UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

CAROLINA NACCARATO NUNES

RESILIÊNCIA DE PROTOCOLOS DE *COMMIT* ATÔMICO CENTRALIZADOS EM SISTEMAS DE BANCOS DE DADOS DISTRIBUÍDOS:

Um estudo sobre o protocolo *Two-Phase Commit*

RIO DE JANEIRO
2025

CAROLINA NACCARATO NUNES

RESILIÊNCIA DE PROTOCOLOS DE *COMMIT* ATÔMICO CENTRALIZADOS EM SISTEMAS DE BANCOS DE DADOS DISTRIBUÍDOS:

Um estudo sobre o protocolo *Two-Phase Commit*

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Profa. Silvana Rossetto

RIO DE JANEIRO

2025

CIP - Catalogação na Publicação

N972r

Nunes, Carolina Naccarato
Resiliência de protocolos de commit atômico centralizados em sistemas de bancos de dados distribuídos: um estudo sobre o protocolo Two-Phase Commit / Carolina Naccarato Nunes. -- Rio de Janeiro, 2025.
64 f.

Orientadora: Silvana Rossetto.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2025.

1. Transação distribuída. 2. Problema do comprometimento atômico. 3. Protocolo de commit atômico. 4. Two-Phase Commit. 5. Protocolo de commit atômico não bloqueante. I. Rossetto, Silvana, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

CAROLINA NACCARATO NUNES

RESILIÊNCIA DE PROTOCOLOS DE *COMMIT* ATÔMICO CENTRALIZADOS EM SISTEMAS DE BANCOS DE DADOS DISTRIBUÍDOS:
Um estudo sobre o protocolo *Two-Phase Commit*

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 27 de Fevereiro de 2025

BANCA EXAMINADORA:

gov.br Documento assinado digitalmente
SILVANA ROSSETTO
Data: 09/03/2025 20:11:12-0300
Verifique em https://validar.iti.gov.br

Silvana Rossetto
D.Sc. (Instituto de Computação - UFRJ)

gov.br Documento assinado digitalmente
VIVIAN DOS SANTOS SILVA
Data: 06/03/2025 14:27:35-0300
Verifique em https://validar.iti.gov.br

Vivian dos Santos Silva
Ph.D. (Instituto de Computação - UFRJ)

gov.br Documento assinado digitalmente
EVANDRO LUIZ CARDOSO MACEDO
Data: 06/03/2025 14:32:23-0300
Verifique em https://validar.iti.gov.br

Evandro Luiz Cardoso Macedo
D.Sc. (Instituto de Matemática e Estatística (IME) - UERJ)

AGRADECIMENTOS

Agradeço ao Instituto de Computação da UFRJ por oferecer a estrutura e os recursos que foram fundamentais para minha formação. Expresso minha gratidão à EJCM pela experiência enriquecedora que me preparou para os desafios do mercado de trabalho. Agradeço especialmente à professora Silvana Rossetto pelo acompanhamento, orientação e incentivo não apenas durante este trabalho, mas ao longo de toda a minha trajetória acadêmica. Estendo meus agradecimentos aos diversos professores que, com seu conhecimento e dedicação, contribuíram para minha formação.

RESUMO

O problema do comprometimento atômico reflete um desafio fundamental no processamento de transações em sistemas de bancos de dados distribuídos: garantir seu término consistente na presença de falhas. Este trabalho realiza um estudo bibliográfico sobre a resiliência de protocolos de *commit* atômico, aqueles que se propõem a resolver o problema supracitado, diante de falhas de nós e de comunicação. Os problemas introduzidos por cada classe de falhas são analisados, bem como as condições sob as quais é viável implementar soluções não bloqueantes. O bloqueio de processos é um efeito indesejável da ocorrência de falhas, motivando o estudo de protocolos de *commit* atômico não bloqueantes. Como estudo de caso, o protocolo *Two-Phase Commit* (2PC) foi analisado, evidenciando sua vulnerabilidade ao bloqueio quando o coordenador sofre uma falha de nó ou ocorre uma falha de comunicação. Verificou-se que protocolos de *commit* podem ser resilientes a falhas de um único nó sob recuperação independente, mas não existem soluções não bloqueantes para falhas concorrentes de múltiplos nós ou para falhas de comunicação, incluindo partições simples e múltiplas da rede.

Palavras-chave: transação distribuída; problema do comprometimento atômico; protocolo de *commit* atômico; *two-phase commit*; protocolo de *commit* atômico não bloqueante.

ABSTRACT

The atomic commitment problem represents a fundamental challenge in transaction processing within distributed database systems: ensuring a consistent termination in the presence of failures. This work conducts a literature review on the resilience of atomic commit protocols, those designed to address the aforementioned problem, when faced with node and communication failures. The issues introduced by each failure class are analyzed, as well as the conditions under which non-blocking solutions can be feasibly implemented. Process blocking is an undesirable effect of failures, motivating the study of non-blocking atomic commit protocols. As a case study, the Two-Phase Commit (2PC) protocol was analyzed, highlighting its vulnerability to blocking when the coordinator experiences a node failure or a communication failure occurs. It was found that commit protocols can be resilient to single-node failures under independent recovery; however, no non-blocking solutions exist for concurrent failures of multiple nodes or for communication failures, including both single and multiple network partitions.

Keywords: distributed transaction; atomic commitment problem; atomic commit protocol; two-phase commit; non-blocking atomic commit protocol.

LISTA DE ILUSTRAÇÕES

LISTA DE PSEUDO CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

2PC	Procotolo *Two-Phase Commit*
3PC	Procotolo *Three-Phase Commit*
SGBD	Sistema Gerenciador de Banco de Dados
SGBDD	Sistema Gerenciador de Banco de Dados Distribuído
ACID	Atomicidade, Consistência, Isolamento e Durabilidade
Log TD	*Log* de Transações Distribuídas
WAL	*Write-Ahead Logging*
Condição NB	Condição Não Bloqueante
q_i	Estado inicial do processo i
w_i	Estado de espera do processo i
a_i	Estado de *abort* do processo i
c_i	Estado de *commit* do processo i
g	Estado global antes da transição
g'	Estado global imediatamente alcançável a partir de g
e	Um estado local arbitrário
m	Mensagem recebida por um estado local
M	Conjunto de mensagens recebidas por um estado local
p	Um processo arbitrário
DTP	*Distributed Transaction Processing*
PA	*Presumed Abort*
PC	*Presumed Commit*

LISTA DE SÍMBOLOS

$\delta(a, b)$	Letra grega delta, representando uma transição do estado a para o estado b na modelagem de um protocolo de *commit* atômico

SUMÁRIO

1 INTRODUÇÃO

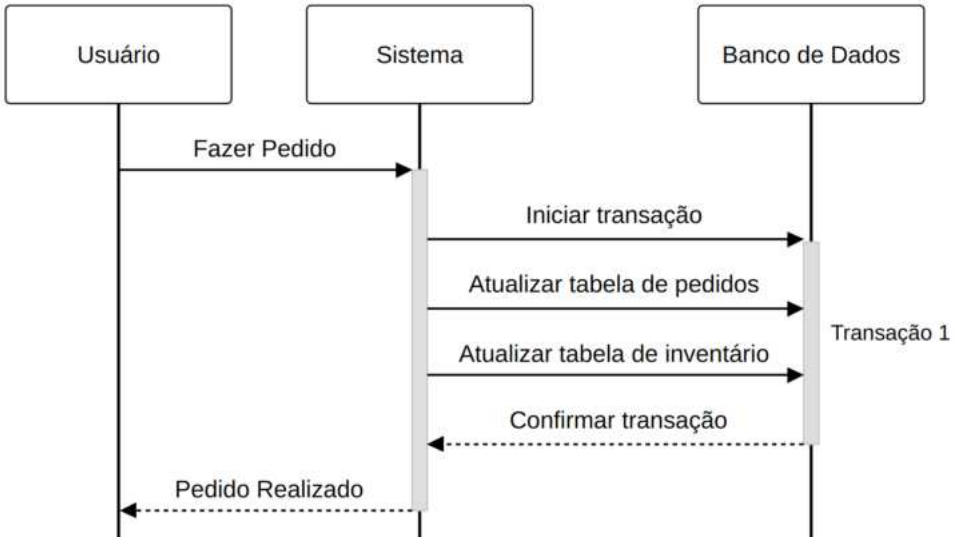
Uma das principais motivações por trás do uso de sistemas de banco de dados distribuídos é a necessidade de compartilhamento de recursos (COULOURIS; DOLLIMORE; KINDBERG, 2011). Como consequência, cresce a demanda por soluções que garantam a gestão eficiente de dados em ambientes distribuídos, motivando o uso de sistemas de bancos de dados distribuídos. Um banco de dados distribuído é uma coleção de bancos de dados logicamente inter-relacionados, hospedados em computadores autônomos (ÖZSU; VALDURIEZ, 1999). Estes computadores, também chamados de nós, podem ficar temporariamente indisponíveis devido a falhas. Os nós trocam mensagens através de uma rede em comum, que também está sujeita a falhas, podendo causar atrasos na entrega de mensagens, ou até mesmo levar à perda destas. Em razão disso, uma operação processada por múltiplos nós deve ser sensível à localização dos dados na rede e resiliente a falhas de nós e na rede (GRAY; TRAIGER; GALTIERI, 1982). Um sistema gerenciador de banco de dados distribuído (SGBDD), por sua vez, é um software que gerencia tais operações e acessos simultâneos a dados em um banco de dados distribuído de maneira transparente, criando a ilusão para o usuário final de que os dados estão armazenados em um único local.

As transações são elementos fundamentais para um sistema gerenciador de banco de dados, pois operações sobre dados são geralmente realizadas por meio de transações (TANENBAUM; STEEN, 2017). No caso de um banco de dados distribuído, cada nó processa uma parte da mesma transação, de forma que todos devem concordar com o resultado final, de sucesso (*commit*) ou fracasso (*abort*) da transação, garantindo seu término consistente. Essa propriedade é conhecida como atomicidade, que exige que todas as operações sobre os dados sejam realizadas com sucesso, ou que todas sejam canceladas. Com o objetivo de garantir a atomicidade e consistência na execução de transações distribuídas, **protocolos de *commit* atômico** são utilizados. Estes algoritmos devem lidar não apenas com o desafio de garantir o término consistente da transação, mas também com as falhas que podem ocorrer durante a execução da mesma.

1.1 MOTIVAÇÃO

Para ilustrar a aplicabilidade dos conceitos discutidos, consideremos um sistema de comércio eletrônico que realiza o processamento de pedidos online. Inicialmente, suponha-se um cenário centralizado, onde todas as informações da aplicação estão armazenadas em um banco de dados monolítico. Nesse modelo, as tabelas “pedido” e “inventário” coexistem em um único sistema gerenciador de banco de dados (SGBD). Quando um usuário efetua um pedido, uma única transação é criada para processar essa operação. O SGBD garante

Figura 1 – Diagrama ilustrando o processo de solicitação de um pedido em um sistema de banco de dados monolítico.



que todas as etapas do pedido, incluindo a criação do registro do pedido e a atualização do estoque, sejam realizadas de maneira atômica, garantindo a consistência dos dados. Esse processo pode ser representado pelo seguinte diagrama ilustrado pela Figura 1.

Agora, considere uma abordagem baseada em microsserviços, especificamente um modelo *database per service*, no qual cada microsserviço gerencia seu próprio banco de dados. Neste contexto, ao processar um pedido, a transação não pode ser realizada de forma única e centralizada, pois cada serviço é responsável exclusivamente pelos seus próprios dados. Nesse modelo, "Pedido" e "Inventário" são agora microsserviços independentes. No exemplo ilustrado pela Figura 2, a transação principal é decomposta em duas subtransações independentes: transação 1, responsável pelo registro do pedido; e transação 2, encarregada da atualização do estoque.

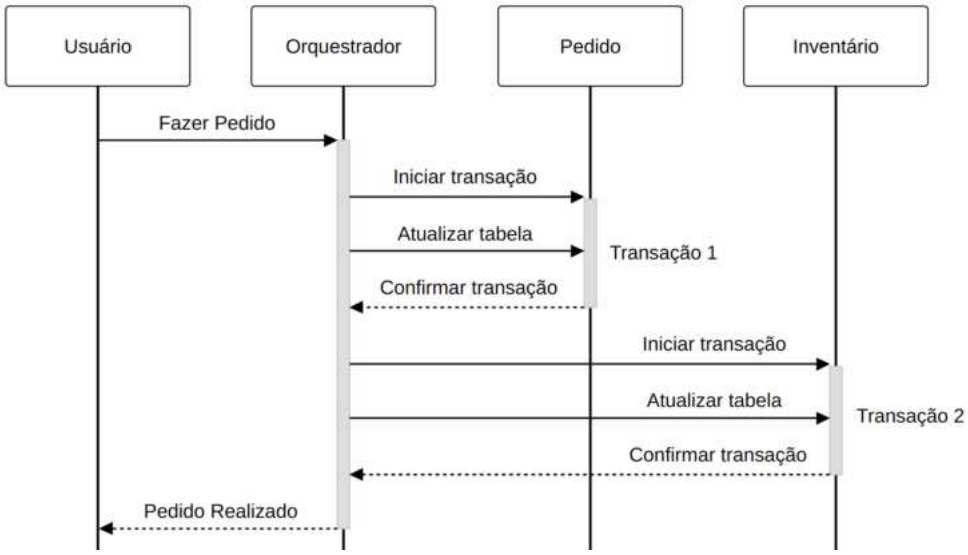
Além disso, observa-se a introdução de um agente denominado Orquestrador na Figura 2. Esse orquestrador, ou coordenador, representa os protocolos de *commit* atômico que serão analisados ao longo deste trabalho. Sua função é garantir a correta subdivisão da transação principal em múltiplas subtransações e coordenar sua execução para assegurar a sincronização e o consenso entre os processos participantes.

1.2 O MODELO DE TRANSAÇÕES

A menor unidade de trabalho e principal componente de um SGBDD são as transações, que são sequências de operações de leitura e escrita sobre dados que transformam o banco de dados de um estado consistente para outro (GRAY; REUTER, 1992). A transação distribuída[1], quando processada pelo gerenciador de transações, é subdividida

[1] Sempre que o termo "transação" for utilizado, deve-se entender que se trata de uma transação distribuída.

Figura 2 – Diagrama ilustrando o processo de solicitação de um pedido em um sistema de banco de dados no modelo de microsserviços.



e encaminhada para os nós apropriados de acordo com as informações que deseja acessar. A subtransação é, então, processada como se fosse uma transação local, e seu resultado é encaminhado para o nó de origem, onde a operação será concluída (BERNSTEIN; HADZILACOS; GOODMAN, 1987). As subdivisões da transação principal não possuem hierarquia entre si, de forma que o *abort* em qualquer instância implica no fracasso de toda a operação.

As transações são caracterizadas pelas suas propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade), propostas por (HAERDER; REUTER, 1983). A atomicidade está associada a dois aspectos fundamentais. O primeiro, a atomicidade relacionada a concorrência, assegura que execuções concorrentes de transações não produzam dados inconsistentes, garantindo que seus efeitos sejam equivalentes ao de uma execução sequencial (BERNSTEIN; HADZILACOS; GOODMAN, 1987). Essa propriedade é informalmente conhecida como "tudo ou nada", pois garante que a transação seja uma operação lógica indivisível, ou seja, todas as operações são realizadas com sucesso ou nenhuma delas é realizada.

