18

3.3.1 IDEIA DO ALGORITMO 18

3.3.2 REPLICAÇÃO DE COMANDOS 21

3.3.3 RECUPERAÇÃO DE FALHAS 26

3.4 INVARIANTES E PROPRIEDADES 28

3.5 CONSIDERAÇÕES PRÁTICAS 28

4 CONCLUSÃO **30**

REFERÊNCIAS **31**

APÊNDICE A – Especificação em TLA$^+$ **33**

1. INTRODUÇÃO

Arquiteturas tradicionais de desenvolvimento de *software* têm sido desafiadas pela crescente demanda por interconectividade. Com a necessidade cada vez mais frequente de expandir seus negócios globalmente, departamentos de tecnologia são forçados a projetar aplicações de alta escalabilidade e tolerantes a falhas. Dentre os vários desafios inerentes a esta categoria de aplicações encontra-se o problema do consenso: como garantir que processos distribuídos e assíncronos concordem sobre um determinado valor proposto. Este é um problema fundamental instanciado nos mais diversos cenários de interesse da Computação Distribuída.

De maneira informal, o consenso consiste em um grupo finito de processos concordarem sobre um valor, proposto entre eles, e que todos adotam. Ou seja, ele representa a capacidade de processos tomarem decisões conjuntas. O problema do consenso deixa de ser trivial quando supomos a possibilidade de falha e que o sistema é assíncrono — nada se pode supor sobre os atrasos de mensagens trocadas e a velocidade relativa dos processos [4]. Visto que a possibilidade de falhas deve inevitavelmente ser considerada em sistemas distribuídos e que na maior parte das vezes o meio de comunicação (rede) apresenta latências não-determinísticas, o estudo e avanço de soluções para o problema do consenso foi e permanece de suma importância no cenário atual de grande conectividade e dependência de sistemas corretos altamente disponíveis.

O trabalho seminal de Leslie Lamport, de 1990, na criação do algoritmo de consenso Paxos [7] (publicado em 1998 e republicado em 2019) deu início a uma nova área de pesquisa que contribui para a criação de sistemas distribuídos tolerantes a falhas. No consenso clássico, existe um processo, normalmente denominado líder, responsável por coordenar as tomadas de decisão que levam ao consenso. Em caso de falha do líder, os demais processos devem eleger um novo processo como líder antes de prosseguir. Intuitivamente, a vazão de tais sistemas é limitada pela capacidade do líder em coordenar o consenso. Ainda, valores propostos devem ser enviados para o líder, o que impõe acréscimo de atraso no tratamento de requisições dos clientes. Esta penalidade aumenta conforme a distribuição geográfica de processos através de redes de longa distância.

Na tentativa de aumentar a vazão em sistemas baseados em consenso, surge na literatura o conceito de consenso generalizado ([14] e [9]). Ao contrário do consenso clássico, que admite que apenas um valor — ou comando — seja coordenado por vez de modo a estabelecer uma ordem total de comandos a serem executados por todos os participantes, o consenso generalizado permite que comandos não-conflitantes sejam coordenados paralelamente. Comandos não-conflitantes são aqueles que do ponto de vista da aplicação podem comutar, ou seja, podem ser executados em qualquer ordem sem alterar o estado final da aplicação. Desta forma, o consenso generalizado adota uma relação de ordem par-

cial entre comandos propostos para permitir maior vazão em cargas de trabalho com baixas taxas de conflito entre comandos.

O próximo passo na evolução da literatura chega em 2013 com o algoritmo Egalitarian Paxos (EPaxos) [12], que introduz o conceito de consenso igualitário. Baseando-se no consenso generalizado, EPaxos permite todos os processos participarem de todas as instâncias de consenso de forma concorrente, aumentando significativamente sua vazão para cargas de trabalho com altas taxas de conflito. Em 2020, o algoritmo Atlas [3], similar a EPaxos, foi publicado trazendo ainda maiores avanços para a vazão neste tipo de sistema.

Outra perspectiva acerca de algoritmos de consenso, tal como documentado pela pesquisa que levou ao desenvolvimento do algoritmo Raft [13], é que a dificuldade de compreensão destes algoritmos representa uma barreira para sua adoção. Na prática, esta dificuldade se traduz em implementações sobre as quais se tem menor confiabilidade acerca de suas propriedades. Sendo assim, Raft foi criado com o intuito de prover um algoritmo legível, de maior facilidade de compreensão, o que resulta em maior probabilidade de implementações aderentes à especificação do protocolo. Como resultado, ele conta hoje com uma rica base de implementações e crescente adoção pela indústria [1]. No entanto, Raft permanece na classe de algoritmos de consenso clássicos com liderança forte, não se beneficiando, assim, dos avanços de vazão e latência alcançados pelos algoritmos de consenso generalizados e igualitários.

Apesar das vantagens descritas na literatura de consenso generalizado e igualitário, tais algoritmos tendem a adicionar funções e regras a algoritmos de complexidade já elevada em termos de compreensão. Não é de nosso conhecimento a tentativa de derivar um algoritmo desta classe, focado na legibilidade de sua especificação, aos moldes de Raft. Supõe-se que tal algoritmo possa acelerar o uso da abordagem de consenso igualitário, na prática, de forma análoga à crescente adoção de Raft. Logo, neste trabalho de conclusão de curso, propomos a pesquisa e o desenvolvimento de um algoritmo de consenso igualitário seguindo os mesmos princípios de projeto deste algoritmo: fácil compreensão e implementação.

Assim, o presente trabalho apresenta as seguintes contribuições:

- Proposta de um algoritmo de consenso igualitário a partir dos princípios de modelagem de Raft;
- Agregação dos principais aspectos de consenso igualitário a partir dos princípios do protocolo Atlas;
- Adequação do funcionamento em fases, do Atlas, para o modelo de replicação de comandos do Raft;

[1] Algumas implementações de código aberto de Raft (https://raft.github.io/#implementations) são parte de outros sistemas como, por exemplo, Etcd e RethinkDB.

- Proposta de um procedimento recuperação de comandos em caso de falha de processos;
- Especificação do algoritmo proposto utilizando a linguagem formal *Temporal Logic of Actions* [2] (TLA$^+$).

O capítulo 2 deste trabalho contextualiza o problema do consenso, oferecendo maiores detalhes sobre o consenso clássico e igualitário. O capítulo 3 descreve algoritmo proposto em relação à sua especificação formal incluída no apêndice A. Considerações finais são feitas no capítulo 4.

[2] https://lamport.azurewebsites.net/tla/tla.html

2. CONTEXTUALIZAÇÃO

O problema do consenso pode ser visto como a capacidade de múltiplos processos de concordarem e adotarem um valor proposto entre eles. Este problema se instancia nas mais diversas áreas da Computação Distribuída, como bancos de dados distribuídos [1], sincronização de configurações em parques de máquinas [2] e orquestradores de ambientes [3] [6].

Para ilustrar uma instância possível do problema do consenso, considere o seguinte exemplo: uma instituição financeira oferece saques e depósitos para clientes ao redor do mundo. Suponha dois clientes com acesso à mesma conta bancária, cujo saldo é de R$100,00 reais. Imagine que ambos os clientes façam, simultaneamente, a solicitação de um saque no valor total do saldo da conta. Intuitivamente, o comportamento esperado é de que apenas um cliente consiga sacar o valor solicitado e que o outro receba uma mensagem informando não haver saldo suficiente para executar o seu pedido. Em uma aplicação não-distribuída, este é um problema trivial, visto que um único processo é responsável por responder às duas solicitações, conseguindo decidir a ordem de execução: autorização da primeira e rejeição da segunda. Por outro lado, em um sistema distribuído, é possível que cada solicitação de saque seja recebida por um processo diferente. Para garantir o mesmo nível de consistência da solução anterior, estes processos devem entrar em consenso sobre quais solicitações estão sendo executadas naquele momento e, principalmente, em que ordem.

Ao generalizar o problema do consenso, obtemos uma classe de algoritmos aplicável a diversos domínios. De maneira genérica, temos que um algoritmo de consenso é projetado para garantir que um conjunto finito de processos distribuídos e assíncronos concordem e adotem um determinado valor proposto — quais saques executar e em qual ordem. Para isso, processos trocam mensagens através de um meio de comunicação (rede). Uma vez determinado o consenso, todos os processos envolvidos devem adotar a decisão tomada, esta sendo imutável. Para considerar um algoritmo de consenso correto, ele deve seguir as seguintes propriedades:

- Terminação: todo processo correto adotará a decisão do consenso em algum momento;
- Validade: se um processo adota um valor v, então v foi proposto por algum processo;
- Integridade: um processo não adota mais de um valor;
- Acordo: todo processo correto deve concordar sobre o mesmo valor.

[1] Fauna (fauna.com), HBase (hbase.apache.org) e MongoDB (mongodb.com).
[2] Zookeeper (zookeeper.apache.org.), Consul (www.consul.io) e Etcd (coreos.com/etcd).
[3] Kubernetes (kubernetes.io), Docker Swarm (github.com/docker/swarm) e Mesos (mesos.apache.org).

Existem diferentes classes de falhas que dificultam preservar tais propriedades, dentre elas, falhas bizantinas tendem a ser particularmente desafiadoras. Em um sistema suscetível a falhas bizantinas, não é possível supor que os processos participantes são corretos. Um processo malicioso pode, intencionalmente, enviar mensagens contraditórias sobre os demais processos — como dizer ter aceitado um saque de R$300,00 reais mesmo com saldo insuficiente, por exemplo. Este é um problema comum em sistemas de *blockchain* nos quais os participantes do consenso podem não ser entidades confiáveis.

Neste trabalho de conclusão de curso focaremos apenas em algoritmos que toleram falhas não-bizantinas, dentre elas a interrupção abrupta de um processo e sua recuperação (modelo *crash-recovery*), além de falhas de comunicação entre processos. Em ambos os casos, para preservar as propriedades do consenso, deve haver uma etapa de recuperação de um processo falho de forma que o mesmo consiga estar ciente de decisões tomadas durante sua falha. É comum a este tipo de algoritmo suportar a falha de uma minoria de processos sem comprometer a disponibilidade do serviço prestado. No entanto, no caso de falhas catastróficas, nas quais a maioria dos processos falha, ainda se faz necessário preservar as propriedades de segurança e funcionamento (*safety*), garantindo a consistência dos dados do sistema, enquanto a terminação (*liveness*) é violada.

Visto o problema do consenso, nas seções seguintes são apresentados algoritmos de consenso clássicos (seção 2.1) e igualitários (seção 2.2).

2.1 Consenso Clássico

2.1.1 Algoritmo Paxos

Dentre os algoritmos de consenso clássicos, Paxos [7] tem notória importância devido à sua contribuição histórica para a área de sistemas distribuídos. Leslie Lamport estabelece nos anos 90 muitos dos fundamentos utilizados até hoje em algoritmos similares. Em Paxos, um processo líder, também chamado *proposer*, é responsável por coordenar uma instância do algoritmo de consenso através de duas trocas de mensagens com os demais processos, denominados *acceptors* e *learners*. *Acceptors* são processos que participam ativamente do consenso enquanto *learners* apenas recebem, ou aprendem, o resultado do consenso. Paxos é dividido em duas fases, normalmente subdivididas em duas etapas cada:

- Fase 1a:
  - *Prepare*(*b*): mensagem enviada pelo *proposer* para *acceptors* onde *b* é o identificador da rodada de troca de mensagens para a instância do consenso que está

sendo executada — Paxos permite múltiplas tentativas, ou rodadas, até que o consenso seja alcançado;

- Fase 1b:
  - *Promise*(b, v): enviada de *acceptors* para *proposer* em resposta a uma mensagem do tipo *prepare*. Nela, v pode ser nulo se b é desconhecido, ou o valor anteriormente aceito caso contrário;
- Fase 2a:
  - *Accept*(b, v): enviada pelo *proposer* para *acceptors* após receber um quórum de respostas do tipo *promise*;
- Fase 2b:
  - *Accepted*(b, v) enviada de *acceptors* para o *proposer* e *learners* em resposta a uma mensagem do tipo *accept*.

A figura 2.1 demonstra uma troca de mensagens possível para uma instância de consenso do algoritmo Paxos. Note que, em um ambiente distribuído e assíncrono, quaisquer destas mensagens podem se perder devido à falha do processo emissor, receptor, ou por falha de rede.

Para preservar as propriedades descritas na seção 2, Paxos emprega uma série de regras derivadas do número identificador b e os valores v trocados entre processos. A sobreposição de estados possíveis se torna, especialmente na presença de falhas, um fator de complexidade não-trivial que dificulta a compreensão e reprodução do algoritmo de forma correta.

Múltiplos refinamentos ao algoritmo Paxos surgiram ao longo dos anos tanto na forma do consenso clássico quanto no consenso generalizado. Dentre eles se destacam:

- Consenso clássico:
  - *Paxos Made Simple* [8]
  - *Cheap Paxos* [11]
  - *Paxos Made Moderately Complex* [16]
- Consenso Generalizado:
  - *Generalized Consensus and Paxos* [9]
  - *Fast Paxos* [10]
  - *On Making Generalized Paxos Practical* [15]

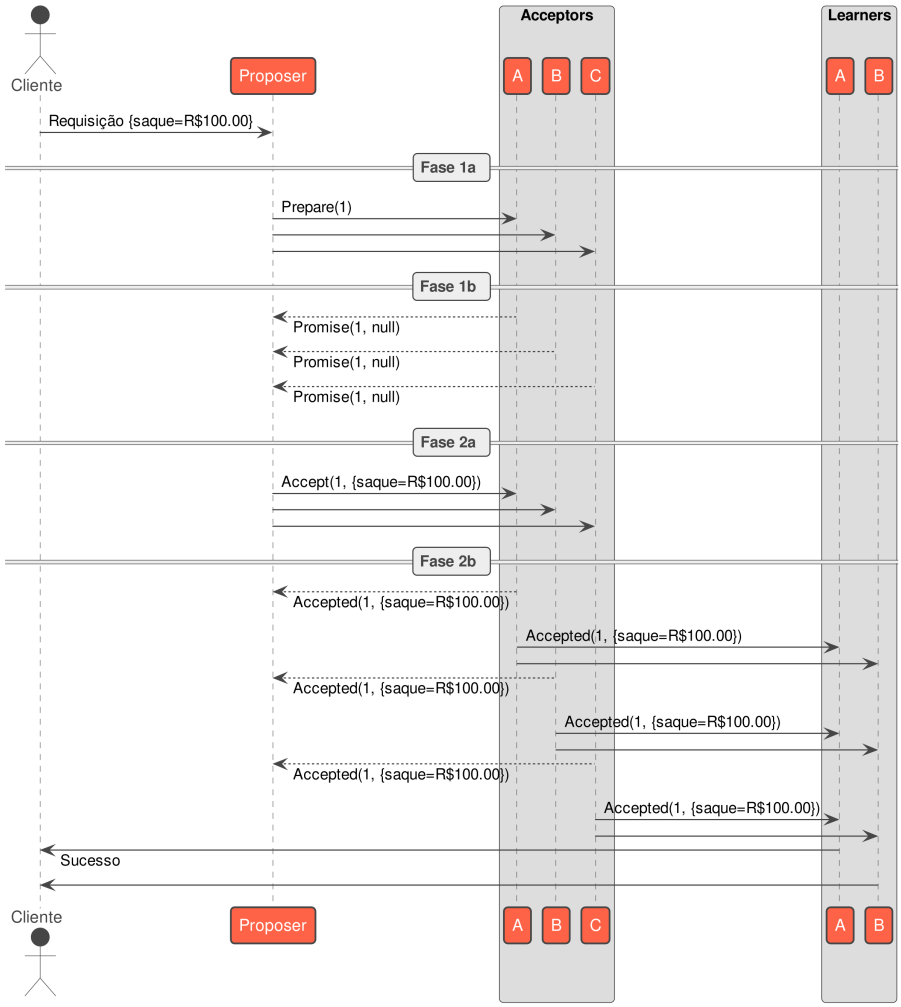


Figura 2.1: Troca de mensagens em uma instância do algoritmo Paxos.

Paxos continua sendo um dos algoritmos fundamentais da área de sistemas distribuídos. A partir dele, surgem os princípios e propriedades básicas que permitem a construção de algoritmos de consenso consistentes e tolerantes a falhas. No entanto, sua complexidade inspirou pesquisadores a buscar novas formas de projetar algoritmos de consenso, como o algoritmo Raft (seção 2.1.2).