A consistência garante que a transação respeitará as restrições de integridade do banco de dados, que são um conjunto de regras que garantem a validade dos dados, dado as regras de negócio. Somado à isso, o espaço de trabalho das transações é privado, mantendo seu isolamento. Para que isso seja possível, no momento que uma transação é criada, esta recebe uma cópia conceitual dos dados do sistema, que é chamado de contexto. Nenhum agente externo consegue acessar as transformações sobre estes dados enquanto a transação não finalizou sua execução. A finalização de uma transação é marcada pela ação de *commit*, ou confirmação, tornando os seus efeitos permanentes e irreversíveis, isto é, duráveis; ou pela ação de *abort*, que desfaz todas as operações realizadas pela transação,

retornando o banco de dados ao estado anterior a ela.

1.3 FALHAS

Diversas falhas podem comprometer a execução de uma transação, impedindo sua conclusão bem-sucedida. O pior efeito possível de uma falha é quando esta resulta em um banco de dados inconsistente, algo intolerável para a maioria das aplicações. Uma consequência tolerável, porém indesejável, é o *bloqueio* do processamento da transação até que a falha tenha sido reparada. O bloqueio preserva a consistência ao reduzir a *disponibilidade* (de processamento) do sistema distribuído. O efeito menos prejudicial de uma falha é tornar indisponível apenas os processos falhos (SKEEN, 1982).

Sabendo que algoritmos de *commit* atômico devem lidar com falhas de nós e na rede, as classes de falha de interesse são as falhas de nós e de comunicação, listadas em ordem de dificuldade. As *falhas de nós*, também conhecidas como falhas de sistema ou de processo, ocorrem quando um ou mais nós do sistema distribuído falham, ficando inoperantes durante seu período de recuperação. Podem ser totais, quando todos os nós ficam indisponíveis, ou parciais, quando um ou mais nós falham. Já as falhas de comunicação consideradas são aquelas causadas pela partição da rede em dois ou mais grupos de nós, interrompendo por completo a comunicação entre diferentes grupos (SKEEN, 1982; HADZILACOS, 1990).

1.4 O PROBLEMA DO COMPROMISSO ATÔMICO

Um dos principais desafios enfrentados por sistemas gerenciadores de banco de dados distribuídos é garantir a consistência dos dados. Para garantir a atomicidade de uma transação que acessa dados distribuídos, todos os nós envolvidos no processamento desta devem coordenar suas execuções de forma que o mesmo resultado final, de *commit* ou *abort*, seja alcançado por unanimidade. Esse desafio dá origem ao que é chamado de **problema do compromisso atômico** (BERNSTEIN; HADZILACOS; GOODMAN, 1987; HADZILACOS, 1990), que busca soluções que possibilitem essa coordenação mesmo na presença de falhas.

O estudo do problema do compromisso atômico busca não apenas definí-lo, como também derivar as condições necessárias que um algoritmo deve satisfazer para preservar a atomicidade e consistência de transações na presença de falhas. Os **protocolos de *commit* atômico**, foco do presente trabalho, são o conjunto de algoritmos que implementam soluções para tais condições. Estes algoritmos são classificados com base no paradigma de comunicação utilizado: *centralizado* e *descentralizado* (SKEEN, 1982). A análise proposta será restrita aos protocolos centralizados, pois servem como base para a compreensão de abordagens descentralizadas.

Um protocolo de *commit* atômico, quando centralizado, faz uso de um processo, o *coordenador*, para direcionar o processamento da transação para todos os demais, chamados de *participantes*. Há uma diferença entre os protocolos executados, dada a função que o processo desempenha − sendo o protocolo do coordenador distinto do protocolo dos participantes. Durante cada fase do algoritmo, o coordenador envia a mesma mensagem para todos os participantes e aguarda as suas respostas. Os participantes, por sua vez, se comunicam exclusivamente com o coordenador (SKEEN, 1981).

Pela sua simplicidade e ampla utilização, os protocolos ***Two-Phase Commit*** (GRAY, 1978) e ***Three-Phase Commit*** (SKEEN, 1982) se destacam. Estes algoritmos se diferenciam quanto à propriedade de bloqueio: o primeiro é bloqueante, enquanto o segundo não é. A característica bloqueante de protocolos de *commit* é indesejada e, para compreender o porquê desta afirmação, é fundamental entender como as transações utilizam os bloqueios. O uso de bloqueios é uma estratégia importante para garantir o controle de concorrência em operações sobre dados distribuídos (e compartilhados). Uma transação que mantém um bloqueio de leitura ou escrita sobre um objeto de dados garante que este não será alterado por outra transação. O bloqueio de escrita também garante que o objeto não será lido por outra transação em uma sequência de eventos que causaria um conflito. Em razão disso, o bloqueio de uma transação pode causar indisponibilidade e congestionamento do sistema, uma vez que o período de tempo em que um nó precisa para se recuperar é estocástico, podendo ser muito maior do que o tempo de vida de uma transação (CHORAFAS, 1998).

1.5 OBJETIVO

Este trabalho tem como objetivo apresentar os desafios associados ao problema do compromisso atômico no processamento de transações distribuídas. Explora-se as classes de falhas que afetam os protocolos de *commit*, destacando os aspectos que viabilizam ou impedem a resiliência desses protocolos, com ênfase nas situações em que soluções não bloqueantes não são viáveis; discute-se como é possível satisfazer as condições impostas pelo problema do compromisso atômico diante de falhas que impedem o prosseguimento da execução de protocolos de *commit*, guiadas por um protocolo de terminação; e, finalmente, aborda-se as condições sob as quais um protocolo não bloqueante pode existir, apresentando uma metodologia para a criação destes, ilustrando a concepção do protocolo *Three-Phase Commit*.

Para isso, duas principais fontes são consultadas: (SKEEN, 1982), que consolida uma análise detalhada das classes de falhas, bem como embasa as condições necessárias e suficientes para a existência de protocolos de *commit* não bloqueantes com a definição do teorema Não Bloqueante; e (BERNSTEIN; HADZILACOS; GOODMAN, 1987), que complementa essa análise, fornecendo a definição formal do problema do compromisso atômico

e a descrição detalhada do protocolo *Two-Phase Commit* e do protocolo de terminação associado. Desde a sua concepção, o protocolo 2PC tem sido objeto de estudo e aprimoramento, resultando no desenvolvimento de diversas variantes, das quais o seu estudo não está no escopo deste trabalho. A escolha por analisar fontes seminais justifica-se pelo objetivo didático de apresentar o protocolo 2PC em sua forma mais básica. Acredita-se que o estudo do protocolo em sua essência proporciona uma base sólida para a compreensão de suas variantes e para o desenvolvimento de novas soluções.

1.6 METODOLOGIA E ESTRUTURA DO TRABALHO

No Capítulo 2, define-se formalmente o problema do compromisso atômico como um conjunto de condições a serem atendidas. Com base nisso, apresenta-se a execução correta (considerando inicialmente um cenário sem falhas) do protocolo *Two-Phase Commit* como uma solução para esse problema. Esse protocolo, simples e amplamente utilizado, serve como fio condutor para a discussão proposta neste trabalho, conduzindo-a para um âmbito concreto e aplicável.

A resiliência de protocolos de *commit* para diferentes classes de falhas refere-se à sua capacidade de implementar soluções não bloqueantes mesmo na presença dessas falhas. No Capítulo 3, investigam-se as falhas de nós e de comunicação, bem como suas respectivas subclasses, apresentando-se resultados que indicam a existência ou a inexistência de protocolos resilientes.

Baseando-se nos resultados apresentados previamente, o Capítulo 4 aprofunda-se nas condições necessárias e suficientes para a existência de protocolos não bloqueantes descritas pelo Teorema Não Bloqueante (SKEEN, 1982). Com isso, apresenta-se uma metodologia que transforma um protocolo bloqueante em não bloqueante, demonstrada com o *Two-Phase Commit*, permitindo a compreensão do surgimento da família de protocolos de *commit* não bloqueantes mais simples: os protocolos de três fases.

Por fim, o Capítulo 5 conclui o trabalho, abordando os principais resultados apresentados e sugerindo direções para trabalhos futuros.

2 O PROBLEMA DO COMPROMISSO ATÔMICO E O PROTOCOLO *TWO-PHASE COMMIT*

"Uma tarefa crítica na execução de uma transação em um sistema de banco de dados distribuído é garantir seu término consistente" (HADZILACOS, 1990). Os nós envolvidos no processamento de uma transação devem alcançar uma decisão consistente apesar da ocorrência de falhas, devendo concordar na sua confirmação ou fracasso. Este é, em poucas palavras, o problema do compromisso atômico.

Os protocolos de *commit* atômico foram desenvolvidos para solucionar esse problema, dentre os quais se destacam o *Two-Phase Commit* (2PC) (GRAY, 1978) e o *Three-Phase Commit* (3PC) (SKEEN, 1982), e suas variantes. A seguir, são descritas suas principais características.

- ***Two-Phase Commit* (2PC):** O 2PC é o protocolo de *commit* mais utilizado em bancos de dados distribuídos, servindo de base para muitos outros protocolos (ELBAGIR; KHALID; KHANFAR, 2016). Ele consiste em duas fases — votação e decisão — durante as quais mensagens são trocadas entre um coordenador e os participantes da transação. Esse protocolo foi incorporado em diversos padrões para transações, como o DTP da X/Open (SPECIFICATION, 1991), além de ter sido implementado em diversos sistemas de banco de dados comerciais (ABDALLAH; GUERRAOUI; PUCHERAL, 1998). O protocolo 2PC, porém, é vulnerável, sobretudo, a falhas do cordenador e falhas de comunicação;

- ***Three-Phase Commit* (3PC):** O protocolo 3PC foi proposto para resolver o problema de bloqueio do 2PC. Este protocolo é não bloqueante em razão de uma fase extra, chamada de fase de pré-*commit*, entre as duas fases do protocolo 2PC. O 3PC executa as operações em três fases: fase de votação, fase de pré-*commit* e fase de decisão (SKEEN, 1982). Ao introduzir a fase de pré-*commit*, o 3PC garante que, mesmo em casos de falha do coordenador, a transação não ficará bloqueada. No entanto, a adição de uma nova fase causa o aumento significativo no custo de mensagens trocadas quando comparado ao 2PC.

Após o desenvolvimento do 2PC, destacam-se duas variantes desse protocolo, que foram propostas com o objetivo de melhorar o seu desempenho, buscando reduzir o número de mensagens trocadas entre o coordenador e os participantes, bem como a quantidade de registros no *log* (MOHAN; LINDSAY; OBERMARCK, 1986). São elas:

- ***Presumed Abort* (PA):** Esta variante do 2PC foi otimizada para lidar com transações somente de leitura e transações parcialmente somente de leitura, ou seja,

aquelas em que alguns dos processos não realizam atualizações no banco de dados. O protocolo presume que, caso uma falha ocorra, a transação será abortada;

- ***Presumed Commit* (PC):** Variante otimizada para transações de atualização, presumindo que, na presença de falhas, a transação será confirmada. Minimiza o número de mensagens necessárias em transações de atualização.

Este capítulo apresenta uma definição formal do problema do compromisso atômico e caracteriza a classe de algoritmos de *commit* atômico como aqueles projetados para resolver esse problema. O foco está no protocolo 2PC, amplamente utilizado em sistemas de banco de dados distribuídos devido à sua simplicidade e eficácia. Em contraste, o protocolo 3PC, embora proposto para resolver o bloqueio do 2PC, não é comumente adotado devido ao aumento da complexidade de mensagens, o que prejudica o seu desempenho (ÖZSU; VALDURIEZ, 1999). EM seguida, o 2PC em sua versão mais simples é apresentado em um cenário sem falhas como uma solução inicial para o problema do compromisso atômico. Discute-se o seu funcionamento normal, como o protocolo atende às condições do compromisso atômico e de que forma ele garante a finalização da transação, explorando seus cenários de sucesso e falha.

2.1 DEFINIÇÃO DO PROBLEMA DO COMPROMISSO ATÔMICO

O modelo computacional assumido no contexto do problema do compromisso atômico envolve uma coleção de processos que executam partes da mesma transação e são capazes de comunicar-se através de mensagens. Em qualquer instante de execução do algoritmo, um processo está operante ou inativo em razão de uma falha. Quando está operacional, o processo está seguindo exatamente as ações especificadas pelo seu programa. Enquanto estiver inativo, o processo não realiza nenhuma ação. Não considera-se a possibilidade de um processo estar operacional e, ao mesmo tempo, apresentar erros em sua execução. A transição de operante para inativo pode ocorrer em qualquer momento, evento conhecido como falha de nó. A transição de inativo para operante é feita mediante um protocolo de recuperação, cujo objetivo é restaurar um estado do qual o processo pode continuar a sua execução. Quando essa transição ocorre, diz-se que a falha de nó foi reparada. Processos *corretos* são aqueles que nunca ficaram indisponíveis como resultado de falhas; caso contrário, são falhos. Observe que um processo pode ser falho e estar operante (HADZILACOS, 1990). ALém disso, considera-se que somente um processo é executado em cada nó.

Um protocolo de terminação, por sua vez, é utilizado quando uma falha impede o prosseguimento da execução do algoritmo, com objetivo de evitar que processos sejam bloqueados. Um protocolo de *commit* deve possuir protocolos de recuperação e terminação associados, que são independentes entre si. Além disso, admite-se a possibilidade de falhas

de comunicação. Quando um processo espera uma mensagem, é definido um período de tempo limite, chamado de *timeout*, que limita quanto tempo o primeiro esperará pela mensagem. Uma falha de comunicação ocorre quando dois processos, ambos operacionais, não conseguem comunicar-se − quando dois processos, um ativo e outro inativo, não conseguem comunicar-se, diz-se que uma falha de nó ocorreu (HADZILACOS, 1990).

Dada a descrição do modelo computacional, define-se o problema do compromisso atômico da seguinte forma. Seja um conjunto de processos dos quais cada um possui uma variável de entrada chamada de "voto", que aceita os valores "sim" ou "não"; e uma variável de saída chamada "decisão", que aceita os valores "*commit*", "*abort*" ou "indefinido". A escrita na variável de "decisão", que possui valor inicial "indefinido", é final. O problema do compromisso atômico busca elaborar algoritmos, conhecidos como **protocolos de commit atômico**, que coordenam tal conjunto de processos com o objetivo de garantir o cumprimento das condições a seguir (HADZILACOS, 1990; BERNSTEIN; HADZILACOS; GOODMAN, 1987).

Condição 1. *A mesma decisão deve ser alcançada por todos os processos que atingirem um estado de decisão;*

Condição 2. *A decisão, uma vez efetuada, é irreversível;*

Condição 3. *A confirmação (commit) global é decidido somente se todos os votos forem "sim";*

Condição 4. *Na ausência de falhas e caso todos os votos sejam "sim", todos os processos decidem pelo commit;*

Condição 5. *Se todas as falhas de nós e de comunicação forem reparadas e não surgirem novas falhas por um período suficientemente longo, todos os processos chegam a uma decisão.*

A Condição 1 exige que a escrita na variável "decisão" seja consistente entre todos os processos. A Condição 2, por sua vez, reforça que as ações de *commit* e *abort* são finais, refletindo a irreversibilidade destas operações. A Condição 3 afirma que a decisão global pelo *commit* deve ser tomada por consenso. A decisão global refere-se à decisão final tomada pelo coordenador, com base nos votos de todos os processos participantes. Em um cenário ideal, a Condição 4 garante que uma transação será concluída com sucesso, descartando, assim, a possibilidade de um protocolo que sempre decida pelo *abort*. Por fim, a Condição 5 identifica as circunstâncias sob as quais os processos são obrigados a chegar a uma decisão.