2.1.2 Algoritmo Raft

Raft [13] é um algoritmo de consenso focado em legibilidade e fácil implementação. Ele surge a partir da frustração de seus autores em compreender e reproduzir variações do algoritmo Paxos. Em sua pesquisa, os autores mencionam como até mesmo especialistas na área de sistemas distribuídos tem, por vezes, dificuldade em explicar em detalhes como Paxos resolve o problema do consenso na presença de falhas. Ainda, o nível de abstração em que Paxos é descrito deixa a cargo do programador resolver questões práticas, porém não triviais, de sua implementação. Por exemplo, é comum em sistemas que precisam re-

solver consenso, o registro (*log*) de decisões passadas para fins de recuperação de falhas. No entanto, Paxos não deixa claro como este *log* deve ser mantido ou quando é seguro remover registros antigos dele.

De modo a prover um algoritmo de consenso de fácil compreensão, Raft toma decisões de projeto pragmáticas com foco na clareza de sua especificação. Como resultado, implementações de Raft tendem a ser mais próximas de sua descrição, dado que o programador tem pouca ou nenhuma questão prática de implementação que não está coberta pela descrição do algoritmo.

Em alto nível, cada processo do Raft pode se encontrar em um dos três estados: “seguidor”, “candidato”, ou “líder”. Um **seguidor** se torna **candidato** se não receber mensagens do **líder** em um determinado período. Ao se tornar candidato, ele promove uma eleição na tentativa de se tornar o líder, ou volta a ser um seguidor caso mal sucedido. A figura 2.2 demonstra as transições possíveis de estado de um processo Raft.

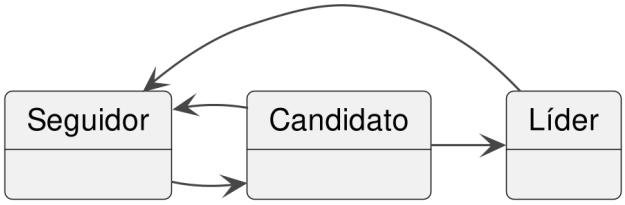


Figura 2.2: Diagrama de transição de estados do algoritmo Raft.

Líderes recebem valores (comandos) de clientes e os adicionam a seu *log*. Periodicamente, o líder envia comandos pendentes para os seguidores, os quais os adicionam a seu *log*. Assim que um comando tenha sido replicado para a maioria dos seguidores, o líder registra o seu índice como persistido (*committed*). Comandos persistidos são executados por todos os processos na ordem do *log* e podem ser removidos se todos os processos já o executaram. Raft emprega apenas um RPC (*Remote Procedure Call*) para replicar comandos entre líder e seguidores:

$$Append(term, leaderID, prevLogIndex, entires, leaderCommit) \rightarrow (term, success)$$

Note que, ao replicar comandos via *Append*, líder e seguidores trocam toda informação necessária para as tomadas de decisão do algoritmo. Sendo elas:

- Requisição:
  - *term*: identificador único incrementado em cada eleição;
  - *leaderID*: identificador do líder associado ao *term* atual;
  - *prevLogIndex*: posição anterior do *log* onde *entries* se inicia;

  - *entries*: quais comandos devem ser adicionados no *log*;
  - *leaderCommit*: qual o maior índice dentre os comandos já persistidos.
- Resposta:
  - *term*: o termo atual do seguidor;
  - *success*: indica se a requisição foi bem sucedida.

Essa estratégia diminui a complexidade do algoritmo visto que a sobreposição possível de estados durante falhas diminui. Ainda, esta decisão de projeto fornece um modelo mental intuitivo para quem implementa o algoritmo: repita o RPC *Append* enquanto o processo for líder; caso contrário, volte ao estado de seguidor.

É importante ressaltar que legibilidade e facilidade de implementação são critérios subjetivos na avaliação de algoritmos. No entanto, Raft traz com sua publicação uma pesquisa qualitativa que sugere uma real vantagem de Raft sobre Paxos na capacidade de programadores não familiarizados em seguirem suas especificações e implementarem uma versão correta de cada algoritmo.

2.2 Consenso Igualitário

O papel de um processo líder em algoritmos de consenso clássicos simplifica o modelo e facilita preservar as propriedades do consenso. Entretanto, ele se torna um fator limitante no desempenho desses algoritmos, principalmente em sistemas distribuídos em longas distâncias geográficas. Em tais sistemas, mesmo que haja um processo geograficamente próximo ao cliente, o mesmo deve enviar seus comandos para o líder, que pode estar localizado em outra região geográfica, causando aumento do tempo de resposta devido à latência de rede.

Alternativas foram propostas para diminuir o papel do líder e aumentar a vazão. Dentre elas, Mencius [1] propõe um algoritmo baseado em Paxos no qual o líder de cada instância de consenso é escolhido deterministicamente de maneira a rotacionar este papel entre os processos participantes. Baseando-se no Paxos Generalizado [9], Mencius consegue concluir instâncias do consenso paralelamente. Este algoritmo promove, então, o balanceamento de carga ao distribuir o papel de liderança entre processos. No entanto, ele ainda é suscetível a alta latência em sistemas geograficamente distribuídos.

O surgimento de Egalitarian Paxos (EPaxos) [12] traz um marco importante na busca por algoritmos de consenso para sistemas distribuídos em redes de longa distância. Esta versão de Paxos permite que cada processo seja líder dos comandos que recebe de seus clientes, reduzindo assim a latência do sistema dado que clientes podem se comunicar com o processo localizado mais próximo de si. Surge, então, uma nova classe de algoritmos

chamados sem-líder (*leaderless*), apesar de o termo “igualitário” ser mais apropriado uma vez que todo processo é líder de seus próprios comandos.

Posteriormente, o artigo “Leaderless State-Machine Replication: Specification, Properties, Limits” [5] reflete sobre esta classe de algoritmos e os generaliza de modo a estabelecer seus componentes e propriedades fundamentais. Conclui-se que algoritmos igualitários requerem uma função de conflito. Dois comandos são ditos conflitantes se, quando executados em diferentes ordens relativas, produzem resultados diferentes. Uma definição mais concreta de conflito estabelece que dois comandos são conflitantes se um escreve no mesmo conjunto de dados acessado (lidos ou escritos) pelo outro comando. Por exemplo, um saque de R$100,00 reais e um depósito de R$100,00 reais executados na mesma conta bancária com saldo atual igual a zero produzem resultados diferentes para o cliente se executados em ordens diferentes.

A primeira etapa de um algoritmo de consenso igualitário é coletar os conflitos de seus comandos pendentes, etapa conhecida como caminho rápido (*fast path*). Comandos sem conflitos são persistidos e executados após esta fase. Comandos conflitantes são promovidos para o chamado caminho lento (*slow path*), o qual requer uma nova troca de mensagens para garantir que todos os processos tomem conhecimento de todos os conflitos para estes comandos. Somente ao fim do caminho lento, comandos conflitantes podem ser persistidos e executados para garantir que todos os processos sigam a mesma ordem relativa entre comandos conflitantes. A figura 2.3 demonstra, em alto nível, uma troca de mensagens possível de consenso igualitário com a presença de conflito.

EPaxos emprega função de conflito, caminhos rápidos e lentos. No entanto, a chegada do algoritmo Atlas [3] traz uma contribuição significativa para a área: ao contrário de EPaxos que requer o caminho lento para quaisquer comandos conflitantes, Atlas é capaz de pular esta etapa se determinado que um quórum de processos tem conhecimento dos mesmos conflitos. Com esta otimização, Atlas demonstrou maior desempenho em avaliação experimental tanto sobre EPaxos como sobre Mencius.

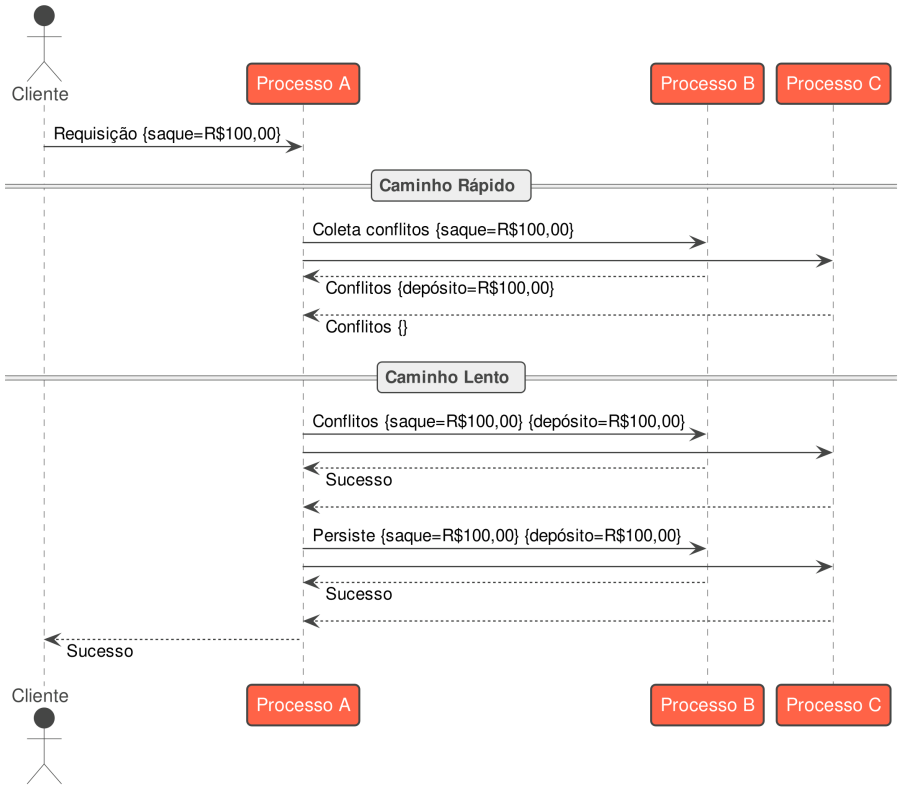


Figura 2.3: Troca de mensagens de consenso igualitário com conflito.

3. RAFT IGUALITÁRIO

3.1 Motivação

Ao analisar as diferentes formas de consenso descritas na literatura, é possível observar a evolução desta área de pesquisa buscando aumentar sua vazão principalmente em sistemas geograficamente distribuídos em redes de longa distância. Todavia, em seu estado atual, Raft permanece incapaz de se beneficiar desta evolução, já que este é um algoritmo clássico com forte papel do processo líder na coordenação do consenso.

Por acreditar que o modelo estabelecido por Raft possa ser adaptado para a classe de algoritmos de consenso igualitário ainda preservando a legibilidade de sua especificação, neste trabalho de conclusão de curso propomos um novo algoritmo de consenso chamado Raft Igualitário, ou Egalitarian Raft (ERaft).

Apesar das mudanças necessárias para adaptar o modelo proposto por Raft ao consenso igualitário, presumimos que, ao adotar Raft como algoritmo base, temos uma plataforma consolidada de fácil compreensão, permitindo tais refinamentos com pouco acréscimo de complexidade. Ademais, o Raft, com todo o seu material disponível, serve como introdução para ERaft da mesma forma que Paxos estabelece a base fundamental para compreender Paxos Generalizado e Paxos Igualitário. Por fim, acreditamos que o modelo criado por Raft pode evoluir e se beneficiar dos avanços documentados na literatura de consenso igualitário com a introdução de ERaft, sendo este possível propulsor da adoção de consenso igualitário pela indústria.

3.2 Modelo do Sistema

Assumimos um sistema distribuído composto por processos interligados. Existe um conjunto ilimitado $C = \{c_1, c_2, ...\}$ de processos clientes e um conjunto limitado $S = \{s_1, s_2, ..., s_n\}$ de processos servidores.

O sistema pode se comportar de forma assíncrona, ou seja, não há limites para atrasos de mensagens e velocidades relativas de processo. Entretanto, para garantia de progresso supõe-se que em algum momento o sistema passa a se comportar de forma síncrona. Este momento é chamado de Tempo Global de Estabilização [2] (GST - *Global Stabilization Time*), sendo desconhecido pelos processos.

Assumimos o modelo de falha de colapso (*crash*) e excluímos comportamentos maliciosos e arbitrários, ou seja, sem falhas bizantinas. O processo pode falhar e se recuperar.

Os processos se comunicam somente por troca de mensagens, usando comunicação um-para-um. Não há memória compartilhada. A comunicação é através das primitivas *send*(*m*) e *receive*(*m*), onde *m* é uma mensagem. Se um remetente enviar uma mensagem várias vezes, um destinatário correto acabará recebendo a mensagem.

3.3 Algoritmo

O algoritmo ERaft desenvolvido neste trabalho foi modelado utilizando a linguagem de especificação formal TLA$^+$ [1] (apêndice A). Sua especificação oferece uma definição precisa para o funcionamento do algoritmo quanto à replicação de comandos e recuperação em caso de falhas. O restante deste capítulo se dedica a apresentar em alto nível o funcionamento do algoritmo de modo a enriquecer o entendimento de sua especificação.

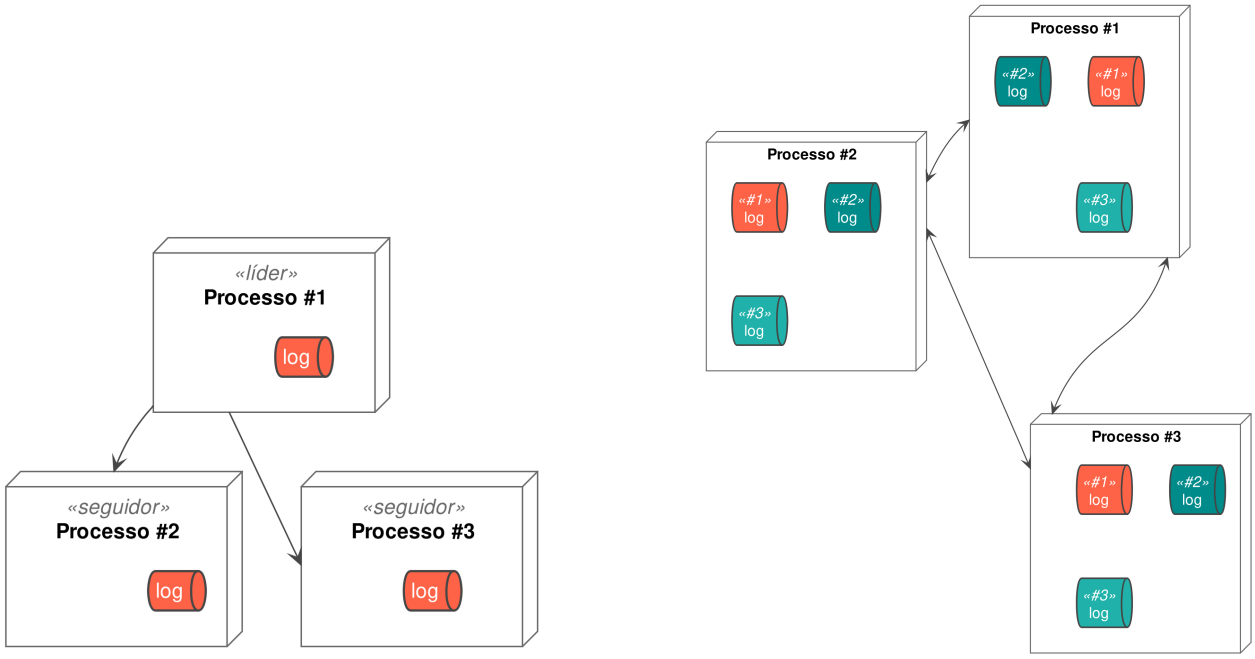
3.3.1 Ideia do Algoritmo

Imagine um sistema com um único processo que implementa o algoritmo Raft. Este processo possui um estado local composto por seus comandos (*log*) e seus índices de replicação, *commit*, e execução. Dado que há apenas um processo, temos que o mesmo será o líder do consenso. Em uma implementação clássica de Raft, ao adicionar novos processos ao sistema, os mesmos se tornam seguidores do líder vigente e passam a replicar o seu *log* (Figura. 3.1a). Já em ERaft, novos processos multiplicam o estado: cada processo tem o seu próprio *log* e replica o *log* dos demais (Figura. 3.1b).