2.2 O PROTOCOLO *TWO-PHASE COMMIT*

Desenvolvido no final dos anos 1970 por (GRAY, 1978; LAMPSON; STURGIS, 1976; LINDSAY, 1979), o protocolo *Two-Phase Commit* (2PC) é o protocolo de *commit* mais simples que permite *aborts* unilaterais. A seguir, será descrito o seu funcionamento correto, desconsiderando a possibilidade de falhas. No próximo capítulo, este protocolo será utilizado como base para o estudo de falhas que acometem protocolos de *commit*.

2.2.1 Descrição

O procedimento de *commit* no protocolo 2PC inicia-se quando o processo de origem, o coordenador, envia mensagens aos demais participantes para solicitar seus votos. O coordenador, tendo recebido as respostas dessa solicitação, passa a ter uma lista completa de todos os processos participantes da transação e inicia o algoritmo. Esta lista é consumida pelo coordenador a cada rodada de mensagens enviada por este aos demais. O controle centralizado que o coordenador exerce sobre os demais participantes permite que os últimos realizem seu processamento de forma concorrente. Esse comportamento é consequência de seu paradigma de comunicação centralizado no coordenador, característica que faz com que essa versão do protocolo seja classificada como centralizada.

Na primeira fase do algoritmo, a *fase de votação*, o coordenador distribui a transação para os participantes com o objetivo de verificar a capacidade de cada processo em confirmar sua parte da transação, que é manifestada através do voto. Cada processo participante, após votar "sim", torna-se incerto, pois entra no seu *período de incerteza*, em que não pode determinar se a transação será confirmada ou abortada. Por outro lado, se um participante votar "não", este aborta unilateralmente. O coordenador, recebendo pelo menos um voto "não", é responsável por propagar a decisão de *abort* para aqueles processos que votaram "sim" e encerra a sua execução. A segunda fase só se inicia se todos os votos forem "sim". Nesta etapa, a *fase de decisão*, o coordenador coleta todos os votos e informa os processos participantes do resultado. Se a transação for confirmada, é nesta fase que as alterações são de fato executadas.

Podemos organizar o funcionamento normal do algoritmo em um cenário sem falhas por meio de uma lista numerada de etapas (BERNSTEIN; HADZILACOS; GOODMAN, 1987), complementando a breve descrição fornecida acima.

1. Coordenador envia mensagens solicitando o voto dos participantes;
2. Participantes devem votar "sim" ou "não" para o *commit* da transação;
   (a) Se o voto for "não", este aborta unilateralmente;
   (b) Se o voto for "sim", o participante entra em espera aguardando a decisão do coordenador.

3. Coordenador coleta os votos de todos os participantes e toma uma decisão;
   (a) Pelo menos um participante votou "não" e coordenador decide abortar;
   (b) Todos votaram "sim" e coordenador decide abortar;
   (c) Todos votaram "sim" e coordenador decide confirmar.
4. Participantes que votaram "sim" aguardam pela mensagem de "*commit*" ou "*abort*" do coordenador.
   (a) Se recebem "*abort*", abortam e encerram;
   (b) Se recebem "*commit*", realizam o *commit* e encerram.

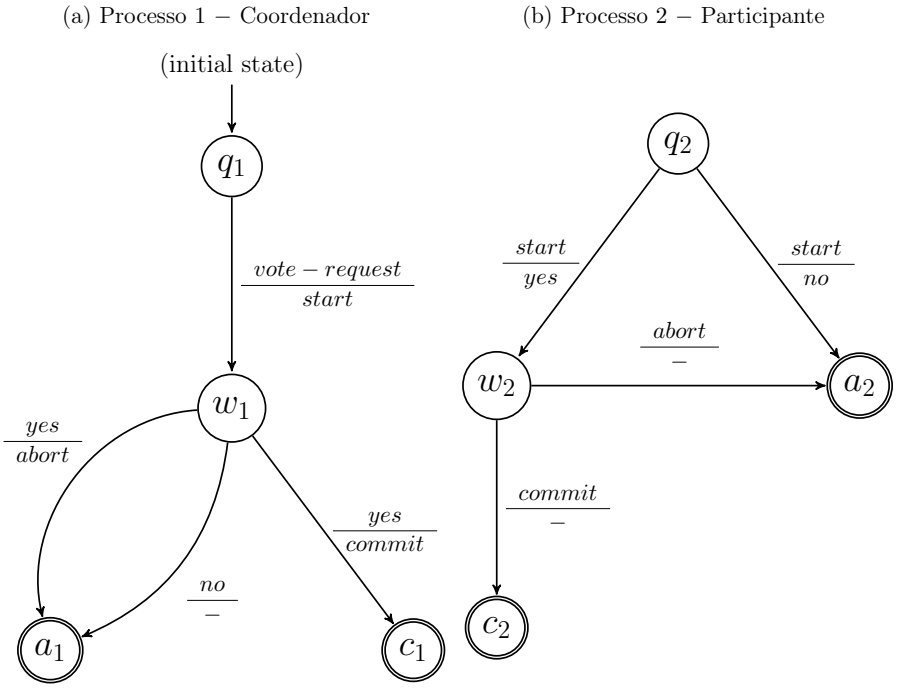
Note que existem três caminhos distintos que levam às ações finais: Pelo menos um participante votou "não", coordenador decide *abort* (etapas 1 - 2. (a) - 3. (a) - 4. (a)); participantes votam "sim", coordenador decide *abort* (etapas 1 - 2. (b) - 3. (b) - 4. (a)); participantes votam "sim", coordenador decide *commit* (etapas 1 - 2. (b) - 3. (c) - 4. (b)). O aprofundamento dessas possibilidades será feito na Seção 2.2.2, após a definição do modelo formal utilizado para representar algoritmos de *commit*, o qual será apresentado seguir.

Esse modelo, proposto por (SKEEN, 1982), descreve a execução da transação em cada processo como autômatos finitos e não determinísticos que consideram as falhas como um tipo especial de transição de estado. Além de ilustrar visualmente e melhor descrever algoritmos de *commit*, o modelo é utilizado como ferramenta para a discussão dos limites da tolerância a falhas e para verificar a sua corretude.

A Figura 3 retrata o algoritmo 2PC com dois processos, ilustrando os *protocolos locais* dos processos coordenador e participante, utilizando o modelo formal de SKEEN. Nesse modelo, a especificação de um protocolo é formalizada como um conjunto de N autômatos, cada um representando um processo. Cada processo executa o seu protocolo local, que possui os seus estados locais. Para identificar o processo i, os seus estados locais e mensagens trocadas, estes são sobrescritos com $i \in \{1, \ldots, N\}$. No caso do 2PC, existem dois tipos de protocolos locais genéricos: aquele executado pelo coordenador, que é sempre o primeiro processo; e o executado pelos participantes. Além disso, existem quatro tipos de estados locais: q_i, o estado inicial; w_i, o estado de espera; a_i, o estado de *abort* ou fracasso; e c_i, o estado de *commit* ou confirmação. Os estados finais são representados por círculos duplos (SKEEN, 1981).

A rede é abstraída como o canal de comunicação que permite a troca de mensagens entre os nós e é capaz de detectar falhas através de ações de *timeout*, por exemplo. Uma transição de estado é provocada pelo recebimento de uma ou mais mensagens, causando uma mudança de estado local e o envio de zero ou mais mensagens. A transição só ocorre quando todas as mensagens necessárias foram enviadas. Mensagens recebidas durante

Figura 3 – Protocolos locais do algoritmo *Two-Phase Commit* centralizado com $N = 2$.



Fonte: SKEEN (1982, figura 3.1, p. 45)

uma transição de estado são aquelas acima da linha horizontal, enquanto as enviadas ficam abaixo.

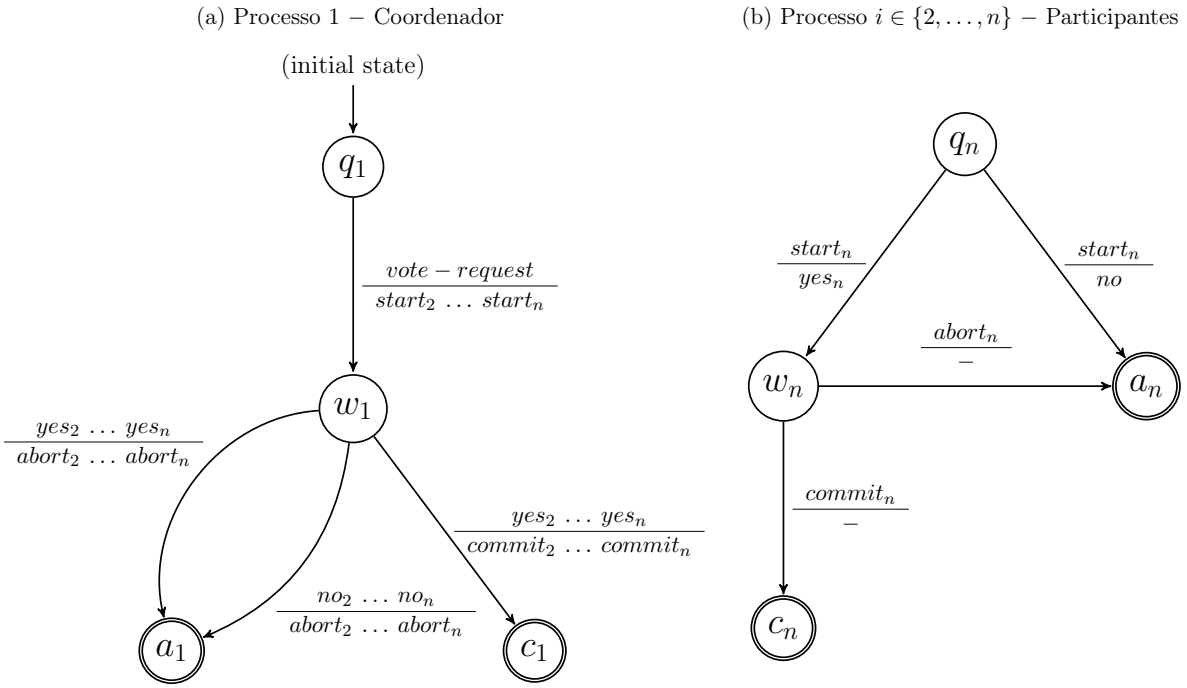
O gerenciador de transações é responsável por realizar a requisição ($vote - request$) que dá início à transação. O coordenador, recebendo esse sinal externo, envia uma requisição de voto ($start$) para o participante e realiza a transição do estado inicial q_1 para o estado de espera w_1, representado pela função $\delta(q_1,\ w_1)$. Nesse momento, o controle da transação passou para o Processo 2, que deve escolher de maneira não determinística entre responder "sim", aceitando a transação; ou "não", para abortá-la unilateralmente. O protocolo continua com rodadas de trocas de mensagens entre coordenador e participante até que ambos alcancem estados finais: *commit* (c_i) ou *abort* (a_i) (SKEEN, 1982).

A Figura 4 representa a modelagem genérica do protocolo 2PC executado com N processos.

2.2.2 Cenários de Finalização

A partir da definição do modelo formal que descreve protocolos de *commit*, é possível discutir as formas pelas quais o algoritmo 2PC pode finalizar sua execução. Retomando a sequência de etapas do algoritmo, a fase de votação abrange as etapas 1 e 2, enquanto a fase de decisão corresponde às etapas 3 e 4. O período de incerteza de um participante começa quando ele vota "sim" (etapa 2) e termina quando recebe a decisão do coordenador (etapa 4). Sabendo disso, é possível analisar os diferentes cenários que podem ocorrer

Figura 4 – Protocolos locais do algoritmo *Two-Phase Commit* centralizado com N processos.



Fonte: SKEEN (1981, figura 2, p. 135)

durante a execução do protocolo, dependendo das decisões tomadas pelos participantes e pelo coordenador.

2.2.2.1 Pelo menos um participante vota "não", coordenador decide "*abort*"

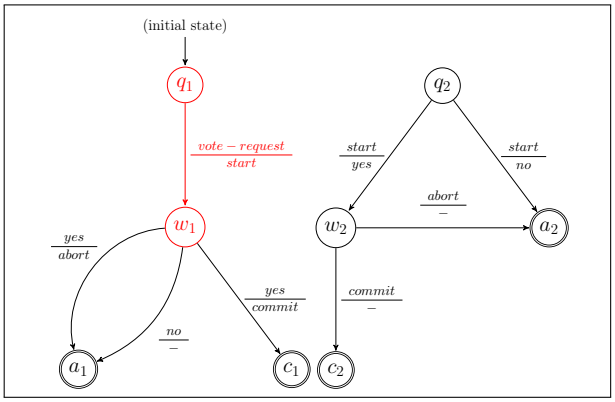
O 2PC é o protocolo de *commit* mais simples que permite o *abort* unilateral (SKEEN, 1982), cenário este representado pelas figuras Figura 5 e Figura 6, que ilustram o algoritmo operando com dois e quatro processos respectivamente. Assim que um participante vota "não", possui autonomia para abortar unilateralmente, conforme ilustrado pela transição $\delta(q_2,\ a_2)$ em ambas imagens. Este comportamento é consequência da Condição 1, que exige que a mesma decisão seja tomada por todos processos; e da Condição 2, que permite o *abort* unilateral somente para os processos que não votaram "sim". A principal diferença entre os dois cenários está na ação tomada pelo coordenador ao receber um voto "não". Apesar de um único voto "não" ser suficiente para determinar a decisão de "*abort*", observe que o coordenador deve aguardar o recebimento de todos os votos antes de prosseguir. Isso garante que ele conheça quais participantes votaram "sim" e possa informá-los adequadamente da decisão final.

Quando o protocolo opera com dois processos, o coordenador não precisa preocupar-se com o envio de mensagens, uma vez que o único processo participante já abortou. O coordenador recebe o único voto "não", aborta e encerra, conforme ilustrado na etapa

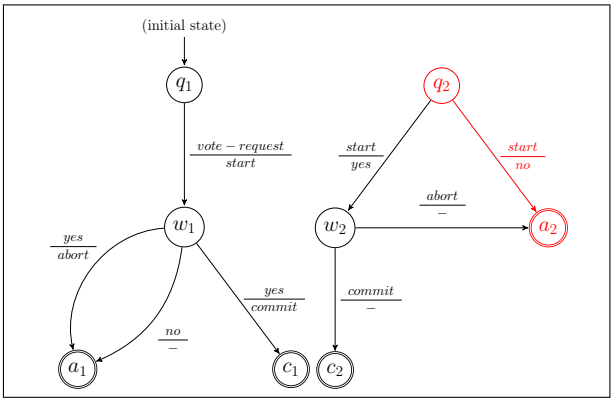
2 da Figura 5. Por outro lado, quando existem mais de dois processos, o coordenador é responsável por propagar sua decisão desfavorável para aqueles que votaram "sim". Na etapa 2 da Figura 6, o participante 2 votou "não", enquanto os participantes 3 e 4 votaram "sim". O primeiro, ao decidir "não", aborta unilateralmente. Já os demais estão bloqueados em razão do seu período de incerteza: não sabem se a transação será confirmada ou fracassada. Na etapa 3, o coordenador propaga sua decisão para aqueles que votaram "sim", aborta e encerra. Os participantes 3 e 4, por sua vez, recebem a decisão de "*abort*", abortam e encerram, como indicado pelas transições $\delta(w_3,\ a_3)$ e $\delta(w_4,\ a_4)$, respectivamente. Esse comportamento se mantém para execuções do protocolo com mais de dois processos.

Figura 5 – Representação do algoritmo *Two-Phase Commit* com dois processos em que o participante aborta unilateralmente.

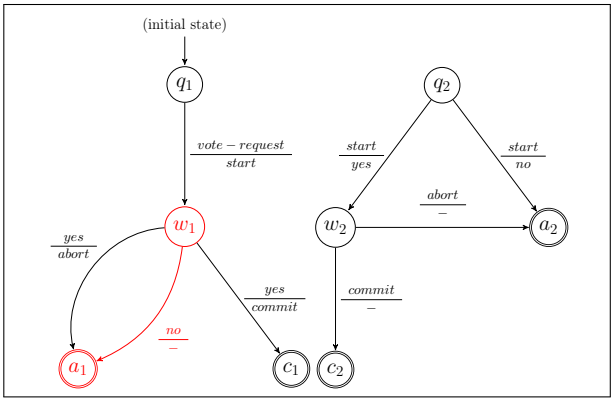
(a) Etapa 1 O coordenador solicita o voto do participante, iniciando o algoritmo.



(b) Etapa 2. Participante votou "não", abortando unilateralmente.



(c) Etapa 3. - O coordenador recebe o voto "não", aborta e encerra.



Adaptado de: SKEEN (1982, figura 3.1, p. 45)

2.2.2.2 Participantes votam "sim", coordenador decide "*abort*"

Embora o protocolo 2PC tenha sido projetado para confirmar uma transação quando todos os participantes votam de maneira favorável, ainda existe a possibilidade, neste cenário, do coordenador decidir abortar. Esse comportamento pode ser observado caso o coordenador experiencie uma falha após coletar todos os votos, mas antes de enviar as mensagens de *commit*; se houver problemas de rede ou *timeouts* que impeçam o coordenador de receber todos os votos "sim" dentro de um período de tempo especificado; podem haver condições ou gatilhos externos que forçam o coordenador a abortar a transação apesar dos votos "sim" unânimes; dentre outros. Note que esse modo de operação é previsto pela Condição 4, que exige que a decisão global seja "*commit*" quando todos os votos são "sim", *desde que não ocorram falhas*.

Esse comportamento pode ser verificado nas Figuras 7 e 8, que representam o protocolo sendo executado com 2 e 4 processos, respectivamente. Estas imagens ilustram como o 2PC se comporta nos casos em que os $i \in \{2, \ldots, N\}$ participantes votaram "sim", representado pela transição $\delta(q_i,\ w_i)$ na etapa 2 de ambas imagens. O processo coordenador, apesar de ter recebido todos os votos "sim", decide "*abort*" (etapa 3, $\delta(w_1,\ a_1)$). Antes de encerrar, o coordenador realiza um *broadcast* com mensagens de "*abort*". Os demais processos participantes, em seguida, encerram seu período de incerteza ao abortar a transação ($\delta(w_i,\ a_i)$).

2.2.2.3 Participantes votam "sim", coordenador decide "*commit*"

Este é o cenário ideal do protocolo 2PC, em que ocorre a confirmação da transação. As Figuras 9 e 10 representam esse cenário no qual o protocolo é executado com 2 e 4 processos. Na etapa 2 da primeira imagem, o processo 2 vota "sim" ($\delta(q_2,\ w_2)$); na mesma etapa da segunda imagem, os processos $i \in \{2, \ldots, 4\}$ votam "sim" ($\delta(q_i,\ w_i)$). O coordenador, ao coletar os votos "sim" unânimes, propaga sua decisão favorável para os demais participantes, realiza o *commit* e encerra − representado pela transição $\delta(w_1,\ c_1)$ na etapa 3 das imagens 9 e 10. O participante 2 na etapa 4 da Figura 9, ao receber a mensagem de "*commit*", o realiza e encerra. Já na Figura 10, na mesma etapa, os participantes $i \in 2, \ldots, 4$, ao receberem a mensagem de "*commit*", executam o *commit* e encerram. Este cenário só é possível quando todos os votos são "sim", conforme a Condição 3.

Com base na análise realizada, pode-se concluir que o 2PC atende às condições 1 à 4. No entanto, o protocolo, conforme descrito até o momento, não cumpre a Condição 5, que exige que uma decisão seja tomada se todas as falhas forem reparadas e não surgirem novas falhas por um período suficientemente longo. A limitação reside no fato de que, em vários pontos do protocolo, os processos devem esperar por mensagens que podem nunca

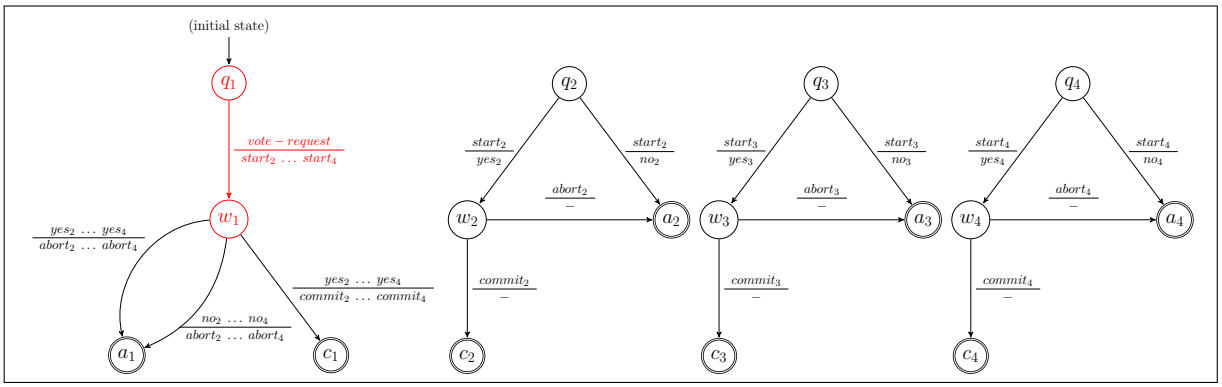
chegar devido a falhas. Para evitar este problema, é necessário incorporar ao modelo um tipo especial de transição, conhecido como ação de *timeout*. Além disso, quando um processo se recupera de uma falha, a Condição 5 exige que este tente chegar a uma decisão consistente com a decisão que os demais podem ter chegado nesse meio tempo. Para que isso seja possível, um processo deve manter algumas informações em armazenamento estável, especificamente no *log* de transação distribuída (BERNSTEIN; HADZILACOS; GOODMAN, 1987).

2.3 RESUMO

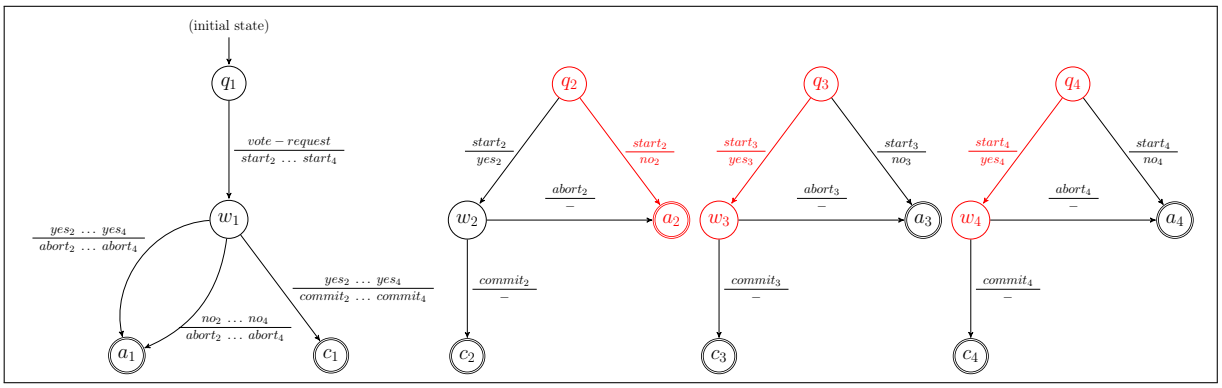
Neste capítulo, o problema do compromisso atômico foi apresentado como um conjunto de condições que um protocolo de *commit* deve satisfazer para garantir a finalização consistente de uma transação distribuída. Tais condições estabelecem que a decisão (de *commit* ou *abort*) seja irreversível e unânime entre os processos. Para isso, foi adotado um modelo computacional que considera a possibilidade de falhas de nós e de comunicação. A partir da formulação do problema, a descrição correta do algoritmo 2PC, considerando um cenário sem falhas de nós e de comunicação, e o estudo de seus cenários de finalização foram apresentados. Verifica-se que, para que o protocolo 2PC satisfaça todas as condições do problema, ele deve implementar ações de *timeout* e especificar quais informações devem ser armazenadas no *log* e como são utilizadas durante o processo de recuperação.

Figura 6 – Representação do algoritmo *Two-Phase Commit* com quatro processos em que pelo menos um participante votou "não".

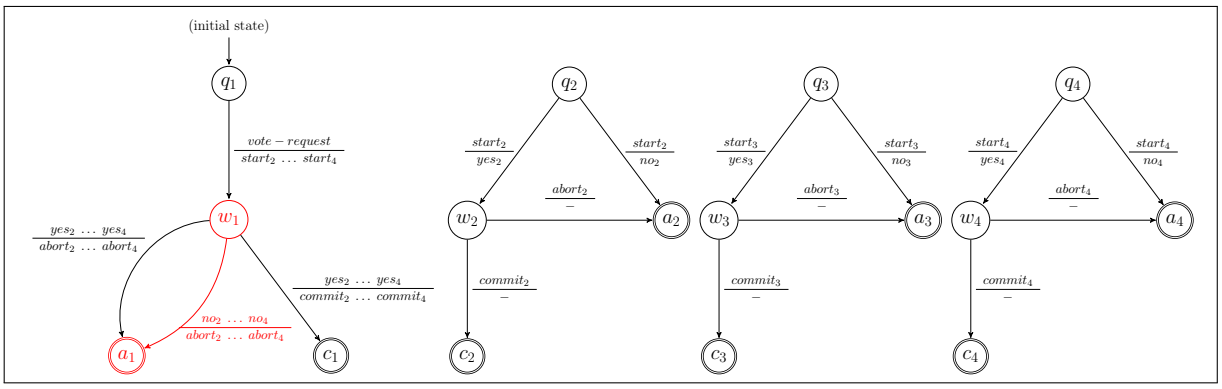
(a) Etapa 1. O coordenador solicita os votos dos participantes, iniciando o algoritmo.



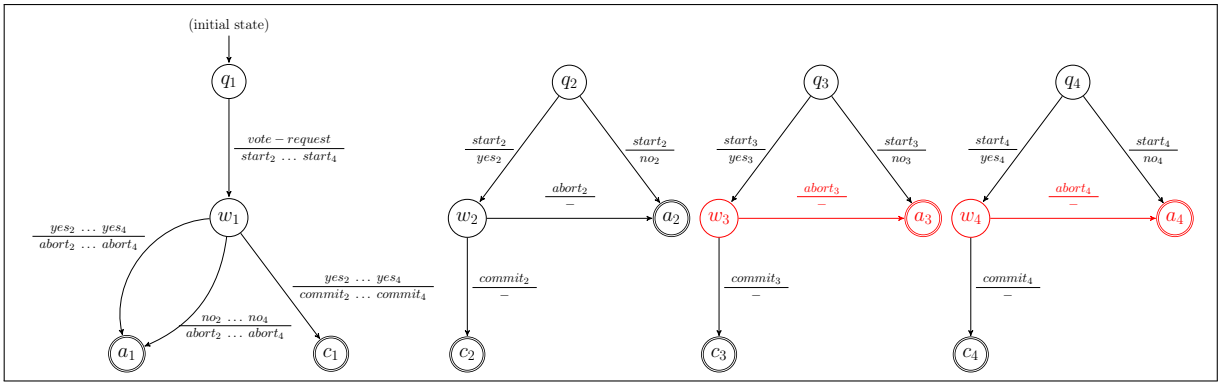
(b) Etapa 2. O participante 2 votou "não", abortando unilateralmente; Participantes 3 e 4 votam "sim", entrando em seu período de incerteza.



(c) Etapa 3. O coordenador recebe o voto "não", envia mensagens de "*abort*", aborta e encerra.



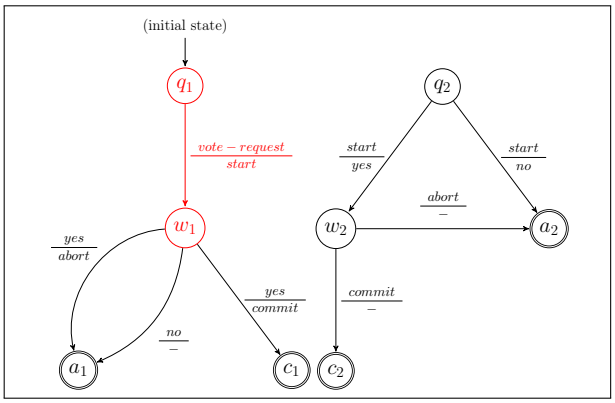
(d) Etapa 4. Os participantes 3 e 4 recebem instrução de "*abort*", o realizam e encerram.



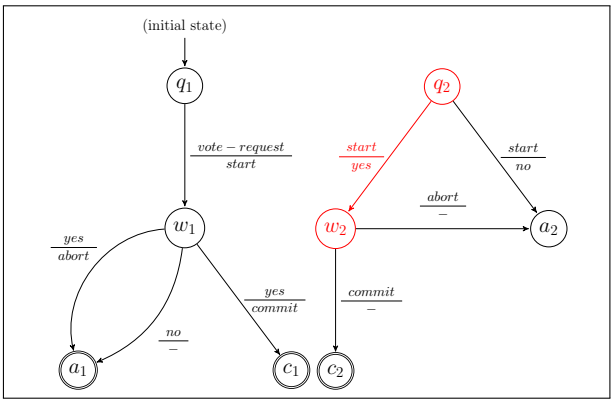
Adaptado de: SKEEN (1982, figura 3.1, p. 45)

Figura 7 – Representação do algoritmo *Two-Phase Commit* com dois processos, em que o participante vota "sim", mas o coordenador decide abortar.

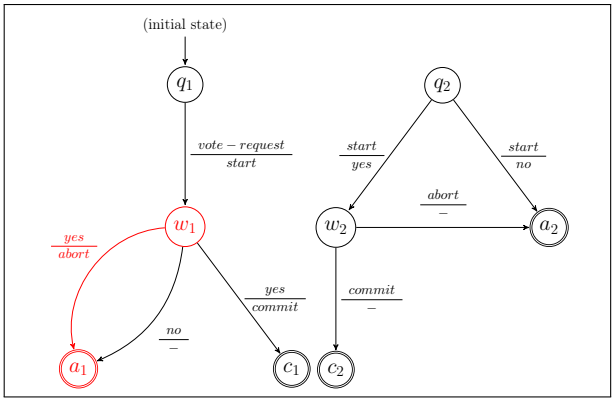
(a) Etapa 1 O coordenador solicita o voto do participante, iniciando o algoritmo.



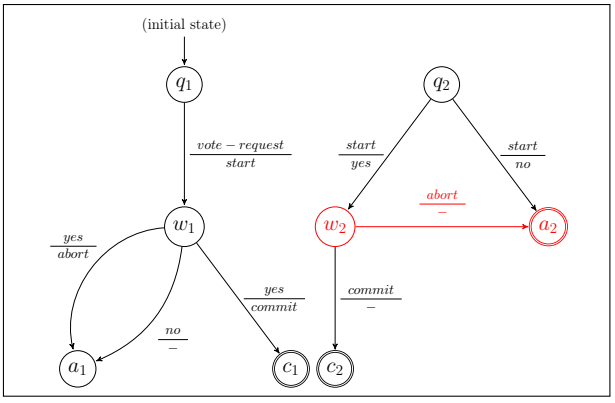
(b) Etapa 2. O participante votou "sim", entrando na sua janela de incerteza.