Em Raft, intervalos de tempo fechados (*timeouts*) são utilizados como mecanismo de tomada de decisão nos processos participantes. Ao iniciar o algoritmo, cada processo começa uma contagem de tempo que, ao término de um intervalo predeterminado, dispara uma tomada de decisão antes de iniciar o próximo intervalo de tempo. ERaft utiliza a mesma abordagem, porém, para suportar múltiplos *logs*, é necessário também manter múltiplos *timeouts*, um por *log*.

Assim como em Raft, todos os processos em ERaft iniciam no estado “seguidor”. Ao término de um ciclo de *timeout*, um “seguidor” passa para o estado de “candidato” e inicia uma eleição na tentativa de se tornar líder de seu próprio *log*. Após o término da eleição, em uma execução sem falhas, o processo candidato torna-se líder e passa a receber comandos dos clientes. ERaft traz uma modificação no diagrama de estados de Raft (Figura. 2.2) na qual, por motivos de recuperação de falhas, é permitido que líderes passem para o estado de candidatos sem antes passar para o estado de seguidores (Figura. 3.2) — maiores detalhes são discutidos na seção 3.3.2.

[1] https://lamport.azurewebsites.net/tla/tla.html



(a) Raft (b) ERaft

Figura 3.1: Modelo de replicação do *log*. À esquerda, Raft com um único *log* cujo líder é responsável pela replicação para os seguidores. À direita, o algoritmo ERaft, em que cada processo replica o seu *log* para os demais.

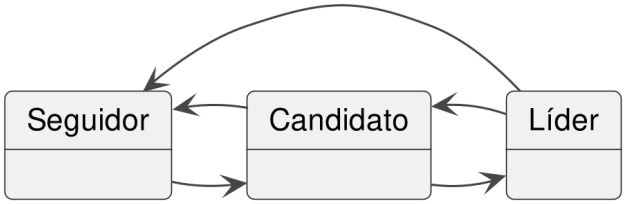


Figura 3.2: Diagrama de estados de processos no algoritmo ERaft.

Em uma aplicação na qual os comandos não conflitam, o *log* replicado de cada processo pode ser executado em qualquer ordem. No entanto, como comandos conflitantes podem ser submetidos a qualquer processo, é necessário que haja um mecanismo de sincronização entre os mesmos para estabelecer uma ordem parcial de execução que garanta a ordem entre comandos conflitantes e assim a consistência dos dados da aplicação. Para isso, ERaft adapta os protocolos de troca de mensagens de Raft para a coleta e replicação de conflitos — também chamados dependências. O consenso garante que um comando decidido carregará as mesmas dependências em todas as réplicas. A fase de execução deve garantir que um comando só execute se suas dependências foram satisfeitas (já executadas).

Da mesma forma que Atlas, comandos no *log* são classificados nas fases: “rápida”, “lenta”, e “concluída” (fig. 3.3). Todo comando recebido por um líder tem seu conjunto inicial de dependências calculado e adicionado ao *log* na fase **rápida**. Comandos nesta fase

são enviados a um subgrupo de processos denominado quórum rápido, os quais calculam sua contribuição para o conjunto de dependências e as retornam ao líder. Ao receber um número suficiente de respostas, o líder decide com base nas dependências recebidas se é seguro promover o comando para fase **concluída**, comunicando assim seus pares sobre esta decisão. Caso não seja seguro concluir o consenso, o comando tem seu conjunto de dependências atualizado e promovido para a fase **lenta**, na qual mais uma rodada de trocas de mensagens ocorre antes de promovê-lo para a fase **concluída**.

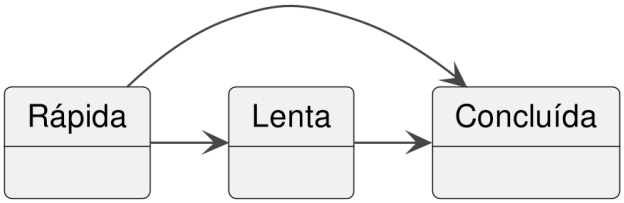


Figura 3.3: Diagrama de transição de fases de um comando em ERaft.

Eventualmente cada processo passa a conter uma cópia do *log* de cada um de seus pares. Comandos na fase concluída podem ser executados a qualquer momento desde que executados após suas dependências. Na prática, isso gera um grafo cíclico dirigido, o qual cada processo deve navegar de forma determinística (fig. 3.4). Comandos executados são, por fim, removidos do *log*.

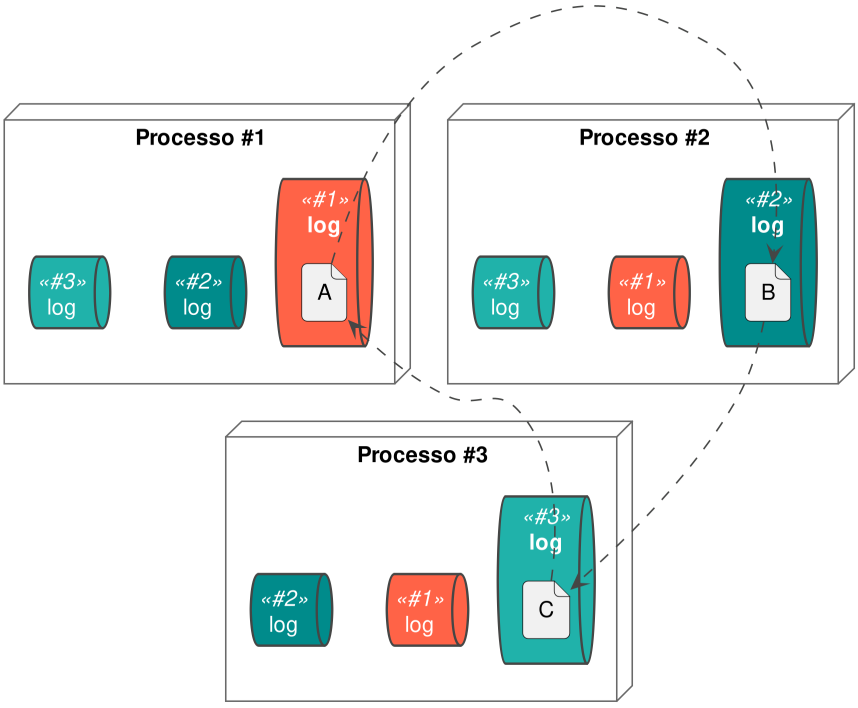


Figura 3.4: Exemplo de ciclos de dependências: comando “A” coordenado pelo processo 1 depende do comando “B” coordenado pelo processo 2 que, por sua vez, depende do comando “C” coordenado pelo processo 3, o qual volta a depender do comando A”.

A recuperação de falhas em algoritmos de consenso é um fator de notável complexidade. Em algoritmos igualitários, é possível que um processo tenha informado seus pares quanto a dependências para seus próprios comandos, os quais não são promovidos para a fase concluída antes de sua falha. Neste caso, execução dos comandos dependentes nos demais processos é comprometida.

Se, ao término de um ciclo de *timeout* um processo não observar mensagens de um determinado par, ele suspeita da falha do mesmo. Caso o mesmo tenha uma dependência não concluída para o par suspeito, ele passa a ser candidato a uma eleição para o *log* deste par. Durante o processo de eleição, o candidato recebe dos demais processos os comandos conhecidos onde o suspeito é líder. Se eleito, o processo conclui o consenso dos comandos recebidos, o que permite que comandos dependentes possam ser executados.

As secções 3.3.2 e 3.3.3 oferecem uma visão em alto nível dos algoritmos de replicação de comandos e recuperação de falhas. Note que os pseudocódigos presentes neste capítulo são um subconjunto mínimo do comportamento do algoritmo, oferecendo uma noção geral de seu funcionamento. A definição precisa dos conceitos aqui introduzidos se encontram na especificação formal no apêndice A.

3.3.2 Replicação de Comandos

ERaft é configurado com base em 3 constantes, sendo elas: um conjunto finito e não vazio de processos conhecidos; uma função booliana e comutativa de conflito definida pela aplicação; e o número f de falhas toleradas pelo sistema (apêndice. A, linhas 113–176). Ainda, de acordo com Atlas, o número de falhas f deve obedecer à relação $1 \leq f \leq \lfloor \frac{n-1}{2} \rfloor$ sendo n o número de processos.

Cada processo possui o estado inicial descrito no algoritmo 3.1. De forma análoga ao estado inicial de Raft, o termo atual para cada processo inicia em zero indicando que não houve eleições de líderes até o momento. Todo processo começa no estado de seguidor e com o *log* vazio. Os índices de replicação de comandos em fase rápida, lenta, e concluída iniciam em zero e o índice do próximo comando a ser enviado para seus pares inicia em 1, dado que não houve comandos submetidos até o momento. Note que, por haver uma sobreposição de estados, todo participante do algoritmo replica o *log* de todos os processos disponíveis e a mesma rotina de inicialização é invocada para cada processo conhecido.

Todo participante do algoritmo escolhe um quórum de processos para coordenar o consenso de comandos na fase rápida. Ao contrário do Atlas, que escolhe um quórum por comando, ERaft escolhe um quórum por processo, simplificando seu algoritmo de recuperação de falhas. Além disso, em conformidade com o Atlas, sua cardinalidade é de $\lfloor \frac{n}{2} \rfloor + f$. A definição formal da escolha de quóruns se encontra no apêndice A (linhas 416–423), sendo equivalente à equação 3.1.

Require: *Servers* ▷ Processos conhecidos
  procedure INIT(*p*)
    $p.currTerm \leftarrow 0$ ▷ Termo atual
    $p.currState \leftarrow follower$ ▷ Estado atual
    $p.log \leftarrow \langle\rangle$ ▷ Sequência ordenada de comandos
    $p.commitIdx \leftarrow 0$ ▷ Prefixo de comandos concluídos

    $p.fastMatchIdx \leftarrow [s \mapsto 0 \mid s \in Servers]$ ▷ Índice de comandos em fase rápida
    $p.slowMatchIdx \leftarrow [s \mapsto 0 \mid s \in Servers]$ ▷ Índice de comandos em fase lenta
    $p.nextIdx \leftarrow [s \mapsto 1 \mid s \in Servers]$ ▷ Próximo comando a ser enviado a seguidores

    $p.fastQuorum \leftarrow FastQuorum(p, Servers)$ ▷ Quórum usado para fase rápida

    $p.votedFor \leftarrow []$ ▷ Votos concedidos durante eleições
    $p.recovery \leftarrow []$ ▷ Estado da recuperação de falhas
  end procedure

Algoritmo 3.1: Algoritmo que define o estado inicial de um processo *p*.

$$FastQuorum(p, Servers) = Q \subseteq Servers : |Q| = \lfloor \frac{n}{2} \rfloor + f \wedge p \in Q \tag{3.1}$$

Uma vez definido seu estado inicial, o processo inicia um ciclo fechado de tempo (*timeout*) por processo conhecido. O ERaft é desenhado para funcionar em uma rotina de tratamento de eventos (*event loop*) no qual apenas um evento é processado por vez, o que garante a atomicidade das ações descritas na especificação formal. Os dois principais tipos de eventos são *AppendToLog*(*cmd*) e *Timeout*(*p*). Os demais se referem ao tratamento de mensagens de rede e serão discutidos posteriormente.

Ao receber um novo comando de um cliente, um evento do tipo *AppendToLog*(*cmd*) é disparado. Ao processá-lo, o líder calcula suas dependências e o adiciona ao seu *log* na fase rápida (algoritmo 3.2, e apêndice A nas linhas 713–729).

  procedure APPENDTOLOG(*cmd*)
    $Deps \leftarrow Dependencies(cmd)$ ▷ Calcula dependências
    $cmd \leftarrow (cmd \mapsto cmd, deps \mapsto Deps, phase \mapsto fast)$ ▷ Cria um novo registro
    $leader.log \leftarrow Append(leader.log, cmd)$ ▷ Adiciona no log
  end procedure

Algoritmo 3.2: Algoritmo que adiciona um novo comando ao *log* do líder.

A função de cálculo de dependências é descrita pela equação 3.2 — formalmente definida no apêndice A, linhas 607–627. Este algoritmo utiliza a função de conflito definida pela aplicação para identificar dependências entre o comando proposto e os demais comandos conhecidos.

$$Dependencies(c) = \bigcup \{c' \in leader.log, leader \in Servers \mid Conflict(c, c')\} \tag{3.2}$$

Quando um ciclo de tempo é concluído para um determinado processo, um evento do tipo *Timeout*(*p*) é gerado, o que dispara a rotina descrita pelo algoritmo 3.3. Nela, caso o processo seja líder, ele verifica se há comandos prontos para mudar de fase antes de disparar a rotina de envio de mensagens para seus pares. Caso contrário, ocorre a suspeita de falha do processo. Em ambos os casos, um novo ciclo de tempo se inicia.

```
procedure OnTimeout(p)
    if p.currState = leader then                    ▷ É líder do consenso
        CommitPending(p)                            ▷ Avança comandos pendentes
        ReplicateCommands(p)                        ▷ Envia mensagens aos seguidores
    else
        SuspectFailure(p)                           ▷ Suspeita de falha do processo
    end if
    ScheduleTimeout(p)                              ▷ Inicia próximo ciclo de tempo
end procedure
```

Algoritmo 3.3: Algorítimo acionado para a um evento do tipo *timeout*.

O algoritmo 3.4 descreve a rotina responsável por enviar comandos de um líder para seus seguidores — modelada formalmente no apêndice A, linhas 749–787. Assim como em Raft, um índice de replicação é utilizado para determinar quais comandos serão enviados para cada seguidor. No entanto, em conformidade com o Atlas, comandos em fase rápida são enviados apenas para seguidores que fazem parte do quórum rápido do líder. Mesmo que o conjunto de comandos a ser enviado seja vazio, o *RPC* de replicação ainda é invocado para que, assim como em Raft, seguidores possam reiniciar o *timeout* do líder, o que impede uma suspeita de falha por inatividade.

```
procedure ReplicateCommands(leader)
    for all follower ∈ Servers \ {leader} do        ▷ Para todo seguidor
        C ← ⟨cmd_i ∈ leader.log | i ≥ leader.nextIdx[follower]⟩   ▷ Comandos a enviar
        P ← C' ⊑ C | ∀c' : c'.phase = fast ⇒ follower ∈ leader.fastQuorum ▷ Prefixo de C
        AppendRPC(follower, leader.commtIdx, leader.nextIdx[follower] − 1, P)
    end for
end procedure
```

Algoritmo 3.4: Algoritmo que envia comandos do líder para os seguidores.

Cada *RPC* em ERaft é modelado por um evento de requisição e um de resposta. O algoritmo 3.5 descreve a rotina invocada pelo seguidor ao receber uma sequência de comandos do líder — modelada formalmente no apêndice A (linhas 789–912). Nela, assim como no algoritmo Atlas, o seguidor contribui para o consenso recalculando o conjunto de dependências de comandos na fase rápida à medida que os comandos são adicionados

ao *log* do seguidor. Comandos em fase lenta ou concluído são aceitos desde que não sobrescrevam comandos já concluídos e que o seguidor tenha conhecimento.

```
procedure OnAppendRequest(leader, follower, leaderCommit, prevLogIndex, Cmds)
    follower.commitIdx ← leaderCommit                          ▷ Aprende índice do líder

    for all cmd_i ∈ Cmds, cmd' ∈ follower.log_{prevLogIndex+i} do
        if cmd' = ∅ then                                       ▷ Não há comando neste posição
            if cmd.phase = fast then                           ▷ Comando recebido em fase rápida
                Deps ← Dependencies(cmd)                       ▷ Recalcula dependências
                cmd.deps ← cmd.deps ∪ Deps                     ▷ Atualiza comando
            end if
            follower.log ← Append(follower.log, cmd)           ▷ Aceita comando
        else                                                   ▷ Comando sobrescreve existente
            if cmd'.phase ≠ committed then                     ▷ Comando existente não concluído
                follower.log ← Replace(follower.log, cmd', cmd)  ▷ Aceita comando
            end if
        end if
    end for
                                                               ▷ Devolve dependências para o líder
    Deps ← ⟨cmd_i.deps ∈ follower.log | prevLogIndex < i ≤ |Cmds|⟩
    Reply(leader, prevLogIndex, Deps)
end procedure
```

Algoritmo 3.5: Algoritmo utilizado pelo seguidor ao receber um comando do líder.