(c) Etapa 3. - O coordenador recebe o voto "sim", mas decide abortar. Propaga sua decisão e encerra.



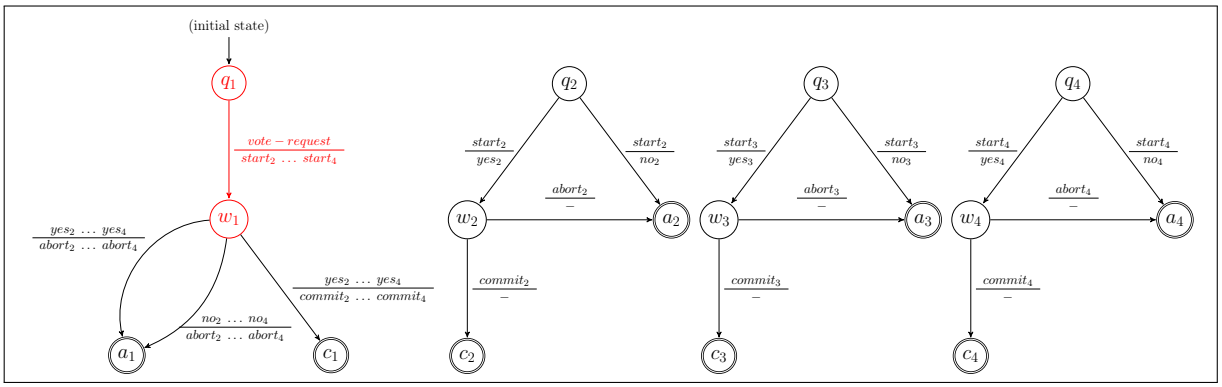
(d) Etapa 4. - O participante recebe instrução de *abort*, o realiza e encerra.



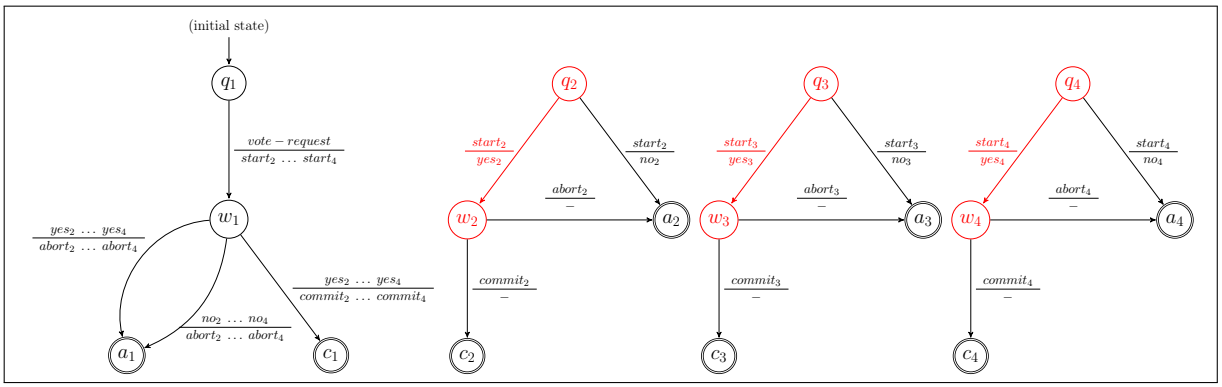
Adaptado de: SKEEN (1982, figura 3.1, p. 45)

Figura 8 – Representação do algoritmo *Two-Phase Commit* com quatro processos em que os $i \in \{2, \ldots, 4\}$ participantes votam "sim", mas coordenador decide abortar.

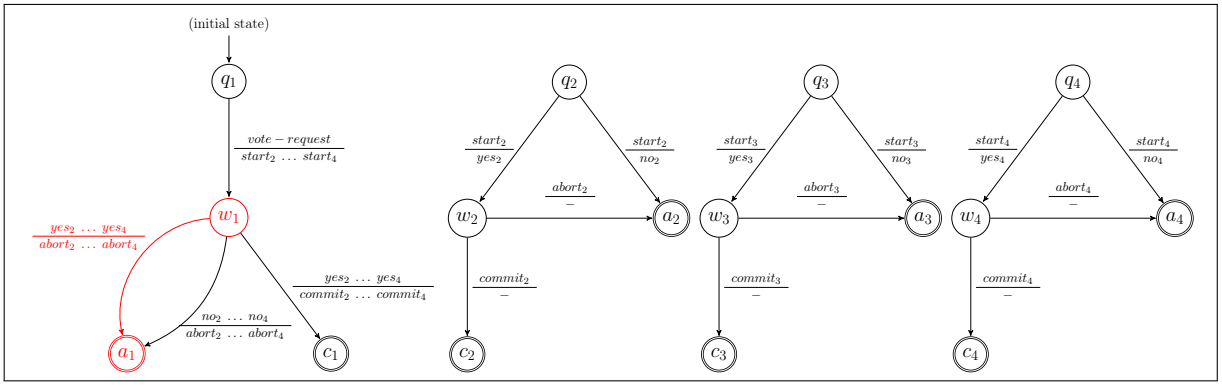
(a) Etapa 1. O coordenador solicita os votos dos participantes, iniciando o algoritmo.



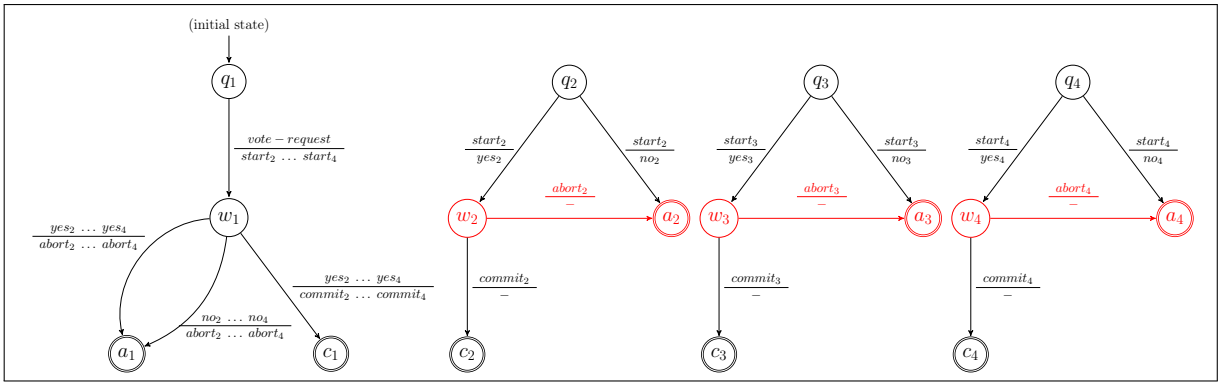
(b) Etapa 2. Os participantes votam "sim", entrando em seu período de incerteza.



(c) Etapa 3. O coordenador recebe os votos "sim", mas decide abortar. Envia mensagens de "*abort*" para aqueles participantes que votaram "sim", aborta e encerra.



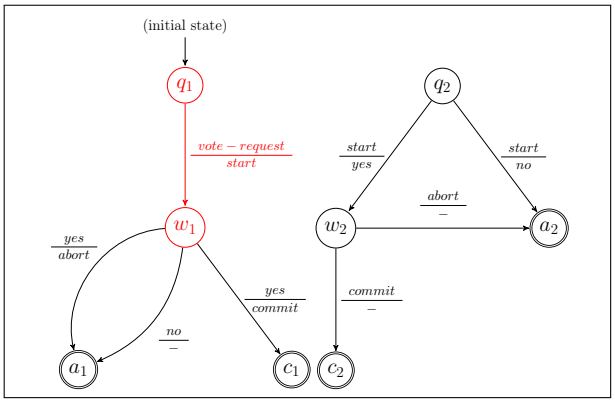
(d) Etapa 4. Os participantes recebem instrução de "*abort*", o realizam e encerram.



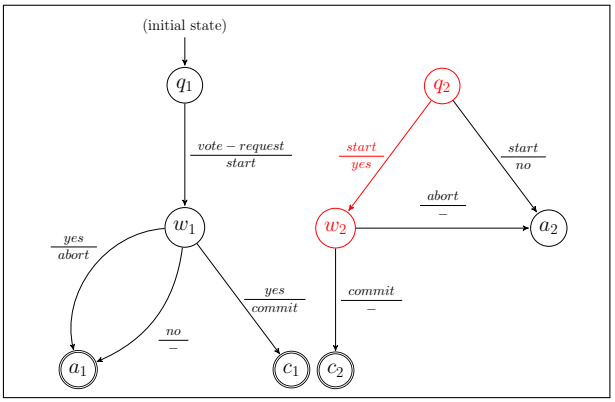
Adaptado de: SKEEN (1982, figura 3.1, p. 45)

Figura 9 – Representação do algoritmo *Two-Phase Commit* com dois processos e decisão final de *commit*.

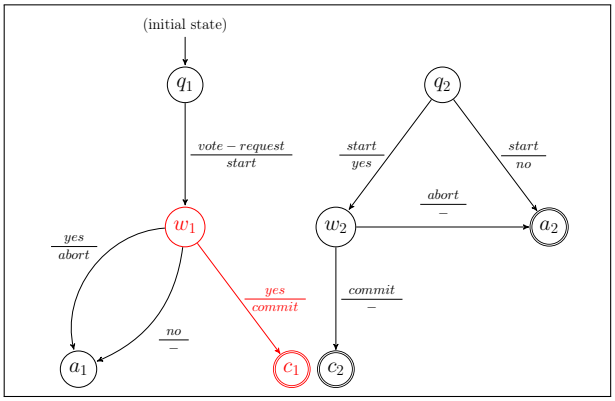
(a) Etapa 1 O coordenador solicita o voto do participante, iniciando o algoritmo.



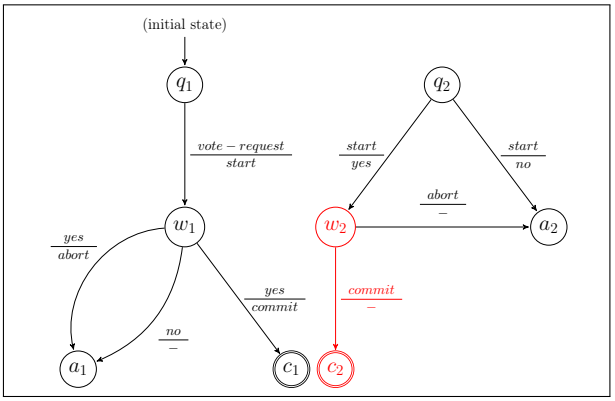
(b) Etapa 2. O participante votou "sim", entrando na sua janela de incerteza.



(c) Etapa 3. - O coordenador recebe o voto "sim" e decide *commit*.



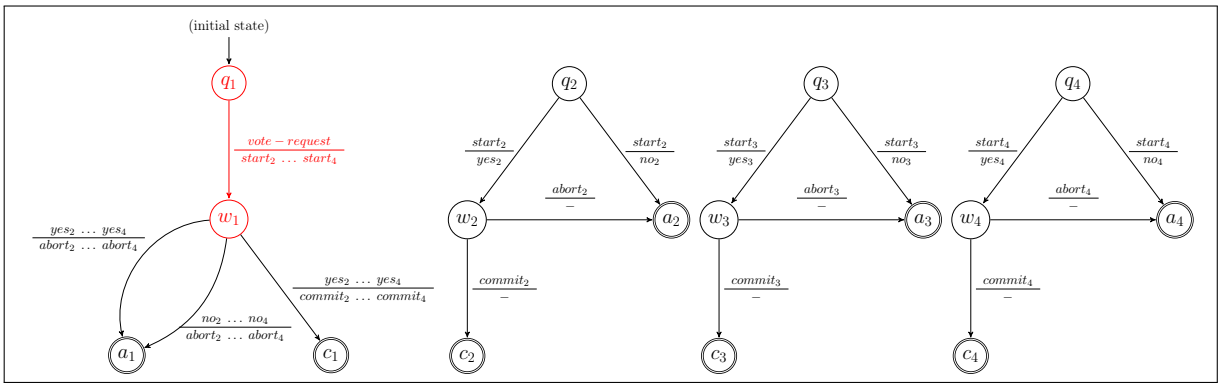
(d) Etapa 4. - Participante recebe a ordem de *commit*, o realiza e encerra.



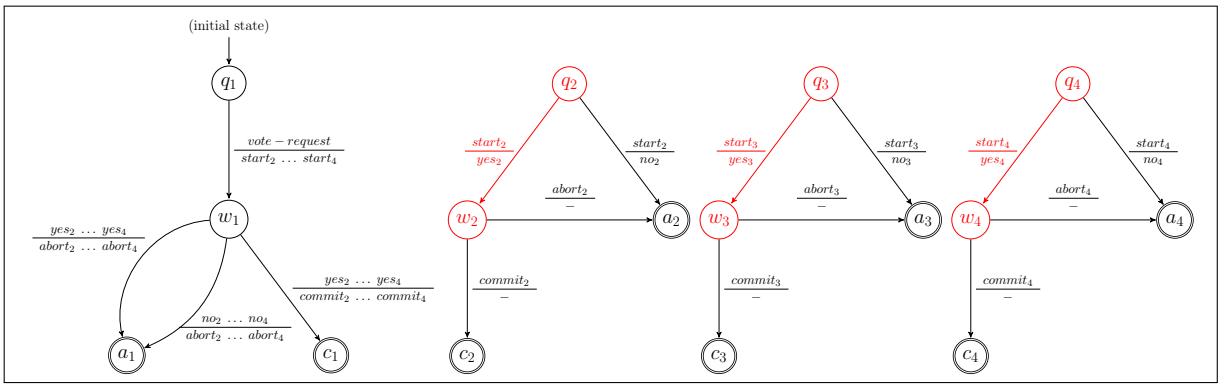
Adaptado de: SKEEN (1982, figura 3.1, p. 45)

Figura 10 – Representação do algoritmo *Two-Phase Commit* com quatro processos e decisão final de *commit*.

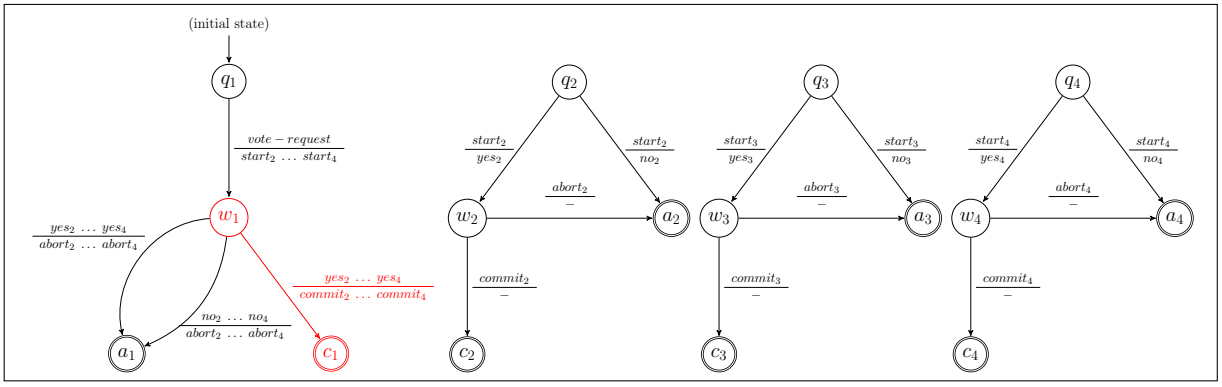
(a) Etapa 1. O coordenador solicita os votos dos participantes, iniciando o algoritmo.



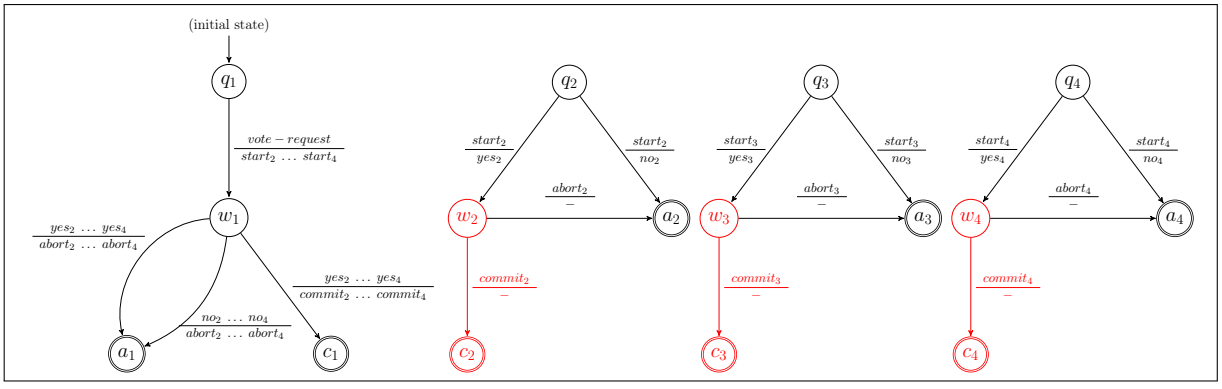
(b) Etapa 2. Os participantes votam "sim", entrando em seu período de incerteza.



(c) Etapa 3. O coordenador recebe os votos "sim", envia mensagens de "*commit*", confirma a transação e encerra.



(d) Etapa 4. Os participantes recebem a instrução de "*commit*", o realizam e encerram.



Adaptado de: SKEEN (1982, figura 3.1, p. 45)

3 IMPACTO DAS FALHAS DE NÓS E DE COMUNICAÇÃO EM PROTOCOLOS DE *COMMIT* ATÔMICO CENTRALIZADOS

Sistemas distribuídos são compostos por dois componentes: nós, que processam informações, e canais de comunicação, que transmitem informações entre nós (BERNSTEIN; HADZILACOS; GOODMAN, 1987). Ambos componentes podem sofrer falhas, classificadas em falhas de nós e de comunicação, respectivamente. Neste capítulo, tais classes de falhas são analisadas, identificando os problemas que cada classe introduz e as condições para que um protocolo seja considerado resiliente. As falhas de nós são classificadas em falhas de um único nó e falhas concorrentes de múltiplos nós, enquanto as falhas de comunicação podem ser causadas por uma partição simples ou múltiplas partições. O protocolo *Two-Phase Commit* é utilizado como estudo de caso, avaliando-se soluções viáveis e as propriedades necessárias e suficientes para garantir a consistência diante de diferentes classes de falhas.

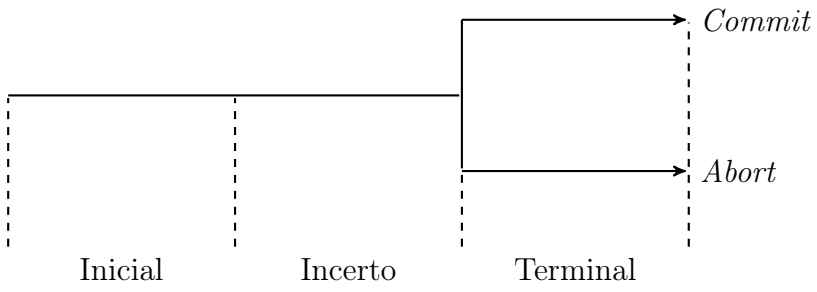
3.1 EFEITO DAS FALHAS NO PROCESSAMENTO DE UMA TRANSAÇÃO DISTRIBUÍDA

Existem dois conjuntos diferentes de problemas causados pela ocorrência de falhas: aqueles enfrentados por processos falhos ao tentarem se recuperar; e aqueles que os processos operacionais enfrentam ao serem impactados por processos falhos (SKEEN, 1982). O modelo de falhas utilizado (conforme definido na Seção 2.1) considera falhas de nós, que causam a inatividade dos processos falhos durante seu período de recuperação; e falhas de comunicação causadas pelo particionamento da rede em grupos disjuntos, o que impossibilita a comunicação entre processos em diferentes grupos. As falhas são detectadas por meio de *timeouts*. Considera-se que um processo detecta que falhou no exato instante em que a falha ocorre. Já os processos operacionais não são capazes de discernir a natureza específica da falha ocorrida, percebendo-a apenas pela ausência do recebimento de uma mensagem esperada, sinalizada por meio de *timeouts*.

Os intervalos definidos pela Figura 11 caracterizam o processamento de uma subtransação por qualquer processo participante que executa um protocolo de *commit* atômico. Note que o participante sempre possui um período em que é incerto, pois não é possível projetar protocolos de *commit* sem períodos de incerteza (BERNSTEIN; HADZILACOS; GOODMAN, 1987). Tratando-se do *Two-Phase Commit*, o intervalo inicial corresponde à fase de votação. Seu fim é marcado pela decisão de *commit* local e pelo início de uma nova fase, a fase de decisão.

Uma falha durante o intervalo inicial faz com que a subtransação seja unilateralmente abortada, causando o fracasso da transação. Diz-se que um processo é incerto durante o intervalo em que votou “sim”, até receber a mensagem com a decisão global, período

Figura 11 – O processamento de uma subtransação em um determinado processo participante.



Fonte: SKEEN (1982, figura 2.3, p. 19)

conhecido como *janela de incerteza*. Se uma falha acontecer durante esse período, o processo deve comunicar-se com os demais seguindo as etapas definidas por um protocolo de terminação. Por fim, o processo que está no intervalo terminal já possui conhecimento da decisão global e a executa. Esse intervalo também é conhecido como *período de escrita*, pois é apenas durante este que operações de escrita sobre os dados são de fato executadas. Se uma falha ocorrer durante o período de escrita, a decisão já foi tomada. O processo pode, após recuperar-se, retomar a execução da operação de *commit* ou *abort*. Conclui-se, então, que a recuperação de falhas nos intervalos inicial e terminal não exige comunicação entre os processos, caracterizando a *recuperação independente*. Já uma falha enquanto houver processos incertos não pode ser resolvida utilizando recuperação independente e pode levar a situações de bloqueio.

Protocolos que garantem a resiliência para uma determinada classe de falhas são aqueles que não permitem que processos operacionais sejam bloqueados em razão de falhas, preservando o término consistente da transação (HADZILACOS, 1990) define a Condição Não Bloqueante.

Condição NB (Não bloqueante)**.** *Todos os processos corretos, aqueles que nunca falharam, chegam a uma decisão.*

Os autores BERNSTEIN; HADZILACOS; GOODMAN trazem uma visão complementar para a definição da Condição NB, afirmando que, "caso existam processos operacionais incertos, então nenhum processo (operacional ou em recuperação) pode decidir pelo commit". Tal afirmação é válida tanto para processos participantes quanto para o processo coordenador (veja Cenário 1). Os protocolos não bloqueantes são aqueles que satisfazem a Condição Não Bloqueante. A noção forte de não bloqueio trata de protocolos que, para todas as execuções possíveis, incluindo todas as combinações de falhas, terminam a transação de maneira consistente nos processos corretos em um número finito de mensagens, e não requer que nenhuma mensagem (com exceção de mensagens de *timeout*) sejam enviadas por um processo após sua falha. Geralmente, tem-se interesse em uma noção

mais fraca de não bloqueio em que o protocolo não é obrigado a terminar a transação, mas apenas deixá-la em um estado do qual pode ser terminada em um número finito de mensagens SKEEN. Os protocolos de *commit* não bloqueantes discutidos neste trabalho satisfazem a noção mais fraca de não bloqueio.

A seguir, exploram-se diversos cenários de execução do protocolo 2PC, com o objetivo de construir uma intuição a respeito de seu comportamento diante de falhas de nós e de comunicação. As seções seguintes formalizam esse estudo, destacando as condições necessárias e suficientes para garantir resiliência a tais classes de falhas.

3.2 UMA ANÁLISE DO PROTOCOLO *TWO-PHASE COMMIT* NA OCORRÊNCIA DE FALHAS

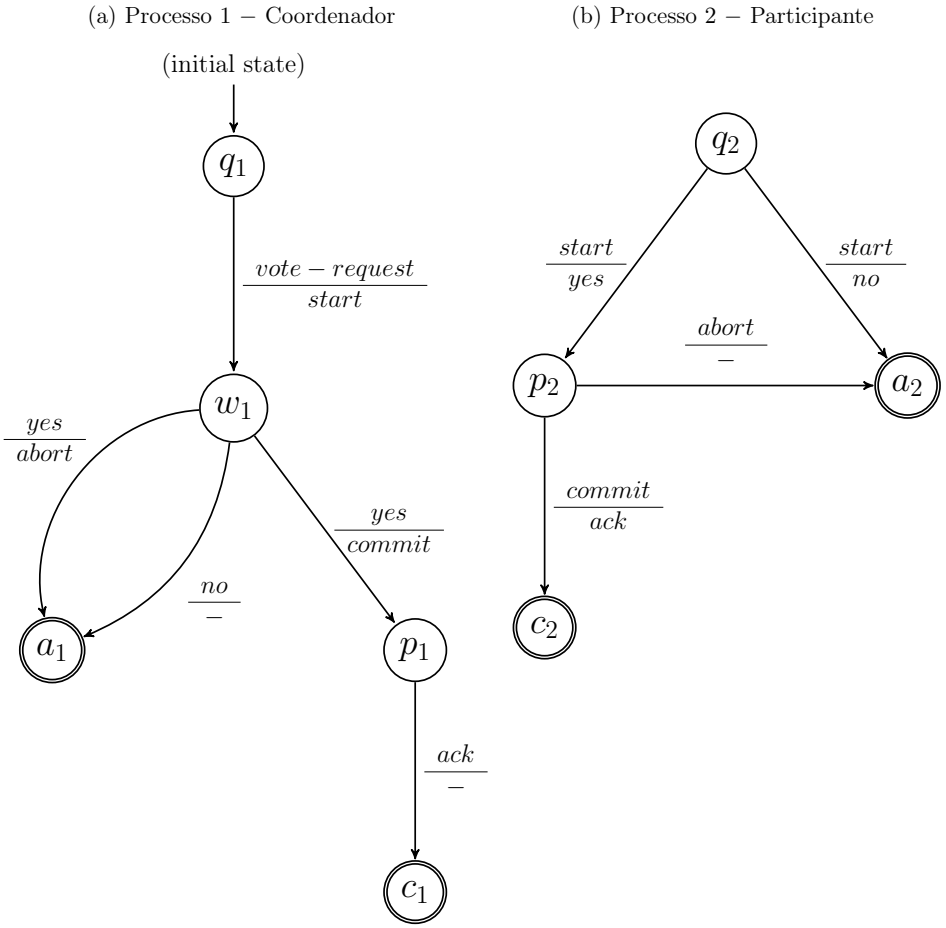
O protocolo 2PC é vulnerável, sobretudo, a falhas do coordenador e partições de rede, pois isolam o processo coordenador (SKEEN, 1982). Para compreender essa afirmação, consideram-se diversos cenários de execução do protocolo *Two-Phase Commit* centralizado.

Cenário 1 (Falha de um único nó). Considera-se o protocolo 2PC em execução com três processos, em que os dois processos participantes votam "sim", tornando-se incertos. O processo coordenador coleta os votos favoráveis e decide confirmar a transação. Registra sua decisão em seu *log* de transação distribuída (TD), porém falha após enviar a primeira mensagem de "*commit*".

A ocorrência de falhas demandam que ações sejam tomadas, modeladas através de transições especiais. Sabendo disso, considere que a recuperação independente foi utilizada como estratégia para lidar com a falha do processo coordenador descrita pelo Cenário 1. O processo coordenador, ao recuperar-se, verifica que existe um registro de "*commit*" no seu *log* TD. Por causa disso, realiza uma *transição de falha* para o estado de *commit*. O Processo 2 recebeu a instrução de "*commit*", o realiza e encerra. Porém, o nó 3 não recebeu a mensagem de decisão. Quando seu período de *timeout* se encerrar, ele realizará uma *transição de timeout* para um estado de *abort*, causando o término inconsistente da transação. Esse comportamento é resultado do envio de mensagens de "*commit*" para processos incertos, violando a condição não bloqueante, isto é, podendo causar situações de bloqueio. Em caso de falha do coordenador, um processo incerto não tem conhecimento do estado da transação nos demais processos participantes. Ele não sabe dizer se todos os demais votaram "sim" ou se pelo menos um processo votou "não" e abortou unilateralmente, não podendo recuperar-se de forma independente sem, potencialmente, violar as condições do problema. Isso demonstra que o protocolo 2PC, conforme definido até então, não é resiliente a falhas de um único nó. Além disso, o Cenário 1 ilustra que não é possível designar transições de falha e de *timeout* sob recuperação independente neste protocolo, pois pode causar o término inconsistente da transação.

Sabendo disso, o protocolo 2PC estendido com mensagens de reconhecimento (Figura 12) apresenta uma diferença crucial em comparação com o protocolo 2PC padrão. Enquanto no último, o coordenador envia mensagens de "*commit*" para processos incertos e avança imediatamente para o estado de *commit*, no protocolo estendido, o coordenador aguarda o reconhecimento da decisão de "*commit*" por parte dos participantes antes de transicionar para o estado final de *commit*. A Figura 12 representa este protocolo com dois processos, introduzindo um estado intermediário entre w_1 e c_1 como uma "preparação para o *commit*", ilustrando que o coordenador, após enviar a mensagem de "*commit*", aguarda o reconhecimento ("*ack*") desta por parte do participante antes de transicionar para o estado de *commit*. Isso permite que o coordenador tenha certeza de que todos os participantes incertos foram informados e, assim, evita o risco de bloqueio, permitindo a recuperação independente em caso de falhas.

Figura 12 – Protocolos locais do algoritmo *Two-Phase Commit* centralizado estendido com mensagens de reconhecimento com $N = 2$.



Fonte: SKEEN (1982, figura 4.1, p. 70)

Quando o coordenador falha na segunda fase do protocolo, frequentemente os processos participantes não conseguem prosseguir com a transação de forma segura (SKEEN,

1982). Isso é especialmente verdade se um participante falhar em conjunto com a falha do coordenador. Nesse contexto, considere o Cenário 2.

Cenário 2 (Falhas de múltiplos nós). Considera-se o protocolo 2PC em execução com quatro processos, em que os três processos participantes votam "sim", tornando-se incertos. O processo coordenador falhou durante o envio das mensagens de "*commit*", de forma que somente o Processo 2 a recebeu. Antes de registrar a decisão de "*commit*" em seu *log* TD, o Processo 2 falha, tornando-se indisponível.