Uma vez recebida a resposta de um seguidor, o líder utiliza a rotina descrita pelo algoritmo 3.6 para avançar seu estado. Nesta etapa, comandos na fase rápida acumulam as dependências reportadas pelo seguidor. Note que, assim como em Atlas, o objetivo da fase rápida é tanto replicar comandos quanto coletar dependências, enquanto a fase lenta tem o objetivo de comunicar aos seguidores as dependências consolidadas pelo líder. Portanto, este algoritmo avança o índice rápido para todo comando observado nesta fase, mas por outro lado, somente avança o índice lento se observados comandos nesta fase que possuam o mesmo conjunto de dependências.

```
procedure OnAppendReply(leader, follower, prevLogIndex, Deps)
    for all D_i ∈ Deps, cmd_{prevLogIndex+i} ∈ leader.log do
        if cmd.phase = fast then                               ▷ Comando na fase rápida
            cmd.deps ← cmd.deps ∪ D                            ▷ Acumula dependências
            leader.fastMatchIdx[follower] ← prevLogIndex + i   ▷ Atualiza o índice rápido
        else if cmd.phase = slow ∧ cmd.deps = D then           ▷ Mesmas dependências
            leader.slowMatchIdx[follower] ← prevLogIndex + i   ▷ Atualiza índice lento
        end if
    end for
end procedure
```

Algoritmo 3.6: Algoritmo utilizado pelo líder ao receber uma resposta do seguidor.

Para cada resposta recebida de um seguidor, o índice do próximo comando a ser enviado a ele é atualizado de acordo com seu estado. Caso o líder observe um comando em fase lenta onde o seguidor reportou um conjunto diferente de dependências, é utilizado o índice deste comando como o próximo a ser enviado a ele de modo a corrigir esta divergência. Caso o líder observe que todos os comandos processados estão em fase rápida, ele define o próximo comando como sendo $leader.fastMatchIdx[follower] + 1$, sinalizando que líder e seguidor estão sincronizados até este ponto (apêndice A, linhas 914–1064).

Em resposta a um evento do tipo *timeout*, líderes invocam a rotina descrita pelo algoritmo 3.7 para avaliar seu progresso. Comandos trocam de fase ao serem replicados para um número suficiente de seguidores conforme sua fase (equação 3.3).

```
procedure CommitPending(leader)
    for all cmd_i ∈ leader.log : leader.commitIdx < i ≤ |leader.log| do
        if cmd.phase ∈ {slow, fast} ∧ Replicated(cmd_i) then      ▷ Comandos replicados
            if cmd.phase = slow ∨ SkipSlow(cmd) then      ▷ Fase lenta ou podem pulá-la
                cmd.phase ← committed                      ▷ Promove para fase concluída
            else if cmd.phase = fast then                    ▷ Não pode pular fase lenta
                cmd.phase ← slow                             ▷ Promove para fase lenta
            end if
        end if
    end for
end procedure
```

Algoritmo 3.7: Algoritmo utilizado pelo líder avalia o progresso do consenso.

$$Replicated(c_i) = |Quorum(c_i)| \geq \begin{cases} \lfloor \frac{n}{2} \rfloor + f & \text{if } c.phase = fast \\ f + 1 & \text{if } c.phase = slow \end{cases} \tag{3.3}$$

$$Quorum(c_i) = \left\{ s \in Servers \mid i \leq \begin{cases} fastMatchIdx[s] & \text{if } c.phase = fast \\ slowMatchIdx[s] & \text{if } c.phase = slow \end{cases} \right\} \tag{3.4}$$

De acordo com o Atlas, comandos em fase rápida são replicados para todo o quórum rápido de processos, composto por $\lfloor \frac{n}{2} \rfloor + f$ pares, antes de trocar de fase. Estes comandos são concluídos se todas as suas dependências foram reportadas por ao menos f processos (equação 3.5). Caso contrário, é necessária mais uma rodada de consenso para informar aos seguidores sobre as dependências consolidadas pelor líder. Neste caso, estes comandos são promovidos para a fase lenta. Por outro lado, comandos na fase lenta podem ser considerados concluídos após replicados por pelo menos $f + 1$ processos.

$$SkipSlow(c) = \forall d \in c.deps : |\{d' \in c.deps : SameDep(d, d')\}| \geq f \quad (3.5)$$

$$SameDep(d, d') = d.leader = d'.leader \wedge d.pos = d'.pos \quad (3.6)$$

Note que, ao trocar da fase rápida para lenta ou concluída, um comando pode ter um conjunto de dependências no líder diferente do conjunto replicado para seus seguidores. Neste caso, após processar as mudanças de fase, o líder ajusta o índice do próximo comando a ser enviado para cada seguidor, repetindo comandos promovidos da fase rápida. Para mais detalhes, consulte o apêndice A, linhas 1066–1185.

3.3.3 Recuperação de Falhas

ERaft segue o mesmo principio de recuperação de falhas que Raft no qual um *RPC* é utilizado para eleger um novo líder do consenso. O mesmo mecanismo é utilizado tanto para eleger um processo como líder de seu próprio *log*, quanto para se eleger líder do log de um par do qual ele deseja recuperar. Ainda, eleições podem ocorrer quando um líder suspeita da falha de um seguidor que pertence ao seu quórum rápido. Neste caso, o líder escolhe um novo quórum no qual o processo suspeito não participa e inicia uma eleição para si mesmo, concluindo seus comandos pendentes antes de utilizar o quórum escolhido — veja apêndice A, linhas 1191–1231.

Toda eleição inicia com o incremento do termo para o processo suspeito de falha e a transição do estado do processo atual para candidato (apêndice A, linhas 651–697). ERaft aplica as mesmas restrições em seus *RPCs* que Raft para garantir que a propagação de novo termo entre líderes e seguidores: o *RPC* de replicação recusa mensagens de termos anteriores, enquanto o *RPC* de eleição se recusa a votar em processos de termos passados. Ainda, qualquer processo volta ao estado de seguidor se receber mensagens de um processo com termo maior que o dele.

Para ser capaz de concluir o consenso dos comandos de um processo suspeito, ERaft modifica o *RPC* de eleição de Raft de modo que além de pedir votos ele também coleta comandos pendentes (apêndice A, linhas 1233–1256 e 1319–1347). Diferente de Raft onde cada processo apenas responde a uma solicitação de voto por termo, em ERaft cada processo vota apenas uma vez por termo, mas toda solicitação é respondida. Esta mudança visa garantir que mesmo que um processo não vote em um candidato, caso ele vença a eleição por maioria simples de votos, o candidato ainda considere os comandos coletados dos pares que não votaram nele.

Um candidato é considerado líder se receber votos de ao menos $\lfloor \frac{n}{2} \rfloor + 1$ processos. No entanto, para garantir que ele observou um conjunto suficiente de comandos para

concluir o consenso, o Atlas exige que $n - f$ respostas sejam coletadas de seus pares. O algoritmo 3.8 descreve como o ERaft consolida os comandos coletados durante a eleição — especificado formalmente no apêndice A, linhas 1349–1513.

```
procedure MERGELOGS(Logs, i)
    if i = 0 then
        return ⟨⟩
    else
        C ← {c_i ∈ Log, Log ∈ Logs}                    ▷ Todos os comandos na posição i

        cmd ← c ∈ C : c.phase = committed    ▷ Escolhe um comando na fase concluída
        if cmd = ∅ then                                  ▷ Se nenhum em fase concluída
            cmd ← c ∈ C : c.phase = slow                 ▷ Escolhe um comando na fase lenta
        end if

        if cmd = ∅ then                                          ▷ Se nenhum em fase lenta
            if |C| ≥ ⌊n/2⌋ then                  ▷ Decide se pode escolher um na fase rápida
                cmd ← c ∈ C                              ▷ Escolhe um comando na fase rápida
                cmd.phase ← slow                         ▷ Promove comando para fase lenta
                cmd.deps ← ⋃{c'.deps | c' ∈ C}           ▷ Consolida dependências
            else
                cmd ← (cmd ↦ ∅, deps ↦ {}, phase ↦ slow)          ▷ Anula esta posição
            end if
        end if

        return Append(MergeLogs(logs, i − 1), cmd)          ▷ Consolida restante do log
    end if
end procedure
```

Algoritmo 3.8: Algoritmo utilizado para consolidar logs coletados durante uma eleição.

Caso o candidato observe um comando concluído ou na fase lenta para uma determinada posição do *log*, ele escolhe este comando para concluir o consenso dado que nestas fases seu conjunto de dependências já foi consolidado pelo líder e deve ser preservado. Por outro lado, se apenas comandos na fase rápida forem observados, não é possível determinar se o líder concluiu o consenso deste comando antes de falhar. Neste caso, Atlas prova que a união das dependências reportadas por $\lfloor \frac{n}{2} \rfloor$ processos recriam o mesmo conjunto que o líder observou, estratégia esta também adotada por ERaft. Em último caso, aquela posição do *log* recebe um comando nulo para desbloquear a execução de comandos que dependam dela.

Uma vez consolidados os comandos pendentes para o processo suspeito, o candidato atualiza o *log* do mesmo, passa para o estado de líder e utiliza o *RPC* de replicação discutido na sessão 3.3.2 para comunicar aos demais processos as decisões tomadas por ele. O processo que falhou, ao se recuperar, utiliza o mesmo mecanismo de eleição para tomar conhecimento das decisões feitas durante sua ausência. Para facilitar o retorno do

processo falho, o líder eleito para a recuperação do mesmo abdica da liderança após concluir o consenso do *log* consolidado (apêndice A, linhas 1515–1541).

3.4 Invariantes e Propriedades

De maneira informal é derivado que por utilizar o mesmo mecanismo e quórum de eleições que Raft, ambos provados corretos, as eleições de líderes em ERaft também são corretas. Durante a replicação de comandos e recuperação de falhas, o ERaft utiliza a mesma estratégia e quóruns que Atlas, o que o torna também correto nestes quesitos uma vez que Atlas é provado correto.

Para mais detalhes veja a especificação formal do algoritmo que conta com uma seção de invariantes e propriedades verificadas durante a exploração do espaço de estados do mesmo (apêndice A, linhas 1543–1781).

3.5 Considerações Práticas

Uma vez concluído o consenso para uma sequência de comandos, líderes comunicam seu progresso via *RPC* de replicação no qual seu índice de *commit* é propagado. Uma vez observados comandos em posição abaixo deste índice, seguidores podem executá-los em sua máquina de estados local desde que respeitando a restrição de ordem imposta pelo conjunto de dependências de cada comando. Na prática, isso gera um grafo dirigido com a possibilidade de ciclos que deve ser navegado de forma determinística por todos os processos. Métodos tradicionais de navegação de grafos, incluindo os métodos utilizados por EPaxos [12] e Atlas [3] podem ser aplicados a ERaft para implementar a execução.

Para garantir que processos não suspeitem da falha de líderes durante o comportamento normal do sistema, ERaft presume o mesmo comportamento que Raft onde líderes utilizam um intervalo de tempo menor que seguidores ao agendar seus *timeouts* e seguidores reiniciam o *timeout* referente ao líder para toda mensagem recebida dele. Também é esperado que *timeouts* sejam agendados com intervalos aleatórios de tempo para diminuir as chances de múltiplos processos suspeitarem da falha de um par simultaneamente, o que pode resultar em uma competição pela liderança do processo suspeito.

O custo da busca por dependências de um determinado comando cresce linearmente com o tamanho dos *logs* mantidos pelos processos. É possível limitar este custo desde que o limite inferior de busca seja adicionado ao conjunto de dependências do comando. Por exemplo, em um processo onde há um *log* com tamanho 100, se a busca por dependências é configurada para analisar apenas os 10 comandos mais recentes, o índice 89 do *log* deve ser considerado como dependência do comando avaliado. Este limite deve

ser configurado com base na velocidade média de execução de comandos durante o funcionamento normal do sistema para que, na maior parte dos casos, quando o comando concluir seu consenso o índice 89 do qual ele depende já foi executado pelos processos.

É documentada na literatura uma condição em que algoritmos de consenso igualitários podem sofrer de postergação indefinida [5]. Basta que um comando fique pendente e que o processo continue a aceitar novos comandos que dependem dele. Para mitigar este efeito, implementações destes algoritmos devem preferir concluir o consenso de comandos pendentes ao invés de aceitar novos comandos. Esta condição pode ser implementada em ERaft como uma restrição baseada na diferença entre o índice de *commit* e o tamanho do *log* onde o processo passa a rejeitar novos comandos se a diferença atingir um valor limite.

Dado que comandos são promovidos após serem replicados e existem índices de replicação para cada fase, é possível utilizar estes índices para determinar se um seguidor já tem conhecimento de um comando. Então, pode-se reduzir tráfego de rede reenviando apenas as dependências de um comando se promovido da fase rápida para lenta ou concluída.

4. CONCLUSÃO

Neste trabalho de conclusão de curso foi introduzido o algoritmo de consenso igualitário ERaft, inspirado pelos algoritmos Raft e Atlas. Assim como o Raft, ele possui apenas dois *RPCs* de comunicação entre processos visando diminuir a complexidade do algoritmo. Ainda em conformidade com Atlas, todo processo é líder de seus próprios comandos, o que tende a aumentar a vazão do sistema através do consenso e execução de comandos em paralelo, exceto na presença de conflito. A maior contribuição deste trabalho é sua especificação formal escrita em TLA$^+$, disponível no apêndice A, que contém a definição precisa do algoritmo proposto.

Trabalhos futuros relacionados a esta pesquisa incluem sua prova via um assistente de provas como Coq [1] ou Isabelle [2] de forma a aumentar a confiança em sua corretude. Além disso, a construção de um simulador visual aos moldes de RaftScope [3] pode servir como ferramenta didática e auxiliar na promoção deste algoritmo. Por fim, a construção de uma implementação concreta de ERaft e sua avaliação experimental de desempenho em comparação ao Raft clássico e similares pode evidenciar possíveis otimizações e pontos de melhoria.

[1] https://coq.inria.fr
[2] https://isabelle.in.tum.de
[3] https://raft.github.io

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Barcelona, C.-S. “Mencius: building efficient replicated state machines for wans”. In: 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08), 2008.

[2] Dwork, C.; Lynch, N.; Stockmeyer, L. “Consensus in the presence of partial synchrony”, *J. ACM*, vol. 35–2, apr 1988, pp. 288–323.

[3] Enes, V.; Baquero, C.; Rezende, T. F.; Gotsman, A.; Perrin, M.; Sutra, P. “State-machine replication for planet-scale systems”. In: Proceedings of the Fifteenth European Conference on Computer Systems, 2020, pp. 1–15.

[4] Fischer, M. J.; Lynch, N. A.; Paterson, M. S. “Impossibility of distributed consensus with one faulty processor”, *Journal of the ACM*, vol. 32–2, 1985, pp. 374–382.

[5] França Rezende, T.; Sutra, P. “Leaderless state-machine replication: Specification, properties, limits”. In: 34th International Symposium on Distributed Computing (DISC 2020), 2020.

[6] Howard, H. “Distributed consensus revised”, Tese de Doutorado, University of Cambridge, 2019.

[7] Lamport, L. “The part-time parliament”, *ACM Trans. Comput. Syst.*, vol. 16–2, may 1998, pp. 133–169.

[8] Lamport, L. “Paxos made simple”, *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, December 2001, pp. 51–58.

[9] Lamport, L. “Generalized consensus and paxos”, Relatório Técnico MSR-TR-2005-33, 2005.

[10] Lamport, L. “Fast paxos”, *Distributed Computing*, vol. 19, October 2006, pp. 79–103.

[11] Lamport, L.; Massa, M. “Cheap paxos”. In: International Conference on Dependable Systems and Networks, 2004, 2004, pp. 307–314.

[12] Moraru, I.; Andersen, D. G.; Kaminsky, M. “There is more consensus in egalitarian parliaments”. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013, pp. 358–372.

[13] Ongaro, D.; Ousterhout, J. “In search of an understandable consensus algorithm”. In: 2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14), 2014, pp. 305–319.

[14] Pedone, F.; Schiper, A. “Handling Message Semantics with Generic Broadcast Protocols”, *Distrib. Comput.*, vol. 15–2, 2002, pp. 97–107, generic broadcast.

[15] Rezende, T. F.; Sutra, P.; Saramago, R. Q.; Camargos, L. “On making generalized paxos practical”. In: 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA), 2017, pp. 347–354.

[16] Van Renesse, R.; Altinbuken, D. “Paxos made moderately complex”, *ACM Comput. Surv.*, vol. 47–3, feb 2015.

APÊNDICE A – ESPECIFICAÇÃO EM TLA+

1 MODULE *ERaft*

Egalitarian Raft (*ERaft*) is an egalitarian – also called leaderless – version of the well known Raft consensus algorithm. *ERaft* heavily draws inspiration from the Atlas consensus protocol to achieve optimal commit latency in the presence of conflicting commands.

The following section offers a high level overview of the *ERaft* algorithm, whereas the specification itself provides a precise definition of each component mentioned.

—— HIGH LEVEL OVERVIEW ——

ERaft works similarly to Raft: every server process starts at the "follower" state; followers can become a "candidate" due to inactivity (timeout); a round of election happens to which only one candidate wins by a majority vote – split votes are disregarded and a new election is called. The winning candidate transitions to the "leader" state and begins managing the *log* of commands submitted by clients. *ERaft* slightly modifies this model by introducing another dimension: each server process has its own *log*.

A server can be simultaneously at the "leader" state for its own *log* while being at the "follower" state for every other server process in the system during normal operation. For example:

Server 1:
(leader) Server 1 → [a, *b*, c]
(follower) Server 2 → [*d*]
(follower) Server 3 → [*e*, f]

Server 2:
(follower) Server 1 → [a, *b*, c]
(leader) Server 2 → [*d*]
(follower) Server 3 → [*e*, f]

Server 3:
(follower) Server 1 → [a, *b*, c]
(follower) Server 2 → [*d*]
(leader) Server 3 → [*e*, f]

Commands are submitted by clients to one of the server processes available. Upon receiving a command, the server will compute the set of known dependencies for the command before appending it to its own *log*. The commands' set of dependencies are used to determine the *log*'s execution order in which commands can only be executed after their dependencies so that logical data consistency is preserved. *ERaft* detect dependencies via an abstract conflict function provided by the application.

As per Raft, a single *RPC* – called "append" – is used to replicate commands between leaders and followers. *ERaft* extends this *RPC* to exchange dependencies discovered during command replication. As per Atlas, commands can be in different phases. *ERaft* establishes that every new command appended to the leader's *log* starts at the "fast" phase and is sent via broadcast to a quorum of server processes which reply with their own set of computed dependencies for it. If the leader determines that the command can be safely committed based on the reported dependencies, the command is moved to the "committed" phase. Otherwise, commands are promoted to the "slow" phase and another call to the append *RPC* is required before committing it.

Committed commands are immutable and can be executed by the application which must respect the ordering constraints established by their dependency set. In practice, this results in a cyclic graph that servers have to traverse and solve cycles deterministically.

Upon suspecting of failures, servers can become candidates for some other server's *log*. The election process derived from Raft – here called "recovery" *RPC* – combines election with command discovery giving the elected recovery leader the necessary view of the failing server's *log* so that it can complete the pending commands or replace them with *NULL* if necessary. A recovery is also triggered when a leader needs to reconfigure its quorums when detecting failed member.

Recovery leaders, once elected, proceed with standard append *RPC* calls in order to replicate and commit commands discovered during its election. Recovery leaders never append new commands to the failed
leader's *log*. Once all known commands are committed by the recovery
leader, it abdicates its leadership by going back to the "follower" state.

This specification is meant as a comprehensive model of both command replication and failure recovery portion of the algorithm while committed commands execution and *log* truncation are left to a separate specification.

—— REFERENCES ——

– *Ongaro*, *D*. and *Ousterhout*, *J*. (2014). In search of an understandable
consensus algorithm. In 2014 {*USENIX*} Annual Technical Conference ({*USENIX*}{*ATC*} 14), pages 305–319.

– *Enes*, *V*., *Baquero*, *C*., *Rezende*, *T*. *F*., *Gotsman*, A., *Perrin*, *M*., and Sutra, *P*. (2020). State-machine replication for planet-scale systems. In Proceedings of the Fifteenth *European* Conference on Computer Systems, pages 1–15.

101 EXTENDS *FiniteSets*, *Naturals*, *Sequences*, *TLC*, *Utils*

103 ├──┤

A finite set of interconnected server processes.

E.g.: {*s*1, *s*2, *s*3}

109 CONSTANT *Servers*

A finite set of commands to be proposed by servers.

E.g.: {*a*, *b*, *c*}

116 CONSTANT *Commands*

A commutative function that takes 2 commands and return TRUE iff they conflict with each other. Returns FALSE otherwise.

E.g.: LAMBDA $x, y : x = y$

124 CONSTANT $Conflict(\_, \_)$

A constant representing the absence of a value.

129 CONSTANT *NULL*

A natural number indicating the amount of tolerated failed processes (as per Atlas).

135 CONSTANT *F*

The number of server processes in the system.

140 $N \triangleq Cardinality(Servers)$

The size of a process fast quorum (as per Atlas).

$FQ \triangleq (N \div 2) + F$

The size of a process slow quorum (as per Atlas).

$SQ \triangleq F + 1$

The size of a recovery quorum (as per Atlas).

$RQ \triangleq N - F$

The size of an election quorum (as per Raft).

$EQ \triangleq (N \div 2) + 1$

ASSUME

Assume *Servers* and *Commands* to be non-empty finite sets, *NULL* values are non-conflicting, the conflict function to be commutative, and F to be properly configured (as per Atlas).

$$
\begin{aligned}
&\land\ IsFiniteSet(Servers) \\
&\land\ Cardinality(Servers) > 0 \\
&\land\ IsFiniteSet(Commands) \\
&\land\ Cardinality(Commands) > 0 \\
&\land\ \forall\, cmd \in Commands : \\
&\qquad Conflict(cmd,\ NULL) = \text{FALSE} \\
&\land\ \forall\, a,\ b \in Commands \cup \{NULL\} : \\
&\qquad Conflict(a,\ b) = Conflict(b,\ a) \\
&\land\ 1 \leq F \land F \leq (N - 1) \div 2
\end{aligned}
$$

STATE VARIABLES

This section describes the state variables used throughout the spec. Notably, the spec state is multi-dimensional. Server variables are defined as a function from "$s1 \rightarrow s2 \rightarrow$ value" where "$s1$" defines the server process in which the state variable resides, and "$s2$" represents a server from which "$s1$" has knowledge about (possibly the same as "$s1$"). For example, a possible slice of the *log* variable is:

$$
\begin{aligned}
&(s1 :> \ (s1 :> \langle [cmd : a,\ \ldots],\ \ldots \rangle) \\
&\quad @@\ (s2 :> \langle [cmd : b,\ \ldots],\ \ldots \rangle))
\end{aligned}
$$

In the example above, both sequences of commands resides in the server $s1$. The server $s1$ has command "a" appended to its own *log*. The server $s2$ has replicated command "b" to server "$s1$".

Variables used during recovery contain a third dimension as they are used to track both the state of the leader processes as well as the state of the recovering process. For example, take a possible slice of the *nextIdx* variable:

$$
\begin{aligned}
&(s1 :> \ (s1 :> \ (s1 :> 2) \\
&\qquad @@\ (s2 :> 1) \\
&\qquad @@\ (s3 :> 1)) \\
&\quad @@\ (s2 :> \ (s1 :> 3) \\
&\qquad @@\ (s2 :> 3) \\
&\qquad @@\ (s3 :> 1)) \\
&\quad @@\ \ldots)
\end{aligned}
$$

Again, the first dimension defines the server process where the state resides. In the example above, $s1$ has its own *nextIdx* set to 1 for both servers $s2$ and $s3$. Moreover, while coordinating the recovery the process $s2$, it has $s2$'s *nextIdx* set to 1 for server $s3$.

A function from a server process to a function from leader to its current term – defined as a natural number. Similarly to Raft, term act as a synchronization mechanism when a leader is suspected to have failed and another server process attempts to recover its *log*.

```
221 VARIABLE currTerm

223 CurrTermOK ≜
      For every known server process, its term is a natural number.
227   ∀ server, leader ∈ Servers : currTerm[server][leader] ∈ Nat

229 InitCurrTerm ≜
      All servers start at the same term 0.
233   currTerm = [server ∈ Servers ↦ [leader ∈ Servers ↦ 0]]
```

A function from server to its current state.

```
238 VARIABLE currState

240 CurrStateOK ≜
      Current state transitions are similar to Raft's:
      − Servers start as followers;
      - Followers transition to candidates;
      - Candidates transition to follower or leader;
      - Leaders transition to follower or candidate.
252   ∀ server, leader ∈ Servers :
253     currState[server][leader] ∈ {"follower", "candidate", "leader"}

255 InitCurrState ≜
      Every leader starts as a follower. It must call for a recovery – analogous to a Raft election – to
      claim ownership of its log.
260   currState = [server ∈ Servers ↦ [leader ∈ Servers ↦ "follower"]]
```

A function from a server to the leader's ordered sequence of commands agreed upon consensus.

```
266 VARIABLE log

268 Dependencies ≜
      Dependencies is a set of records containing the reporting peer, the command's leader, and its
      position in the leader's log.
273   [reporter : Servers, leader : Servers, pos : Nat]

275 LogEntries ≜
      A log entry is composed of its current phase, a single command, and its set of dependencies.
      Although not necessary for correctness, the log entry also carries its position in the leader's log
      to simplify specification.
```

```
282   [phase : {"fast", "slow", "committed"},
283    cmd   : Commands ∪ {NULL},
284    deps  : SUBSET Dependencies,
285    pos   : Nat]

287 LogOK ≜
```

For every server and leader's *log*, each *log* entry is correctly indexed, and contained within the set of possible *log* entries.

Moreover *NULL* commands have no dependencies. During failure recovery, *NULL* values can replace commands from which peers depend on but wasn't yet replicated by the leader before failing.

```
296   ∀ server, leader ∈ Servers :
297     ∀ i ∈ DOMAIN log[server][leader] :
298       LET
299         entry ≜ log[server][leader][i]
300       IN
301         ∧ entry.pos = i
302         ∧ entry ∈ LogEntries
303         ∧ entry.cmd = NULL ⇒ entry.deps = {}

305 InitLog ≜
```

Servers start with an empty sequence of commands for every known leader.

```
310   log = [server ∈ Servers ↦ [leader ∈ Servers ↦ ⟨⟩]]
```

A function from server process to a function from leaders to the followers' next index: a natural number indicating the position of the next command in the leader's *log* to be sent to the associated follower (as per Raft).

Note that the *nextIdx* can move ahead of the *log* length. If $nextIdx \leq Len(log)$, there are commands to be sent to the follower. Otherwise, leader has already sent all its *log* entries to the follower, thus resulting in *nextIdx* being set to $Len(log) + 1$.

```
323 VARIABLE nextIdx

325 NextIdxOK ≜
```

For every server, every leader's next index with regards to its follower is a natural number greater than zero.

```
330   ∀ server, leader, follower ∈ Servers :
331     nextIdx[server][leader][follower] ∈ Nat \ {0}

333 InitNextIdx ≜
```

Since logs start empty, *nextIdx* starts as 1 for every known leader which indicates that at start there are no commands to be replicated.

```
338   nextIdx = [
339     server ∈ Servers ↦ [
340       leader ∈ Servers ↦ [
341         follower ∈ Servers ↦ 1
342     ]]]
```

Functions from server processes to a function from leaders to follower's match index: a natural number indicating the position where the follower matches the *log* from the leader with regards to fast and slow phases, respectively.

This is an adaptation of Raft's match index concept to allow for Atlas fast and slow phase commits.

```
353 VARIABLES fastMatchIdx, slowMatchIdx

355 MatchIndexesOK ≜
```

For every known server, the leader's match indexes with regards to each follower is be a natural number.

```
360   ∀ server, leader, follower ∈ Servers :
361     ∧ fastMatchIdx[server][leader][follower] ∈ Nat
362     ∧ slowMatchIdx[server][leader][follower] ∈ Nat

364 InitMatchIndexes ≜
```

Since logs start empty, match indexes start at 0 for every known follower indicating no commands were replicated yet.

```
369   ∧ fastMatchIdx = [
370       server ∈ Servers ↦ [
371         leader ∈ Servers ↦ [
372           follower ∈ Servers ↦ 0
373     ]]]
374   ∧ slowMatchIdx = [
375       server ∈ Servers ↦ [
376         leader ∈ Servers ↦ [
377           follower ∈ Servers ↦ 0
378     ]]]
```

A function from servers to leader's commit index: a natural number indicating the highest command position committed for a given leader, or zero if unknown (as per Raft).

```
385 VARIABLE commitIdx

387 CommitIdxOK ≜
```

For every known leader, its commit index is a natural number.

```
391   ∀ server, leader ∈ Servers : commitIdx[server][leader] ∈ Nat

393 InitCommitIdx ≜
```

Start with zero as the commit index for every known server process.

```
397   commitIdx = [server ∈ Servers ↦ [leader ∈ Servers ↦ 0]]
```

A function from leaders to their fast quorum of server processes, including itself, from which non-diverging responses can be safely committed without a second round of consensus (as per Atlas).

```
404 VARIABLES fastQuorum

406 FastQuorumOK ≜
```

A leader's fast quorum is correct if it's a subset of server processes, is properly sized, and contains the leader itself.

```
411   ∀ server ∈ Servers :
412     ∧ fastQuorum[server] ⊆ Servers
413     ∧ Cardinality(fastQuorum[server]) = FQ
414     ∧ server ∈ fastQuorum[server]

416 ChooseFastQuorum(leader, peers) ≜
      Arbitrarily chooses a fast quorum for the given leader among its available peers.
421   CHOOSE quorum ∈ SUBSET (peers ∪ {leader}) :
422     ∧ leader ∈ quorum
423     ∧ Cardinality(quorum) = FQ

425 InitFastQuorum ≜
      Arbitrarily chooses a fast quorum per leader among all server processes at start.
430   fastQuorum = [leader ∈ Servers ↦ ChooseFastQuorum(leader, Servers)]

      A function from server process to a function from term to server tracking for which server the
      current process voted in the associated term.

      As per Raft, a server only votes for a single peer for a given election.
439 VARIABLE votedFor

441 VotedForOK ≜
      For every leader known by server, known terms are natural numbers, and the associated cast
      vote is a server process.
446   ∀ server, leader ∈ Servers :
447     ∀ term ∈ DOMAIN votedFor[server][leader] :
448       ∧ term ∈ Nat
449       ∧ votedFor[server][leader][term] ∈ Servers

451 InitVotedFor ≜
      Start with an empty mapping function as no votes have been cast.
455   votedFor = [
456     server  ∈ Servers ↦ [
457       leader  ∈ Servers ↦ [
458         term ∈ {} ↦ {}
459   ]]]

      A function from a server process to a leader from which the server is attempting to recover.
465 VARIABLE recovery

467 Recovery ≜
      Recovery is a temporary state maintained by candidates during the recovery process. It tracks
      the highest known command index for the failing leader.

      Also, it tracks replies collected during election, which are composed of a reporting peer, the
      commit index for the failed leader known by the peer, a flag indicating if the vote has been
      granted, and a slice of the log from the failing leader according to the reporting peer.
```

```
478   [replies : SUBSET [
479      reporter     : Servers,
480      commitIdx    : Nat,
481      voteGranted  : BOOLEAN ,
482      slice        : Seq(LogEntries)
483    ],
484    maxIdx  : Nat]

486 RecoveryOK ≜
```

For every server, the state of each leader where server attempts recovery belongs to the set of possible recovery states.

```
491   ∀ server ∈ Servers :
492     ∀ leader ∈ DOMAIN recovery[server] :
493       ∧ leader ∈ Servers
494       ∧ recovery[server][leader] ∈ Recovery

496 InitRecovery ≜
```

Start with no recovery in progress.

```
500   recovery = [server ∈ Servers ↦ [leader ∈ {} ↦ {}]]
```

A set of messages in the system's network. Note that the spec makes no assumption about message ordering. The usage of a set implies message uniqueness but, the spec makes no attempt to prevent duplicated message handling, thus modeling message loss and repeated message delivery from which the algorithm is safe against.