Diante da falha do coordenador, visto que existem processos incertos, os nós que permanecem operacionais devem comunicar-se para verificar se algum deles possui conhecimento da decisão global. Essa comunicação ocorre por meio de um protocolo de terminação, que se baseia na interpretação dos registros do *log* TD para evitar situações de bloqueio. A escrita no *log* TD segue o método *Write-Ahead Logging* (WAL), que exige o registro das operações de escrita antes de sua execução. No Cenário 2, se o processo 2 estivesse disponível, os demais participantes acessariam a decisão global por meio do protocolo de terminação. Entretanto, como 2 falha, os processos 3 e 4 permanecem bloqueados até que o coordenador ou o participante 2 se recuperem. Esse cenário demonstra a inexistência de soluções não bloqueantes para falhas concorrentes de nós sob recuperação independente no protocolo 2PC.

As consequências de falhas de comunicação são ilustradas pelos Cenários 3 e 4.

Cenário 3 (Partição simples). Considera-se o protocolo 2PC em execução com dois processos, em que o único processo participante vota "sim", tornando-se incerto. Antes que o coordenador pudesse enviar sua mensagem de decisão, uma falha na rede causa a separação dos nós coordenador e participante em dois grupos disjuntos. Ambos os nós estão operacionais, mas não conseguem se comunicar.

No Cenário 3, a única forma de existir um protocolo capaz de ser resiliente a partições simples é a adoção de uma abordagem em que a ocorrência de uma partição de rede seja detectada por um *timeout*, permitindo que os processos realizem transições apropriadas. Nesse modelo, a partição é representada como uma transição global de estado que apaga todas as mensagens pendentes. Cada processo envolvido na partição detecta a falha por meio de um *timeout* e realiza uma transição correspondente. Essa abordagem cria uma analogia entre partições de rede e falhas simultâneas de nós, uma vez que, em ambos os casos, a ausência de mensagens leva à ativação de um *timeout* e à adoção de medidas de recuperação. A resiliência a partições simples, portanto, depende da capacidade da rede de permitir que os processos detectem a perda de mensagens por meio de transições de *timeout*, condição necessária para que essas medidas possam ser aplicadas com segurança.

Cenário 4 (Múltiplas partições). Considera-se o protocolo 2PC em execução com quatro processos, em que os três processos participantes votam "sim", tornando-se incertos.

O coordenador, então, envia a mensagem de "*commit*" para o Processo 2, mas falha antes de comunicar os demais participantes, pois a rede foi particionada em três grupos distintos: o coordenador e o nó 4 isolados, e os nós 2 e 3 agrupados.

Diante do Cenário 4, os processos 2 e 3 podem interagir entre si, e como o Processo 2 recebeu a decisão do coordenador, ele pode retransmiti-la ao nó 3, permitindo que ambos concluam a transação de maneira consistente com a decisão global. No entanto, o nó 4, estando completamente isolado, não pode comunicar-se com nenhum outro processo e permanecerá bloqueado até que a rede seja restaurada. Esse contexto ilustra a inexistência de protocolos resilientes a múltiplas partições (Teorema 3), pois essas podem impedir que processos incertos tenham conhecimento da decisão global, inviabilizando a terminação segura da transação até que a comunicação seja restabelecida.

3.3 ESTADO GLOBAL DA TRANSAÇÃO

No Cenário 2, verifica-se que a recuperação independente não pode ser utilizada quando falhas concorrentes ocorrem. Isso decorre do fato de um participante estar ciente somente do próprio voto. Esse comportamento traz reflexão a respeito dos estados (locais) observados nos protocolos locais em perspectiva com o *estado global da transação*, motivando o assunto desta seção (SKEEN, 1982).

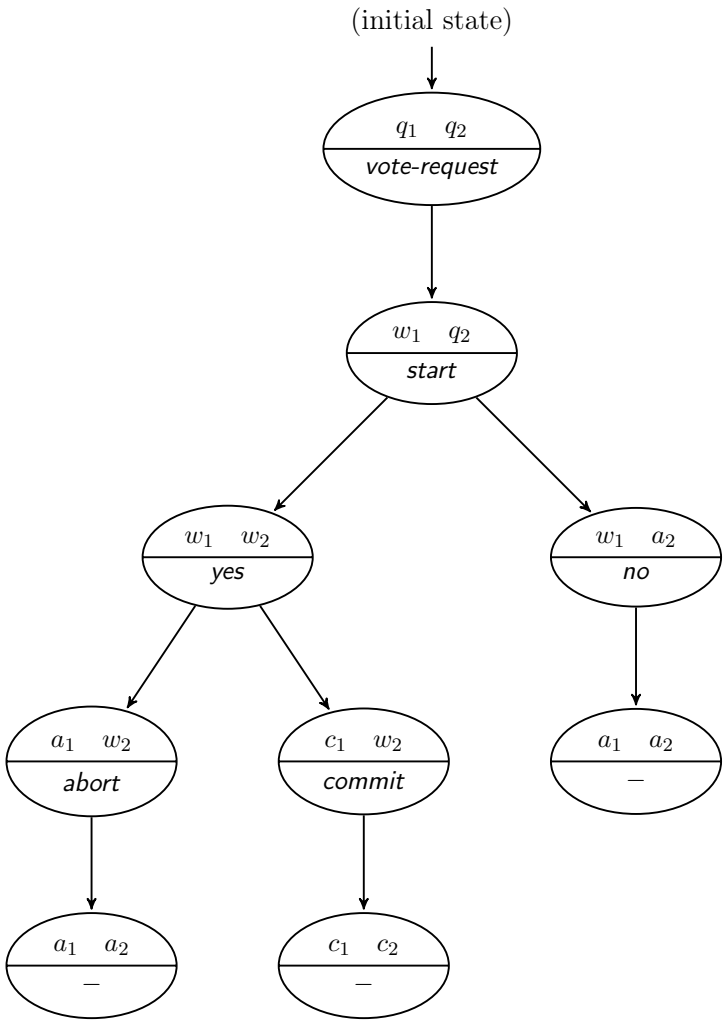
O estado global de uma transação distribuída descreve por completo o processamento da transação, e é definido a seguir.

1. Um vetor global de estados contendo os estados (locais) dos protocolos locais;
2. As mensagens pendentes na rede.

Uma transição de estado global ocorre sempre que uma transição de estado local ocorre. Se existe uma transição global do estado global g para o estado global g', então g' é dito *imediatamente alcançável* a partir de g (SKEEN, 1982). Apesar da facilidade para geração de tal grafo, sua complexidade cresce exponencialmente na medida que N cresce. Felizmente, uma quantidade pequena de nós já é suficiente para ilustrar o protocolo para fins didáticos. Será considerado o protocolo 2PC com dois processos.

Cada nó no grafo da Figura 13 representa um estado global. Acima da linha horizontal encontra-se o vetor global de estados, que reúne os estados locais dos protocolos executados por cada nó envolvido na transação em dado instante de execução da transação; e abaixo, o estado da rede, composto pelas mensagens pendentes necessárias para realizar a transição de estado global. Cada nó desse grafo pode ser interpretado como uma "foto" da transação, retratando quais estados locais podem ser ocupados de maneira concorrente em cada etapa do algoritmo. Isso significa que, se esse vetor contiver ambos estados de *commit* e *abort*,

Figura 13 – Grafo de estados globais alcançáveis do protocolo *Two-Phase Commit* com $N = 2$.



Fonte: SKEEN (1982, figura 3.4, p. 55)

este é dito *inconsistente* − pois isso significaria haver dois processos em estados (locais) finais divergentes (violando a Condição 1).

A Figura 14 contribui para o entendimento do grafo de estados globais alcançáveis. Nela, fica claro que este grafo busca representar todas as possíveis combinações de estados globais em uma execução sem a ocorrência de falhas, destacando os estados locais que podem ser ocupados ao mesmo tempo. Considera-se que somente uma transição local pode ocorrer por vez. Consequentemente, em cada transição global, apenas um processo altera seu estado local. O primeiro estado global representado na Figura 13 é o estado inicial do protocolo, em que ambos processos ocupam o estado inicial, q_1 e q_2, correspondendo ao coordenador e participante, respectivamente. Note que os estados identificados com o subscrito 1 correspondem ao coordenador, enquanto aqueles com 2 correspondem ao participante. A primeira transição ocorre quando o coordenador inicia o protocolo e

envia uma solicitação de voto ao participante. Este evento é representado pela transição de estado $\delta(q_1,\ w_1)$, onde o coordenador avança para o estado de espera, indicando que aguarda a resposta do participante.

O participante, por sua vez, pode responder ao coordenador com um voto positivo ou negativo, resultando em duas possíveis transições. Se vota "não", as transições $\delta(q_2,\ a_2)$ e $\delta(w_1,\ a_1)$ ocorrem nessa ordem, correspondendo, respectivamente, a um estado global em que os estados $(w_1,\ a_2)$ e $(a_1,\ a_2)$ são ocupados pelos processos no mesmo instante. Esse é o cenário em que o participante vota "não" e o coordenador decide "*abort*" (Figura 14 (a)). Se o participante vota "sim", a transição $\delta(q_2,\ w_2)$ é executada, deixando os processos coordenador e participante nos estados de espera w_1 e w_2. A partir desse instante, o coordenador possui duas opções: confirmar ou abortar a transação. Se decide abortar, as transições $\delta(w_1,\ a_1)$ e $\delta(w_2,\ a_2)$ são executadas. Esse cenário reflete a situação em que o participante vota "sim", mas o coordenador decide abortar a transação (Figura 14 (b)). Por outro lado, se o coordenador decide pelo *commit* após o voto positivo do participante, as transições $\delta(w_1,\ c_1)$ e $\delta(w_2,\ c_2)$ são executadas, resultando no estado global $(c_1,\ c_2)$, em que a transação é confirmada (Figura 14 (c)).

Um estado global é *final*, marcando o fim do processamento da transação, se todos os estados locais do seu vetor de estados possuir somente estados finais (c_i ou a_i). Um *estado terminal*, por sua vez, é aquele estado global que não possui sucessores imediatamente alcançáveis. Um caminho a partir do estado inicial para um estado terminal representa uma possível execução do protocolo. Um estado terminal que não é um estado final é um estado de *deadlock*, uma vez que não existem estados para transicionar e o estado não é final.

Com base nessas definições, é possível compreender por que o grafo de estados alcançáveis é uma medida forte de corretude: ele não pode admitir estados inconsistentes, nem estados terminais que não sejam finais. Se tais estados forem observados no grafo, isso indica que a transação não está operacionalmente correta.

Outro estado de interesse é o estado *confirmável* (tradução livre). A ocupação deste estado por qualquer nó implica que todos os demais processos participantes do protocolo votaram "sim" para o *commit* da transação. Por outro lado, aquele processo que ocupa um *estado não confirmável* não possui informações sobre o voto dos demais participantes. No protocolo 2PC, o único estado confirmável é o estado de *commit* (c_i); todos os demais, incluindo o estado de *abort*, são não confirmáveis.

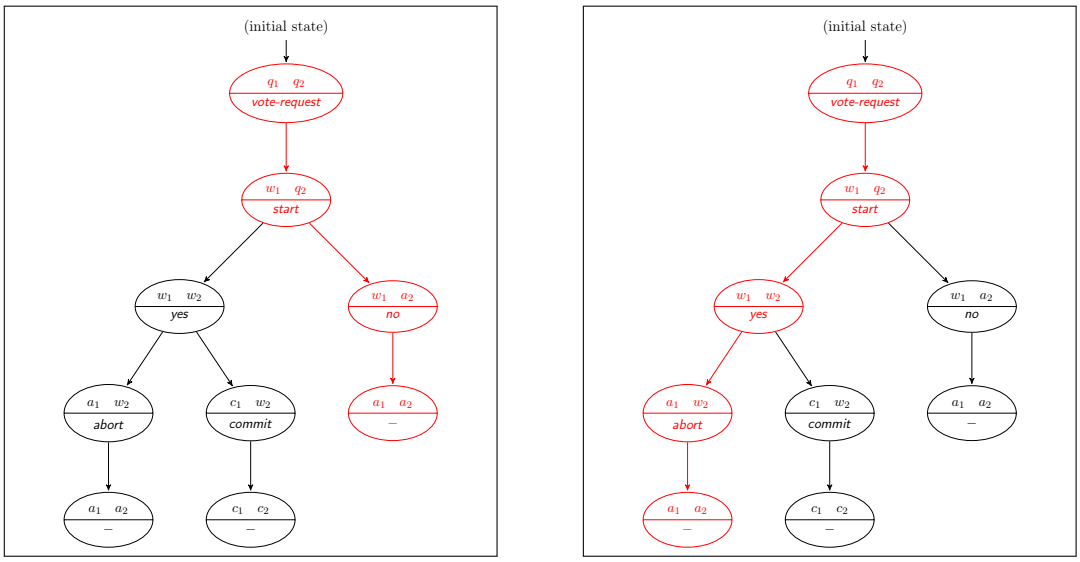
3.4 CONJUNTOS DE CONCORRÊNCIA E DE REMETENTES

Dois conjuntos (SKEEN, 1982) que desempenham um papel importante ao longo deste trabalho são definidos nesta seção. Ambos podem ser construídos a partir do grafo de estados globais. O conjunto de concorrência é fundamental para modelar falhas concor-

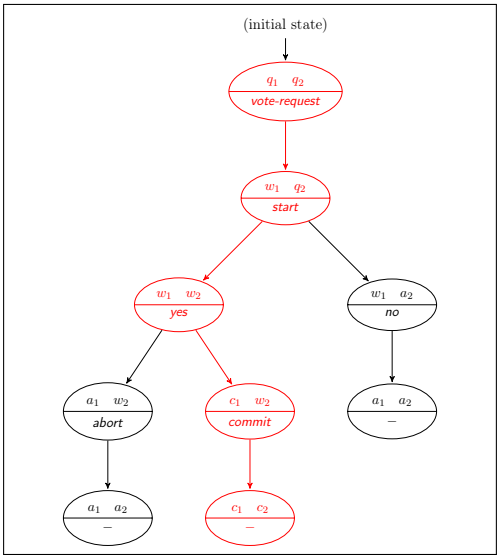
Figura 14 – Cenários de finalização do protocolo 2PC com 2 nós representados através de seu grafo de estados globais alcançáveis.

(a) Cenário em que o participante vota "não" e coordenador decide "*abort*"

(b) Cenário em que o participante vota "sim" e coordenador decide "*abort*"



(c) Cenário em que o participante vota "sim" e coordenador decide "*commit*"



Adaptado de: SKEEN (1982, figura 3.4, p. 55)

rentes de nós, pois identifica o conjunto de estados que pode afetar um estado arbitrário caso ocorra uma falha de múltiplos nós. Já o conjunto de remetentes é essencial para modelar falhas de comunicação, uma vez que aponta os estados responsáveis por enviar mensagens que podem ser perdidas, impactando o avanço da execução do protocolo.

A partir do grafo de estados alcançáveis, é possível perceber que, a cada instante de execução da transação distribuída, diferentes processos ocupam estados locais de forma

concorrente. Por exemplo, no estado global $(w_1,\ w_2)$, o coordenador está no estado w_1 e o participante está no estado w_2, indicando que os dois processos ocupam estes estados locais ao mesmo tempo. Isso significa que w_1 é potencialmente concorrente com w_2. Diz-se "potencialmente" porque o processo 2 pode estar ocupando, além de w_2, o estado a_2. Veja definição formal do conjunto de concorrência.

Definição 1 (Conjunto de Concorrência). Seja e um estado local arbitrário. O conjunto de concorrência de e é o conjunto de estados locais que são potencialmente concorrentes com ele. Este conjunto é denotado por $C(e)$.