Note that three fields are present in every message: type, indicating the message type; *src* representing the message's source server; and *dest*, modeling the message's destination server. These fields are meant to model the network stack. Concrete implementations of the algorithm can omit them.

```
515 VARIABLE messages

517 AppendRPC ≜
```

As per Raft, the *AppendRPC* is used to replicate commands from leaders to followers. Since a recovery leader can temporarily assume the leadership of a peer's *log*, the leader field indicates which leader's *log* is being replicated at the time. A message with *src* ≠ *leader* can be seen as the recovery leader *src* replicating *log* on behaf of the failed leader.

The append *RPC* has been modified to include in its response a sequence of dependencies set for each replicated command, thus allowing for dependency collection like in the Atlas protocol.

```
530   LET
531     Request ≜
532       [type           : {"append-request"},
533        src            : Servers,
534        dest           : Servers,
535        leader         : Servers,
536        term           : Nat,
537        leaderCommit   : Nat,
538        prevLogIndex   : Nat,
539        entries        : Seq(LogEntries)]
```

```
    Reply ≜
      [type          : {"append-reply"},
       src           : Servers,
       dest          : Servers,
       leader        : Servers,
       success       : BOOLEAN ,
       term          : Nat,
       prevLogIndex  : Nat,
       dependencies  : Seq(SUBSET Dependencies)]
  IN
    Request ∪ Reply

RecoveryRPC ≜
```

The *RecoveryRPC* coordinates an election analogous to Raft. Moreover, it allows for participating servers to report back on the known leader's *log* so that the elected leader can consolidate and make progress on pending commands on behalf of the failing leader.

```
  LET
    Request ≜
      [type          : {"recovery-request"},
       src           : Servers,
       dest          : Servers,
       leader        : Servers,
       term          : Nat,
       commitIdx     : Nat]

    Reply ≜
      [type          : {"recovery-reply"},
       src           : Servers,
       dest          : Servers,
       leader        : Servers,
       term          : Nat,
       maxIdx        : Nat,
       commitIdx     : Nat,
       voteGranted   : BOOLEAN ,
       slice         : Seq(LogEntries)]
  IN
    Request ∪ Reply

MessagesOK ≜
```

A network message is valid if it's part of the either append or recovery *RPCs*.

```
  messages ⊆ AppendRPC ∪ RecoveryRPC

InitMessages ≜
```

The network starts with an empty set of messages.

```
  messages = {}
```

```
    The sequence of all variables defined in the spec.
598 vars ≜ ⟨
599   currTerm, currState, log, nextIdx, fastMatchIdx, slowMatchIdx,
600   commitIdx, fastQuorum, votedFor, recovery, messages
601 ⟩
602 ├──────────────────────────────────────────────────────────────────────┤
    ── Helper Functions ──

607 ComputeDeps(peer, command) ≜
    Compute dependencies for the given command according to the peer's view of all leaders' logs.
612   LET
613     allLogs ≜ {
614       ⟨leader, SetOf(log[peer][leader])⟩ : leader ∈ Servers
615     }
616     allEntries ≜ {
617       {⟨leader, entry⟩ : entry ∈ llog} : ⟨leader, llog⟩ ∈ allLogs
618     }
619     conflicts ≜ {
620       ⟨leader, entry⟩ ∈ UNION allEntries : Conflict(command, entry.cmd)
621     }
622   IN  {
623     [reporter ↦ peer,
624      leader   ↦ leader,
625      pos      ↦ entry.pos
626     ] : ⟨leader, entry⟩ ∈ conflicts
627   }

629 MaxDepPos(server, leader) ≜
    Returns the highest uncommitted index from which any command known by server depend on
    the given leader.
634   LET
635     allLogs ≜ UNION {
636       SetOf(log[server][peer]) : peer ∈ Servers
637     }
638     uncommittedDeps ≜ {
639       entry ∈ allLogs :
640         ∃ dep ∈ entry.deps :
641           ∧ dep.leader = leader
642           ∧ dep.pos > commitIdx[server][leader]
643     }
644     uncommittedDepsPos ≜ {
645       entry.pos : entry ∈ uncommittedDeps
646     }
647   IN
648     CHOOSE i ∈ uncommittedDepsPos ∪ {0} :
```

```
649         ∀ j ∈ uncommittedDepsPos : i ≥ j

651 StartRecovery(server, leader) ≜
```

Set all variables needed to start a recovery for the given leader with server as a candidate.

Similarly to Raft, elections start by incrementting the leader's current term, changing the server's state to candidate with regards to the leader, and voting for itself. Additionally, the temporary state of recovery is reset to account for the candidate server vote.

```
661     ∧ currTerm' = [
662         currTerm EXCEPT ![server][leader] = @ + 1
663       ]
664     ∧ currState' = [
665         currState EXCEPT ![server][leader] = "candidate"
666       ]
667     ∧ votedFor' = [
668         votedFor EXCEPT ![server][leader] =
669           (currTerm'[server][leader] :> server) @@ @
670       ]
```

The maximum index for the leader's *log* is the highest command position known by the candidate, rather the *log* length or a dependency reported by the leader but not replicated to the candidate. Moreover, considering that a candidate always vote for itself, it fills the replies field with an equivalent reply record accounting for its own.

```
679     ∧ recovery' = [
680         recovery EXCEPT ![server] =
681           (leader :> [
682              maxIdx  ↦ Max(
683                MaxDepPos(server, leader),
684                Len(log[server][leader])
685              ),
686              replies ↦ {[
687                voteGranted ↦ TRUE,
688                reporter    ↦ server,
689                commitIdx   ↦ commitIdx[server][leader],
690                slice       ↦ SubSeq(
691                  log[server][leader],
692                  commitIdx[server][leader] + 1,
693                  Len(log[server][leader])
694                )
695              ]}
696            ]) @@ @
697       ]

699 ├──────────────────────────────────────────────────────────────────────┤
```

── Actions ──

Like Raft, *ERaft* is design to work on a timeout loop. Since *ERaft* maintains multiple logs, Raft's original timeout design must also be adapted to work as one timeout per *log*. Although time isn't modeled by the spec, the following actions describe operations ran in response to timeouts.

Note that, although any interleaving among defined actions is possible, each action is itself an atomic operation.

```
713 AppendCommandToLog(leader, command) ≜
```

This action models a client submitting a command to given leader.

Given a leader and a command, compute the command's set of dependencies and append it to the leader's *log* in the fast phase.

```
720  ∧ currState[leader][leader] = "leader"
721  ∧ log' = [
722      log EXCEPT ![leader][leader] =
723        Append(@, [
724          phase ↦ "fast",
725          cmd   ↦ command,
726          deps  ↦ ComputeDeps(leader, command),
727          pos   ↦ Len(log[leader][leader]) + 1
728        ])
729    ]
```

The leader won't attempt to replicate its own *log* to itself, thus, the following indexes won't be updated in response to the append *RPC* call. Therefore, adjusting their values here will simplify decisions made later in the algorithm as the leader won't have to be treated as a special server elsewhere.

```
737  ∧ nextIdx' = [nextIdx EXCEPT ![leader][leader][leader] = @ + 1]
738  ∧ fastMatchIdx' = [fastMatchIdx EXCEPT ![leader][leader][leader] = @ + 1]
739  ∧ slowMatchIdx' = [slowMatchIdx EXCEPT ![leader][leader][leader] = @ + 1]
740  ∧ UNCHANGED ⟨
741      currTerm, currState, commitIdx, fastQuorum,
742      votedFor, recovery, messages
743    ⟩
```

—— REPLICATION ACTIONS ——

```
749 SendAppendRequest(server, leader) ≜
```

This action models a leader timeout, either original (*server* = *leader*) or recovery leader (*server* ≠ *leader*).

Upon a timeout, initiates the append *RPC* from server on behalf of the leader by sending an "append-request" message in order to replicate pending commands and/or its latest commit index.

As per Raft, *log* entries sent are a slice of the leader's *log* from *nextIdx* to *Len*(*log*). Note that, as per Atlas, commands in the "fast" phase are only sent to the leader's fast quorum. Otherwise, commands are sent via broadcast to all servers.

```
763  ∧ currState[server][leader] = "leader"
764  ∧ messages' = messages ∪ {
765      [type          ↦ "append-request",
766       src           ↦ server,
767       dest          ↦ follower,
```

```
768         leader          ↦ leader,
769         term            ↦ currTerm[server][leader],
770         leaderCommit ↦ commitIdx[server][leader],
771         prevLogIndex  ↦ nextIdx[server][leader][follower] − 1,
772         entries         ↦ TakeWhile(
773           SubSeq(
774             log[server][leader],
775             nextIdx[server][leader][follower],
776             Len(log[server][leader])
777           ),
778           LAMBDA entry :
779             entry.phase = "fast" ⇒
780               follower ∈ fastQuorum[leader]
781         )
782       ] : follower ∈ Servers \ {server}
783     }
784   ∧ UNCHANGED ⟨
785       currTerm, currState, log, nextIdx, fastMatchIdx, slowMatchIdx,
786       commitIdx, fastQuorum, votedFor, recovery
787     ⟩

789 HandleAppendRequest(msg) ≜
      This action models a server receiving an "append-request" message from an alleged leader.
794   ∧ msg.type = "append-request"
        Discard old/repeated messages with outdated commit indexes.
798   ∧ msg.leaderCommit ≥ commitIdx[msg.dest][msg.leader]
        As per Raft, leaders may attempt to proceed with log replication after failing. If noticing an
        outdated term, reject the append and reply with the most recent term for the leader.

        Moreover, if the follower is behind the leader's log, which can occur when the elected recovery
        leader has more commands in its log than the follower, reply with the Len(log) so that the
        leader can adjust its nextIdx for the next append request.
809   ∧ IF  ∨ msg.term < currTerm[msg.dest][msg.leader]
810         ∨ msg.prevLogIndex > Len(log[msg.dest][msg.leader])
811       THEN
812         ∧ messages′ = messages ∪ {
813             [type          ↦ "append-reply",
814              success       ↦ FALSE,
815              src           ↦ msg.dest,
816              dest          ↦ msg.src,
817              leader        ↦ msg.leader,
818              term          ↦ currTerm[msg.dest][msg.leader],
819              prevLogIndex ↦ Len(log[msg.dest][msg.leader]),
820              dependencies ↦ ⟨⟩
821             ]
```

```
822             }
823           ∧ UNCHANGED ⟨
824               currTerm, currState, log, nextIdx, fastMatchIdx, slowMatchIdx,
825               commitIdx, fastQuorum, votedFor, recovery
826           ⟩
827         ELSE
828           ∧ LET
                Append behaves differently depending on the command's current phase. Commands in the slow or committed phase are simply accepted as long as they don't override an existing committed command.

                New commands in the fast phase are accepted with an updated set of dependencies that takes the follower's view of the commands; dependencies set into account.

                On the other hand, existing commands in the fast phase are preserved since repeated calls to the append RPC requires the set of dependencies to remain the same until the leader makes a decision to either commit or promote the command to the slow phase (as per Atlas).
845             entries ≜ Map(msg.entries, LAMBDA entry :
846               LET
847                 llog ≜ log[msg.dest][msg.leader]
848                 curr ≜ llog[entry.pos]
849               IN
850                 IF entry.phase ≠ "fast"
851                    THEN
852                     IF ∧ entry.pos ≤ Len(llog)
853                        ∧ curr.phase = "committed"
854                        THEN curr
855                        ELSE entry
856                    ELSE
857                     IF entry.pos ≤ Len(llog)
858                        THEN curr
859                        ELSE [
860                          entry EXCEPT !.deps =
861                            @ ∪ ComputeDeps(msg.dest, entry.cmd)
862                        ]
863             )
864           IN
                If receiving an append from a leader with higher term, abdicate your current state (candidate or leader) and start following the new leader (as per Raft).
870           ∧ currTerm′ = [
871               currTerm EXCEPT ![msg.dest][msg.leader] =
872                 msg.term
873             ]
874           ∧ currState′ = [
875               currState EXCEPT ![msg.dest][msg.leader] =
876                 "follower"
```

```
877                 ]
                    Update the follower's log according to the updated log entries and reply back to
                    leader if there are any dependencies to report. Note that the leader might send
                    empty append messages to restate its leadership and prevent recoveries from
                    happening, as well as communicate progress on its commit index (as per Raft).
886             ∧ log' = [
887                 log EXCEPT ![msg.dest][msg.leader] =
888                   ReplaceSlice(@, msg.prevLogIndex, entries)
889               ]
890             ∧ commitIdx' = [
891                 commitIdx EXCEPT ![msg.dest][msg.leader] =
892                   msg.leaderCommit
893               ]
894             ∧ IF Len(entries) > 0
895                THEN
896                 messages' = messages ∪ {
897                   [type          ↦ "append-reply",
898                    success       ↦ TRUE,
899                    src           ↦ msg.dest,
900                    dest          ↦ msg.src,
901                    leader        ↦ msg.leader,
902                    term          ↦ msg.term,
903                    prevLogIndex  ↦ msg.prevLogIndex,
904                    dependencies  ↦ Map(entries, LAMBDA e : e.deps)
905                   ]
906                 }
907                ELSE
908                 UNCHANGED messages
909           ∧ UNCHANGED ⟨
910               nextIdx, fastMatchIdx, slowMatchIdx,
911               fastQuorum, votedFor, recovery
912             ⟩

914 HandleAppendReply(msg) ≜
      This action models a server receiving an "append-reply" message form a follower.
919   ∧ msg.type = "append-reply"
        Reject replies when the destination is not the log's leader or from outdated terms (as per
        Raft).
924   ∧ currState[msg.dest][msg.leader] = "leader"
925   ∧ msg.term ≥ currTerm[msg.dest][msg.leader]
        Reject old/repeated messages with log slices starting at an outdated replication position.

        If the log slice starts ahead the fast match index, this is a slice of new commands being
        replicated to the follower. As a result of the append RPC, the fast match index gets updated
        to the maximum log position known to be replicated to the follower.
```

If the *log* slice starts behind of the fast match index, this is either: a slice containing commands promoted to the slow phase and are, therefore, being replicated again to the follower, to which case the *log* slice starts ahead the slow match index; or else it's an old message and must be discarded.

```
941  ∧ msg.prevLogIndex < fastMatchIdx[msg.dest][msg.leader][msg.src] ⇒
942      msg.prevLogIndex ≥ slowMatchIdx[msg.dest][msg.leader][msg.src]
```

On a failed append request, consider outdated follower or leadershop loss.

```
947  ∧ IF ¬msg.success
948      THEN
949       IF msg.term = currTerm[msg.dest][msg.leader]
950          THEN
```

If still the leader, the follower must be behind in *log* replication and has set the *prevLogIndex* in the reply to its known *log* length. Reset the next index at the leader so that next append *RPC* can succeed. This is slight different than traditional Raft design where it simply decrements the next index by one.

```
959             ∧ nextIdx′ = [
960                 nextIdx EXCEPT ![msg.dest][msg.leader][msg.src] =
961                   msg.prevLogIndex + 1
962               ]
963             ∧ UNCHANGED ⟨
964                 currTerm, currState, log, fastMatchIdx, slowMatchIdx,
965                 commitIdx, fastQuorum, votedFor, recovery, messages
966               ⟩
967          ELSE
```

If receiving a reply from a peer with a higher term, fallback to being a follower (as per Raft).

```
972             ∧ currState′ = [
973                 currState EXCEPT ![msg.dest][msg.leader] =
974                   "follower"
975               ]
976             ∧ currTerm′ = [
977                 currTerm EXCEPT ![msg.dest][msg.leader] =
978                   msg.term
979               ]
980             ∧ UNCHANGED ⟨
981                 log, nextIdx, fastMatchIdx, slowMatchIdx, commitIdx,
982                 fastQuorum, votedFor, recovery, messages
983               ⟩
984      ELSE
985       ∧ LET
```

When handling a reply, map over the commands in the same *log* slice and update their dependencies set depending on their current phase.

Commands in the slow or committed phase are kept as they are, while commands in the fast phase accumulate dependencies reported by their followers.

```
995             entries ≜ Map(
```

```
996             SubSeq(
997               log[msg.dest][msg.leader],
998               msg.prevLogIndex + 1,
999               msg.prevLogIndex + Len(msg.dependencies)
1000            ),
1001            LAMBDA entry :
1002              IF entry.phase ≠ "fast"
1003                THEN entry
1004                ELSE [
1005                  entry EXCEPT !.deps =
1006                    @ ∪ msg.dependencies[entry.pos − msg.prevLogIndex]
1007                ]
1008          )
              For commands in the slow phase, identify the prefix of commands with matching
              dependencies with the replying follower.
1014          slow ≜ SelectSeq(entries, LAMBDA entry :
1015            entry.phase = "slow"
1016          )
1017          matching ≜ TakeWhile(slow, LAMBDA entry :
1018            entry.deps = msg.dependencies[entry.pos − msg.prevLogIndex]
1019          )
1020      IN
              Update the log to account for new dependencies learned.
1025        ∧ log′ = [
1026            log EXCEPT ![msg.leader][msg.leader] =
1027              ReplaceSlice(@, msg.prevLogIndex, entries)
1028          ]
              Fast match index is updated to match the highest command position known to
              be replicated at the replying follower.
1034        ∧ fastMatchIdx′ = [
1035            fastMatchIdx EXCEPT ![msg.dest][msg.leader][msg.src] =
1036              Max(@, Last(entries).pos)
1037          ]
              Slow match index is updated to the highest contiguous command known to be
              have matching dependencies with the replying follower.
1043        ∧ slowMatchIdx′ = [
1044            slowMatchIdx EXCEPT ![msg.dest][msg.leader][msg.src] =
1045              IF Len(matching) > 0 THEN Last(matching).pos ELSE @
1046          ]
              Next index is set to fast match + 1 if no slow commands are being handled in
              this reply, slow match + 1 if there are matching commands in the slow phase,
              or else remains unchanged.
1053        ∧ nextIdx′ = [
1054            nextIdx EXCEPT ![msg.dest][msg.leader][msg.src] =
1055              CASE Len(slow) = 0 →
```

```
                                fastMatchIdx'[msg.dest][msg.leader][msg.src] + 1
                        □   Len(matching) > 0 →
                                slowMatchIdx'[msg.dest][msg.leader][msg.src] + 1
                        □   OTHER → @
                    ]
        ∧ UNCHANGED ⟨
            currTerm, currState, commitIdx, fastQuorum,
            votedFor, recovery, messages
          ⟩

CommitReplicatedCommands(server, leader) ≜
```

This action models the part of the leader's timeout loop where it decides which commands to commit (as per Raft).

```
    ∧ currState[server][leader] = "leader"
    ∧ LET
```

Select commands above the leader's commit index which are known to be replicated to a sufficient number of followers according to their phase.

```
        replicated ≜ TakeWhile(
          SubSeq(
            log[server][leader],
            commitIdx[server][leader] + 1,
            Len(log[server][leader])
          ),
          LAMBDA entry :
            CASE  entry.phase = "committed" → TRUE
              □   entry.phase = "slow" →
                    LET matching ≜ {
                      follower ∈ Servers :
                        entry.pos ≤ slowMatchIdx[server][leader][follower]
                      }
                    IN
                      Cardinality(matching) ≥ SQ
              □   entry.phase = "fast" →
                    LET matching ≜ {
                     follower ∈ Servers :
                       entry.pos ≤ fastMatchIdx[server][leader][follower]
                    }
                    IN
                      Cardinality(matching) ≥ FQ
        )
```

Update replicated commands: commands committed are preserved; commands in the slow phase are promoted to the committed phase; commands in the fast phase are promoted to either the committed or slow phases.

As per Atlas, commands in the fast phase can skip the slow phase if at least F nodes, excluding itself, have reported the same dependency. Otherwise commands must be promoted to the slow phase for another append *RPC* call.

```
1112        updated ≜ Map(replicated, LAMBDA entry :
1113          CASE entry.phase = "committed" → entry
1114            □  entry.phase = "slow" → [entry EXCEPT !.phase = "committed"]
1115            □  entry.phase = "fast" →
1116                 LET
1117                   depsReportedByPeers ≜ {
1118                     dep ∈ entry.deps :
1119                       dep.reporter ≠ leader
1120                   }
1121                   canSkipSlowPhase ≜
1122                     ∀ dep ∈ depsReportedByPeers :
1123                       LET
1124                         sameDepReports ≜ {
1125                           d ∈ depsReportedByPeers :
1126                             ∧ d.leader = dep.leader
1127                             ∧ d.pos = dep.pos
1128                         }
1129                       IN
1130                         Cardinality(sameDepReports) ≥ F
1131                 IN  [
1132                   entry EXCEPT !.phase =
1133                     IF canSkipSlowPhase
1134                       THEN "committed"
1135                       ELSE "slow"
1136                 ]
1137        )
1138        promoted ≜ SelectSeq(updated, LAMBDA entry :
1139          ∧ log[server][leader][entry.pos].phase = "fast"
1140          ∧ entry.phase ∈ {"slow", "committed"}
1141        )
1142        committed ≜ TakeWhile(updated, LAMBDA entry :
1143          entry.phase = "committed"
1144        )
1145      IN
```

Only executes this action if promoting or committing commands.

```
1150        ∧ ∨ Len(promoted) > 0
1151          ∨ Len(committed) > 0
```

Updates the *log* to account for new command phases.

```
1155        ∧ log' = [
1156            log EXCEPT ![server][leader] =
1157              ReplaceSlice(@, commitIdx[server][leader], updated)
1158          ]
```

Commit index is updated to the highest contiguous *log* position that is committed at the leader.

```
1163        ∧ commitIdx' = [
1164            commitIdx EXCEPT ![server][leader] =
1165              IF Len(committed) = 0 THEN @ ELSE  Last(committed).pos
1166          ]
```

When promoting an entry from fast phase to either slow or committed, it's possible that not all followers are aware of all the dependencies collected by the leader during the initial append *RPC* call. Therefore, adjust the next index for all followers to ensure that receive an updated set of dependencies from the leader.

```
1175        ∧ nextIdx' = [
1176            nextIdx EXCEPT ![server][leader] =
1177              IF Len(promoted) = 0 THEN @ ELSE
1178                (server :> @[server]) @@ [
1179                  other ∈ Servers ↦ Head(promoted).pos
1180                ]
1181          ]
1182        ∧ UNCHANGED ⟨
1183            currTerm, currState, fastMatchIdx, slowMatchIdx,
1184            fastQuorum, votedFor, recovery, messages
1185          ⟩
```

── FAILURE RECOVERY ACTIONS ──

```
1191  SuspectFailure(server, leader) ≜
```

This action models a follower timeout while waiting for leader messages. Start a recovery process if suspecting of its own failure (*server* = *leader*), or suspecting of some other process failure from which server has a dependency on.

```
1198   ∧ currState[server][leader] = "follower"
1199   ∧ ∨ server = leader
1200     ∨ MaxDepPos(server, leader) > 0
1201   ∧ StartRecovery(server, leader)
1202   ∧ UNCHANGED ⟨
1203       log, nextIdx, fastMatchIdx, slowMatchIdx,
1204       fastQuorum, commitIdx, messages
1205     ⟩

1207  ReconfigureFastQuorum(server, leader) ≜
```

This action models a part of the follower timeout where it determines if the suspected failed leader is part of the server's fast quorum. Since Atlas require a fast quorum of responses before making progress on the fast phase, this quorum must be reconfigured upon failures to restore the server's progress.

Upon identifying a failure of a leader in its fast quorum, the server chooses a different fast quorum that does not contain the failing process. Additionally, it calls for its own recovery to complement pending its pending commands before start using the newly chosen quorum.

```
1221   ∧ server ≠ leader
1222   ∧ currState[server][server] = "leader"
1223   ∧ leader ∈ fastQuorum[server]
1224   ∧ fastQuorum' = [
```

```
1225          fastQuorum EXCEPT ![server] =
1226            ChooseFastQuorum(server, Servers \ {leader})
1227        ]
1228      ∧ StartRecovery(server, server)
1229      ∧ UNCHANGED ⟨
1230          log, nextIdx, fastMatchIdx, slowMatchIdx, commitIdx, messages
1231        ⟩

1233  SendRecoveryRequest(server, leader) ≜
```

As per Raft, elections happen as part of the server's timeout loop. As long as server is a candidate for the leader's *log*, send a "recovery-request" message via broadcast to all peers.

Note that since recoveries start a new term, and that appends from previous terms are discarded, the leader's commit index will remain the same until the end of the recovery process.

```
1243      ∧ currState[server][leader] = "candidate"
1244      ∧ messages' = messages ∪ {
1245          [type       ↦ "recovery-request",
1246           src        ↦ server,
1247           dest       ↦ follower,
1248           leader     ↦ leader,
1249           term       ↦ currTerm[server][leader],
1250           commitIdx  ↦ commitIdx[server][leader]
1251          ] : follower ∈ Servers \ {server}
1252        }
1253      ∧ UNCHANGED ⟨
1254          currTerm, currState, log, nextIdx, fastMatchIdx, slowMatchIdx,
1255          commitIdx, fastQuorum, votedFor, recovery
1256        ⟩

1258  HandleRecoveryRequest(msg) ≜
```

This action models a server receiving a recovery request from a candidate peer for a given leader's *log*.

Messages from older terms are discarded. Otherwise, if receiving a recovery request from a higher term, it abdicates its current state and fallback to the follower state (as per Raft election rules).

```
1267      ∧ msg.term ≥ currTerm[msg.dest][msg.leader]
1268      ∧ currTerm' = [
1269          currTerm EXCEPT ![msg.dest][msg.leader] =
1270            msg.term
1271        ]
1272      ∧ currState' = [
1273          currState EXCEPT ![msg.dest][msg.leader] =
1274            IF msg.term = currTerm[msg.dest][msg.leader]
1275               THEN @ ELSE "follower"
1276        ]
```

Only vote for a single candidate per term. Note that the semantics of f @@ g preserves the domain of f, meaning that if $msg.term$ already belongs to $votedFor[msg.dest][msg.leader]$, the association $(msg.ter :> msg.src)$ is discarded.

```
1283    ∧ votedFor' = [
1284        votedFor EXCEPT ![msg.dest][msg.leader] =
1285          @ @@ (msg.term :> msg.src)
1286      ]
```

Reply back with the server's commit index for the failed leader, and a *log* slice starting from the candidate's commit index. This mechanism allows the candidate to become aware of committed commands it wasn't aware at the time it initiated the recovery process. Also, replies with the voter's maximum known command index for the failed leader.

```
1295    ∧ messages' = messages ∪ {
1296        [type       ↦ "recovery-reply",
1297         src        ↦ msg.dest,
1298         dest       ↦ msg.src,
1299         leader     ↦ msg.leader,
1300         term       ↦ msg.term,
1301         commitIdx  ↦ commitIdx[msg.dest][msg.leader],
1302         voteGranted ↦ votedFor'[msg.dest][msg.leader][msg.term] = msg.src,
1303         maxIdx     ↦ Max(
1304           MaxDepPos(msg.dest, msg.leader),
1305           Len(log[msg.dest][msg.leader])
1306         ),
1307         slice      ↦ SubSeq(
1308           log[msg.dest][msg.leader],
1309           msg.commitIdx + 1,
1310           Len(log[msg.dest][msg.leader])
1311         )
1312        ]
1313      }
1314    ∧ UNCHANGED ⟨
1315        log, nextIdx, fastMatchIdx, slowMatchIdx,
1316        commitIdx, fastQuorum, recovery
1317      ⟩

1319  HandleRecoveryReply(msg) ≜
```

This action models a candidate receiving "recovery-reply" messages from its peers. Replies from different terms are discarded, as well as already registered votes.

```
1325    ∧ currState[msg.dest][msg.leader] = "candidate"
1326    ∧ msg.term = currTerm[msg.dest][msg.leader]
1327    ∧ msg.src ∉ {
1328        reply.reporter : reply ∈ recovery[msg.dest][msg.leader].replies
1329      }
```

Keep track of replies received as well as the maximum command index known among all peers.

```
1334    ∧ recovery' = [
1335        recovery EXCEPT
1336          ![msg.dest][msg.leader].maxIdx = Max(@, msg.maxIdx),
1337          ![msg.dest][msg.leader].replies = @ ∪ {
1338            [reporter   ↦ msg.src,
1339             commitIdx  ↦ msg.commitIdx,
1340             voteGranted ↦ msg.voteGranted,
1341             slice      ↦ msg.slice]
1342          }
1343      ]
1344     ∧ UNCHANGED ⟨
1345         currTerm, currState, log, nextIdx, fastMatchIdx, slowMatchIdx,
1346         commitIdx, fastQuorum, votedFor, messages
1347       ⟩

1349  ClaimLeadership(server, peer) ≜
        This action models the part of the server's timeout loop where it decides to claim ownership of the leader's log if sufficient votes were gathered.
1355     ∧ currState[server][peer] = "candidate"
1356     ∧ LET
1357         replies   ≜ recovery[server][peer].replies
1358         maxIdx    ≜ recovery[server][peer].maxIdx
1359         votes     ≜ {r ∈ replies : r.voteGranted}
1360         commitIdxs ≜ {r.commitIdx : r ∈ replies}

          If elected, the recovery leader computes the current state of the failed leader's log by merging the log slices received by the voters.
1367       MergedSlice[n ∈ Nat] ≜
1368         IF n = 0 THEN ⟨⟩ ELSE
1369           LET
                Define the set of relevant replies for the nth position on the merged log slice as the set of replies with slice length greater or equal to n. Commands in the fast, slow, and committed phase are derived from relevant replies.
1376             relevant  ≜ {r ∈ replies   : n ≤ Len(r.slice)}
1377             fast      ≜ {r ∈ relevant : r.slice[n].phase = "fast"}
1378             slow      ≜ {r ∈ relevant : r.slice[n].phase = "slow"}
1379             committed ≜ {r ∈ relevant :
1380                  ∨ r.slice[n].phase = "committed"
1381                  ∨ r.slice[n].pos ≤ r.commitIdx
1382             }
                The pick function chooses an arbitrary command for the nth position of the merged log slice.
1387             Pick(replySet) ≜ (CHOOSE r ∈ replySet : TRUE).slice[n]
1388           IN
```

To ensure consistency, the following rules must be respected:

– *Commands* in the committed phase MUST be preserved as consensus have been achieved and some processes might already have executed them;

– *Commands* in the slow phase MUST be preserved as their dependency set have been defined by the leader, which may have committed and executed it already.

– *Commands* in the fast phase reported by at least $\mathit{floor}(N/2)$ MUST be preserved;

As per *Altlas*, if $\mathit{floor}(N/2)$ replies contains a command in the fast phase, the recovery leader can't determine if the original leader had received all $\mathit{floor}(N/2)+F$ replies required to promote the command to the slow or committed phases. Therefore, the command MUST be preserved with the SAME set of dependencies.

Atlas proves that the union of the dependencies reported by $\mathit{floor}(N/2)$ servers is guaranteed to be the same as the original leader's set of dependencies if it had decided to skip the slow phase for a given command. Otherwise, the leader promoted the command to slow phase but didn't replicated it to enough servers since the candidate didn't see it. In either case, preserve the command and promote it to the slow phase for another round of consensus.

- Otherwise, nullify the command.

If the command in the nth position is in the fast phase but not yet replicated to $\mathit{floor}(N/2)$ servers, the recovery leader is certain that the original leader didn't promote it to the slow or committed phases yet.

On the other hand, if the nth position is unknown by any of the replying hosts, it means that the leader had reported n as a dependency to one of its peers but it hasn't replicated it anywhere yet.

In both cases, replace the command with *NULL* so that, after committing it, servers can make progress on dependent commands' execution.

$$
\begin{array}{l}
\mathit{MergedSlice}[n-1] \circ \langle \\
\quad \text{CASE } \mathit{committed} \neq \{\} \rightarrow \mathit{Pick}(\mathit{committed}) \\
\qquad \square \quad \mathit{slow} \neq \{\} \qquad \rightarrow \mathit{Pick}(\mathit{slow}) \\
\qquad \square \quad \mathit{Cardinality}(\mathit{fast}) \geq (N \div 2) \rightarrow [\\
\qquad\quad \mathit{phase} \mapsto \text{“slow”}, \\
\qquad\quad \mathit{pos} \quad \mapsto \mathit{Pick}(\mathit{fast}).\mathit{pos}, \\
\qquad\quad \mathit{cmd} \quad \mapsto \mathit{Pick}(\mathit{fast}).\mathit{cmd}, \\
\qquad\quad \mathit{deps} \quad \mapsto \text{UNION } \{r.\mathit{slice}[n].\mathit{deps} : r \in \mathit{fast}\} \\
\qquad\quad] \\
\qquad \square \quad \text{OTHER} \quad \rightarrow [\\
\qquad\quad \mathit{phase} \mapsto \text{“slow”}, \\
\qquad\quad \mathit{cmd} \quad \mapsto \mathit{NULL}, \\
\qquad\quad \mathit{pos} \quad \mapsto \mathit{commitIdx}[\mathit{server}][\mathit{peer}] + n, \\
\qquad\quad \mathit{deps} \quad \mapsto \{\} \\
\qquad\quad] \\
\quad \rangle \\
\text{IN}
\end{array}
$$

As per Atlas, a recovery can only occur after $N - F$ (RQ) nodes reply. As per Raft, a candidate can only get elected with a majority vote of $\mathit{floor}(N/2) + 1$ (EQ).

```
1459        ∧ Cardinality(replies) ≥ RQ
1460        ∧ Cardinality(votes)   ≥ EQ
```

Provided enough replies and votes, the candidate becomes the leader for the failed server's *log*.

```
1465        ∧ currState' = [
1466            currState EXCEPT ![server][peer] =
1467              "leader"
1468          ]
```

The leader's *log* get replaced by the merged slice computed among all the received replies.

```
1473        ∧ log' = [
1474            log EXCEPT ![server][peer] =
1475              ReplaceSlice(@,
1476                commitIdx[server][peer],
1477                MergedSlice[maxIdx]
1478              )
1479          ]
```

Commit index is set to the highest commit index reported by the peers.

```
1484        ∧ commitIdx' = [
1485            commitIdx EXCEPT ![server][peer] =
1486              CHOOSE i ∈ commitIdxs ∪ {@} :
1487                ∀ j ∈ commitIdxs : i ≥ j
1488          ]
```

Replication indexes are reset based on the previous commit index known by the recovery leader, therefore, commencing new rounds of the append *RPC* in order to replicate the merged *log* slice to its peers.

```
1495        ∧ nextIdx' = [
1496            nextIdx EXCEPT ![server][peer] =
1497              (server :> Len(log[server][peer]) + 1) @@ [
1498                other ∈ Servers ↦ commitIdx[server][peer] + 1
1499              ]
1500          ]
1501        ∧ fastMatchIdx' = [
1502            fastMatchIdx EXCEPT ![server][peer] =
1503              (server :> Len(log[server][peer])) @@ [
1504                other ∈ Servers ↦ commitIdx[server][peer]
1505              ]
1506          ]
1507        ∧ slowMatchIdx' = [
1508            slowMatchIdx EXCEPT ![server][peer] =
1509              (server :> Len(log[server][peer])) @@ [
1510                other ∈ Servers ↦ commitIdx[server][peer]
1511              ]
1512          ]
```

```
1513        ∧ UNCHANGED ⟨currTerm, fastQuorum, votedFor, recovery, messages⟩

1515  AbdicateLeadership(server, peer) ≜
```

This action models a part of the recovery leader's timeout loop where it determines if the recovery has been successful (commit index = *log* length).

Considering that the recovery leader never appends new commands to the failed leader's *log*, eventually in a correct system, it will commit the *log* slice collected during its election. At that point, the recovery leader can abdicate its leadership of the failed leader's *log* by resetting its current state to "follower".

The failed leader promotes a new recovery for its own *log* once it comes back, which lets it learn about the consensus made by the recovery leader in its absence.

```
1531    ∧ server ≠ peer
1532    ∧ currState[server][peer]   = "leader"
1533    ∧ commitIdx[server][peer] = Len(log[server][peer])
1534    ∧ currState' = [
1535        currState EXCEPT ![server][peer] =
1536          "follower"
1537      ]
1538    ∧ UNCHANGED ⟨
1539        currTerm, log, nextIdx, fastMatchIdx, slowMatchIdx,
1540        commitIdx, fastQuorum, votedFor, recovery, messages
1541      ⟩

1543  ----------------------------------------------------------------------
```

── INVARIANTS, ACTION PROPERTIES, AND LIVENESS PROPERTIES ──

```
1548  TypeOK ≜
```

Invariant: types must stay valid for every state reached.

```
1552    ∧ CurrTermOK
1553    ∧ CurrStateOK
1554    ∧ LogOK
1555    ∧ NextIdxOK
1556    ∧ MatchIndexesOK
1557    ∧ CommitIdxOK
1558    ∧ FastQuorumOK
1559    ∧ VotedForOK
1560    ∧ RecoveryOK
1561    ∧ MessagesOK

1563  CurrStateTransitions ≜
```

Action Property: state transitions are valid.

Rules:

- Followers stay as followers or transition to candidates;
- Candidates can stay as candidates, transition to follower when receiving an append or recovery reply message with higher term, or transition to leader after winning an election;

- Leader's can stay as leaders, become candidates if suspecting of a failure in its fast quorum, or fallback to follower if receiving an append or recovery request message with higher term.

$\Box[\forall\, server,\, leader \in Servers :$
  LET
    $curr \triangleq currState[server][leader]$
    $next \triangleq currState'[server][leader]$
  IN
    CASE $curr =$ "follower" $\rightarrow next \in \{$"follower", "candidate"$\}$
      $\Box$ $curr =$ "candidate" $\rightarrow next \in \{$"follower", "candidate", "leader"$\}$
      $\Box$ $curr =$ "leader" $\rightarrow next \in \{$"follower", "candidate", "leader"$\}$
$]_{currState}$

── REPLICATION INVARIANTS AND ACTION PROPERTIES ──

$NextIdxIsBoundedByTheLog \triangleq$

Invariant: next index is always $1 \leq next$ index $\leq Len(log) + 1$.

$\forall\, server,\, leader,\, follower \in Servers :$
  $nextIdx[server][leader][follower] \in 1 \,..\, Len(log[server][leader]) + 1$

$MatchIdxsFollowEachOther \triangleq$

Invariant: fast match index is always greater or equal to the slow match index.

$\forall\, server,\, leader,\, follower \in Servers :$
  LET
    $fastIdx \triangleq fastMatchIdx[server][leader][follower]$
    $slowIdx \triangleq slowMatchIdx[server][leader][follower]$
  IN
    $fastIdx \geq slowIdx$

$CommitIdxMovesForward \triangleq$

Action Property: commit index is always moving forward.

$\Box[\forall\, server,\, leader \in Servers :$
  LET
    $curr \triangleq commitIdx[server][leader]$
    $next \triangleq commitIdx'[server][leader]$
  IN
    $next \geq curr$
$]_{commitIdx}$

$LogAlwaysGrow \triangleq$

Action Property: *log* length is either unchanged or it grows.

$\Box[\forall\, server,\, leader \in Servers :$
  $Len(log'[server][leader]) \geq Len(log[server][leader])$
$]_{log}$

$EntryPhaseTransitions \triangleq$

Action Property: command phase transitions are valid.

Rules:

– *Commands* in the fast phase can be promoted to the slow or committed phase;

– *Commands* in the slow phase can only be promoted to the committed phase;

– *Commands* in the committed phase remains in the same phase.

```
1646  □[∀ server, leader ∈ Servers :
1647      ∀ curr ∈ SetOf(log[server][leader]) :
1648        LET next ≜ log′[server][leader][curr.pos]
1649        IN   CASE curr.phase = "fast"      → next.phase ∈ {"fast", "slow", "committed"}
1650               □  curr.phase = "slow"      → next.phase ∈ {"slow", "committed"}
1651               □  curr.phase = "committed" → next.phase = "committed"
1652  ]_log
```

1654 *CommittedEntriesMatch* ≜

A *log* entry committed by a leader, if known to be committed by a follower, must contain the same command and set of dependencies.

```
1659  ∀ server, leader ∈ Servers :
1660    LET
1661      committed ≜ {
1662        entry ∈ SetOf(log[server][leader]) :
1663          ∨ entry.phase = "committed"
1664          ∨ entry.pos ≤ commitIdx[server][leader]
1665      }
1666    IN
1667      ∀ entry ∈ committed :
1668        ∀ follower ∈ Servers \ {server} :
1669          ( ∧ entry.pos ∈ DOMAIN log[follower][leader]
1670            ∧ ∨ entry.pos ≤ commitIdx[follower][leader]
1671              ∨ log[follower][leader][entry.pos].phase = "committed"
1672          ) ⇒
1673            ∧ log[follower][leader][entry.pos].cmd = entry.cmd
1674            ∧ log[follower][leader][entry.pos].deps = entry.deps
```

1676 *CommittedEntriesDoNotChange* ≜

Action Property: commands in the committed phase don't change.

```
1680  □[∀ server, leader ∈ Servers :
1681      LET
1682        committed ≜ {
1683          entry ∈ SetOf(log[server][leader]) :
1684            ∨ entry.phase = "committed"
1685            ∨ entry.pos ≤ commitIdx[server][leader]
1686        }
1687      IN
1688        ∀ curr ∈ committed :
```

```
1689            curr = log'[server][leader][curr.pos]
1690      ]_log

          ── RECOVERY INVARIANTS AND ACTION PROPERTIES ──

1696  CurrTermMovesForward ≜
          Action Property: a leader's current term can only move forward.
1700      □[∀ server, leader ∈ Servers :
1701          currTerm'[server][leader] ≥ currTerm[server][leader]
1702        ]_currTerm

1704  OnlyVoteOncePerTerm ≜
          Action Property: A server only votes for a single candidate on a given term.
1709      □[∀ server, leader ∈ Servers :
1710          ∀ term ∈ DOMAIN votedFor[server][leader] :
1711            LET
1712              curr ≜ votedFor[server][leader][term]
1713              next ≜ votedFor'[server][leader][term]
1714            IN
1715              curr = next
1716        ]_votedFor

1718  NoDoubleLeader ≜
          Invariant: for a given term, only one leader gets elected.
1722      LET
1723        terms ≜ {
1724          currTerm[server][peer] :
1725            server, peer ∈ Servers
1726        }
1727      IN
1728        ∀ ⟨leader, term⟩ ∈ Servers × terms :
1729          LET
1730            leaders ≜ {
1731              server ∈ Servers :
1732                  ∧ currTerm[server][leader] = term
1733                  ∧ currState[server][leader] = "leader"
1734            }
1735          IN
1736            Cardinality(leaders) ≤ 1

1738  RecoveryLeaderDoNotUseFastPhase ≜
          Invariant: recovery leader never submit replicate commands in the fast phase. Commands in
          the fast phase discovered during its election are kept or nullified but, in either case, they are
          promoted to the slow phase for the next round of consensus.
1745      ∀ server, leader ∈ Servers :
1746        server ≠ leader ∧ currState[server][leader] = "leader" ⇒
```

```
1747        ∀ entry ∈ SetOf(log[server][leader]) :
1748          entry.phase ≠ "fast"

1750    NoDivergingRecoveryReplies ≜
        Invariant: candidates only consider one vote per election participant.
1755    ∀ server, leader ∈ Servers :
1756      currState[server][leader] = "leader" ⇒
1757        ∀ peer ∈ Servers :
1758          LET
1759            replies ≜ {
1760              reply ∈ recovery[server][leader].replies :
1761                reply.reporter = peer
1762            }
1763          IN
1764            Cardinality(replies) ≤ 1

        LIVENESS PROPERTIES

1770    LeaderReplicatesItsLog ≜
        In a scenario with bounded terms and log lengths, leaders eventually finish consensus on every
        command.
1775    ◇□∀ leader ∈ Servers :
1776          ∧ commitIdx[leader][leader] = Len(log[leader][leader])
1777          ∧ ∀ follower ∈ Servers :
1778              ∧ commitIdx[follower][leader] = commitIdx[leader][leader]
1779              ∧ ∀ entry ∈ SetOf(log[leader][leader]) :
1780                  ∧ log[leader][follower][entry.pos].cmd = entry.cmd
1781                  ∧ log[leader][follower][entry.pos].deps = entry.deps

1783 ├──────────────────────────────────────────────────────────────────────┤
        ── SPEC BEHAVIOR ──

1788    Init ≜
1789      ∧ InitCurrTerm
1790      ∧ InitCurrState
1791      ∧ InitLog
1792      ∧ InitNextIdx
1793      ∧ InitMatchIndexes
1794      ∧ InitCommitIdx
1795      ∧ InitFastQuorum
1796      ∧ InitVotedFor
1797      ∧ InitRecovery
1798      ∧ InitMessages

1800    AppendCommands ≜
1801      ∃ ⟨leader, cmd⟩ ∈ Servers × Commands :
```

```
1802      AppendCommandToLog(leader, cmd)

1804  ReplicateCommands ≜
1805    ∃ server, leader ∈ Servers :
1806       ∨ SendAppendRequest(server, leader)
1807       ∨ CommitReplicatedCommands(server, leader)

1809  RecoverFailures ≜
1810    ∃ server, leader ∈ Servers :
1811       ∨ SuspectFailure(server, leader)
1812       ∨ ReconfigureFastQuorum(server, leader)
1813       ∨ SendRecoveryRequest(server, leader)
1814       ∨ ClaimLeadership(server, leader)
1815       ∨ AbdicateLeadership(server, leader)

1817  HandleNetworkMessages ≜
1818    ∃ msg ∈ messages :
1819       CASE msg.type = "append-request"    → HandleAppendRequest(msg)
1820         □  msg.type = "append-reply"      → HandleAppendReply(msg)
1821         □  msg.type = "recovery-request"  → HandleRecoveryRequest(msg)
1822         □  msg.type = "recovery-reply"    → HandleRecoveryReply(msg)

1824  Next ≜
1825    ∨ AppendCommands
1826    ∨ ReplicateCommands
1827    ∨ RecoverFailures
1828    ∨ HandleNetworkMessages
1829    ∨ UNCHANGED vars

1831  Fairness ≜
1832    ∧ WF_vars(ReplicateCommands)
1833    ∧ WF_vars(HandleNetworkMessages)

1835  Spec ≜
1836    ∧ Init
1837    ∧ □[Next]_vars
1838    ∧ Fairness
1839 ═══════════════════════════════════════════════════════════════════════

      \* Modification History
      \* Last modified Wed Jun 07 21:20:36 BRT 2023 by erickpintor
      \* Created Thu Aug 12 10:51:52 BRT 2021 by erickpintor
```

```
1 ---------- MODULE Utils ----------
  Utils offer common functionality that is not part of TLA+ std libs.
5 LOCAL INSTANCE Naturals
6 LOCAL INSTANCE Sequences

8 Max(a, b) ≜
    Given two numbers a and b, returns a iff a succeeds b.
12  IF a > b THEN a ELSE b

14 SetOf(seq) ≜
    Given a sequence seq, return a set of all its elements.
18  {seq[x] : x ∈ DOMAIN seq}

20 Last(seq) ≜
    Given a non-empty sequence seq, returns its last element.
24  seq[Len(seq)]

26 TakeWhile(seq, P(_)) ≜
    Given a sequence seq, returns the prefix of elements satisfying P.
30  LET
31    RECURSIVE Take(_)
32    Take(s) ≜
33      IF Len(s) > 0 ∧ P(Head(s))
34       THEN ⟨Head(s)⟩ ∘ Take(Tail(s))
35       ELSE ⟨⟩
36  IN
37    Take(seq)

39 Map(seq, F(_)) ≜
    Given a sequence of elements, return a sequence of elements mapped to the function F.
44  [i ∈ DOMAIN seq ↦ F(seq[i])]

46 ReplaceSlice(seq, prev, replacement) ≜
    Given a sequence and a previous index, replaces the sub sequence delimited from (prev,
    | replacement |] with the given replacement.
51  LET prefix ≜ SubSeq(seq, 1, prev)
52      suffix ≜ SubSeq(seq, prev + Len(replacement) + 1, Len(seq))
53  IN  prefix ∘ replacement ∘ suffix
54 ===============================

  \* Modification History
  \* Last modified Tue Mar 21 12:30:14 BRT 2023 by erickpintor
  \* Created Mon Oct 04 07:27:56 BRT 2021 by erickpintor
```