Para construir esse conjunto, basta analisar os estados que possuem transições diretas a partir de e, isto é, são imediatamente alcançáveis a partir de e. Isso traduz-se nos estados que compartilham estados globais com e. A ideia por trás dessa definição deriva do fato de que os processos participantes executam a transação de forma concorrente, de modo que, em um dado instante em que um participante ocupa o estado e, os demais ocupam estados pertencentes ao conjunto de concorrência de e. Considere o protocolo 2PC com dois processos, representado pela Figura 3, cujo grafo de estados globais alcançáveis é determinado pela Figura 13. O conjunto de concorrência de w_2, por exemplo, representado por $C(w_2)$, consiste dos estados $\{w_1,\ a_1,\ c_1\}$. Note que $C(w_2)$ é inconsistente, representando a vulnerabilidade do 2PC em decidir *commit* enquanto existem processos incertos: em um cenário de falha, podem haver processos que atingem decisões finais divergentes, conforme representado no Cenário 2. Observe também que os estados locais pertencentes ao conjunto $C(w_2)$ referem-se exclusivamente ao participante 1, pois não é razoável que um estado local seja concorrente a outro estado que pertence ao protocolo local do mesmo processo.

Além do conjunto de concorrência, outro conjunto relevante para a análise do comportamento de protocolos distribuídos é o conjunto de remetentes. Em protocolos distribuídos, como o 2PC, ao se considerar a transição de um processo de um estado local para outro, é conveniente observar as mensagens enviadas e recebidas durante essa transição, pois elas determinam a sequência de eventos no sistema. Formalmente, o conjunto de remetentes de um estado local e é definido da seguinte forma:

Definição 2 (Conjunto de Remetentes). Seja e um estado local arbitrário, e seja M o conjunto de mensagens recebidas por e. O conjunto de remetentes de e, denotado por $S(e)$, é definido por $\{j \mid j \text{ envia } m \text{ e } m \in M\}$.

A ideia por trás da definição deste conjunto é identificar os estados locais que enviam mensagens para um estado arbitrário e. A partir do grafo de estados alcançáveis, o conjunto de remetentes pode ser construído observando as mensagens pendentes na rede em cada transição. Por exemplo, no caso do 2PC, somente o estado w_1 envia mensagens para o estado w_2, de forma que $S(w_2) = \{w_1\}$. Isso pode ser verificado nos estados globais

$(a_1,\ w_2)$ e $(c_1,\ w_2)$, em que existem as mensagens "*abort*" e "*commit*" pendentes na rede, que são mensagens que w_1 pode enviar para w_2.

3.5 FALHAS DE NÓS

Um processo está operando corretamente, ou inativo em razão de uma falha. Em um dado instante da execução de um algoritmo de *commit*, diferentes nós podem estar em diferentes estados, o que pode incluir combinações variadas de nós falhos. Uma *falha parcial* acontece com a inatividade de um ou mais nós; a *falha total* ocorre quando todos os nós param de funcionar. Um protocolo de *commit* é resiliente a falhas de nós se satisfaz a Condição NB (SKEEN, 1982), garantindo que processos corretos nunca sejam bloqueados em razão da indisponibilidade de processos falhos.

As próximas subseções detalham as circunstâncias sob as quais é possível realizar a recuperação independente e, quando não for possível, destaca-se a necessidade de aplicar o protocolo de terminação. Para isso, a análise baseia-se (SKEEN, 1982), que investiga as limitações dos protocolos de *commit*, e é demonstrada no contexto do 2PC. Também será discutido detalhadamente o funcionamento do protocolo de terminação, com ênfase nos pontos do algoritmo dos quais são necessários registros no *log* de transações para assegurar que o processo retome seu estado após uma falha.

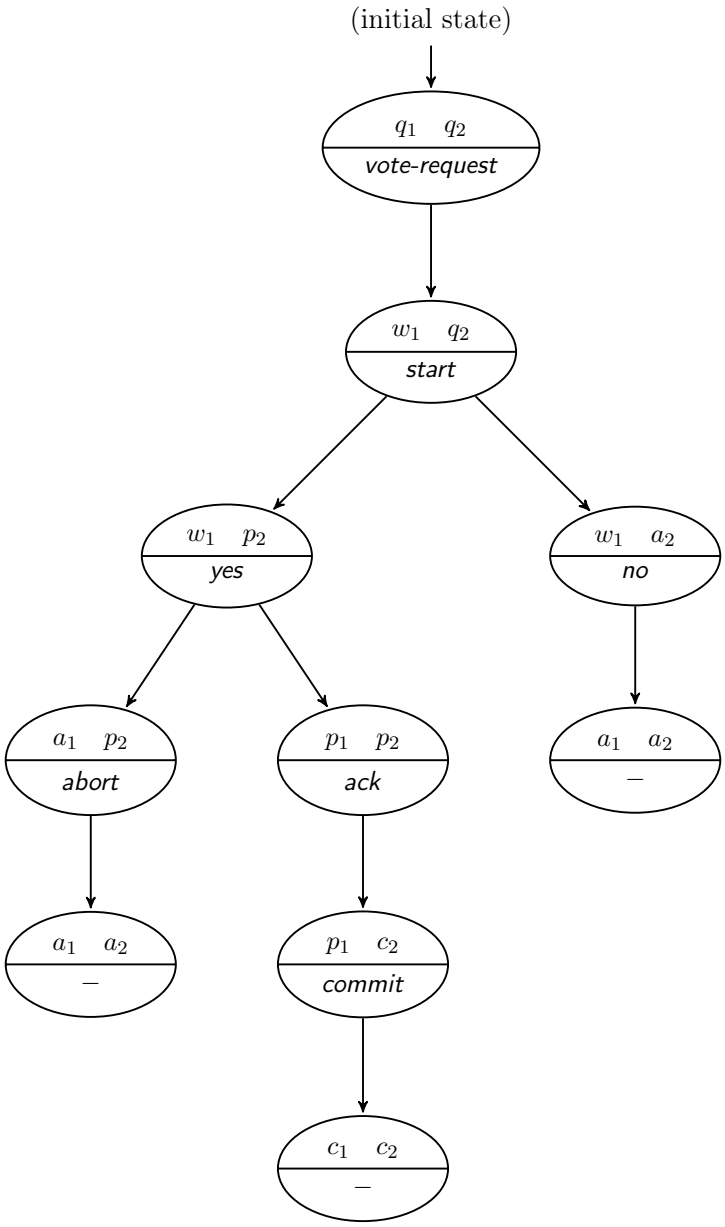
3.5.1 Falha de um único nó

Da mesma forma que um vetor de estados pode ser inconsistente, um conjunto de concorrência possui esta classificação se contiver ambos estados finais de *commit* e *abort*. Nem todo protocolo de *commit* pode ser resiliente a falhas de um único nó, pois, "se um protocolo contiver um estado local do qual o seu conjunto de concorrência é inconsistente, então, sob recuperação independente, não é resiliente a falha de um único nó" (Lema 4.1, SKEEN, 1982). Isso ocorre porque um processo p com um conjunto de concorrência inconsistente não pode fazer com segurança uma transição de falha para um estado de *commit* porque outro processo pode ter abortado a transação; da mesma forma, não pode haver uma transição de falha para um estado de *abort*, uma vez que pode haver pelo menos um processo que confirmou a transação. Portanto, p não pode realizar uma recuperação independente de forma segura.

O estado w_2, com seu conjunto de concorrência $C(w_2) = \{w_1,\ a_1,\ c_1\}$, viola esse lema, indicando que o protocolo *Two-Phase Commit* não é resiliente a esse tipo de falha. Como consequência, o coordenador avança para o estado de *commit* enquanto os processos estão incertos. O protocolo representado pela Figura 12, que ilustra o protocolo 2PC estendido com mensagens de reconhecimento, pode tornar-se resiliente a falhas de um único nó pois envia mensagens de "*commit*" para processos confirmáveis. Seu grafo de estados alcançáveis é representado pela Figura 15. Note que, a partir do estado global $(w_1,\ p_2)$,

não há mais transições diretas para ambos estados finais como ocorre no protocolo 2PC. O estado p_i é um "estado de *buffer*", funcionando como um estado intermediário entre a decisão de *commit* e a execução da operação de fato, fazendo alusão ao conceito de *buffer* em computação.

Figura 15 – Grafo de estados globais alcançáveis do protocolo *Two-Phase Commit* estendido com mensagens de reconhecimento com $N = 2$.



Fonte: SKEEN (1982, figura 3.4, p. 55)

Uma vez que o protocolo 2PC estendido com mensagens de reconhecimento pode ser resiliente a falhas únicas, é possível associar transições de falha e *timeout* de acordo com as regras a seguir (SKEEN, 1982), que correspondem às transições mencionadas nesta sequência.

Regra 1. Para cada estado intermediário e: Se $C(e)$ contém um estado de *commit*, logo associe uma transição de falha de e para um estado de *commit*; caso contrário, associe uma transição de falha para um estado de *abort*.

Uma transição de falha para um estado de *commit* pode ser considerada contra intuitiva, mas trata-se da recuperação de um processo que já havia decidido confirmar a transação, ou havia recebido a ordem para tal.

Regra 2. Para cada estado intermediário e_i: Se e'_j pertence a $S(e_i)$, e e'_j possui uma transição de falha para um estado de *commit* (*abort*), então associe uma transição de *timeout* de e_i para um estado de *commit* (*abort*).

Para compreender a Regra 2, imagine um cenário em que um estado $e'_j \in S(e_i)$ falha e realiza uma recuperação independente para um estado final. Neste caso, e_i não irá receber as mensagens necessárias para realizar determinada transição que depende do estado e'_j, resultando em uma espera infinita. Para que isso não aconteça, as transições de *timeout* são associadas de acordo com esta regra.

Tais regras sempre produzem protocolos resilientes a falhas únicas sob recuperação independente (Teorema 4.2, SKEEN, 1982). Na Figura 16, são designadas as transações de falha e *timeout* no protocolo 2PC estendido com mensagens de reconhecimento de acordo com as regras mencionadas.

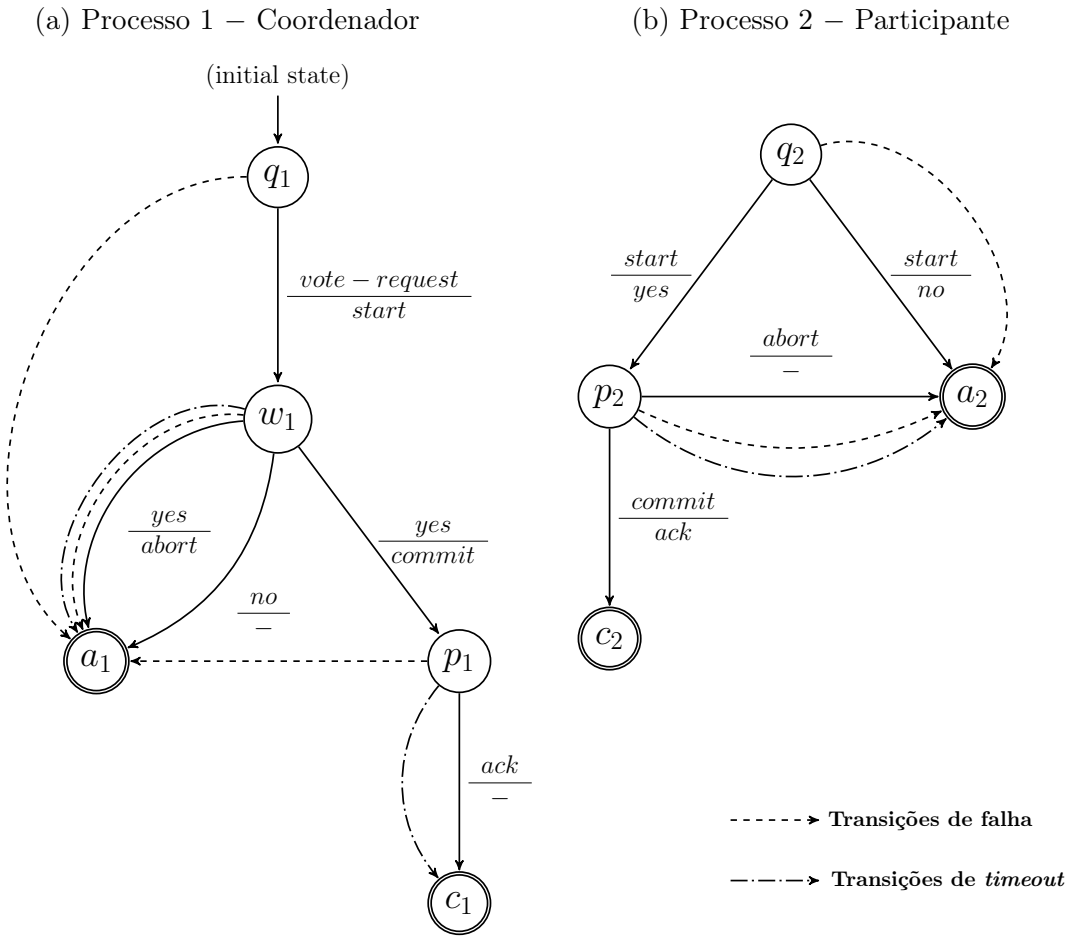
3.5.2 Falhas de dois processos

Um protocolo ser resiliente a pontos únicos de falha não é condição suficiente para ser resiliente a falhas de dois processos. Veja que, se o Processo 2 sofrer uma falha no estado p_2, fará uma transição de falha para o estado de *abort*, a_2; o Processo 1, por sua vez, não receberá a mensagem de reconhecimento do participante, realizando uma transição de *timeout* para o estado de *commit*, c_1, resultando em estados finais inconsistentes. Como não é possível haver outro arranjo de transições de falha e *timeout* para o protocolo da Figura 16, este não é (e não pode ser) resiliente a falhas de dois processos. Essa limitação é formalizada pelo teorema (Teorema 4.3, SKEEN, 1982) a seguir, que também é válido para o caso de múltiplas falhas:

Teorema 1. Não existem protocolos que utilizem recuperação independente e que sejam resilientes a falhas de dois processos

A impossibilidade de protocolos de *commit* garantirem resiliência a falhas múltiplas através da recuperação independente foi demonstrada por meio de uma análise do comportamento de um protocolo em execução. Para isso, considera-se um protocolo que sempre preserva a consistência na ausência de falhas, focando especificamente em execuções cujo estado final seja o de *commit*. A prova mostra que, para cada execução livre de falhas que

Figura 16 – Protocolos locais do algoritmo *Two-Phase Commit* estendido com mensagens de reconhecimento, acrescido de transições de falha e *timeout*



Fonte: SKEEN (1982, figura 4.3, p. 72)

confirma a transação, é possível identificar um ponto no caminho de execução do qual, caso ocorram falhas múltiplas, o sistema se tornará inconsistente. Ou seja, as transições de falha e *timeout* após tais falhas não podem ser organizadas de maneira que levem a um estado global consistente, pois pelo menos um desses processos vai realizar uma transição para o estado de *commit*, enquanto o outro realiza uma transição para o estado de *abort*. Isso mostra que, para recuperar-se de falhas múltiplas, os processos falhos não podem basear-se apenas em informações locais, ressaltando a importância de protocolos de terminação.

3.5.3 Evitando situações de bloqueio

Um protocolo de terminação é utilizado para resolver situações de bloqueio quando uma falha impede que processos operantes continuem suas execuções ou após a recuperação de um participante falho. Os dois problemas são considerados na ordem em que foram citados.

Um participante, p, foi afetado por uma falha de nó em outro processo e inicia o

protocolo de terminação cooperativa. O processo p é o iniciador deste protocolo e comunica-se com o participante q, que é o respondente. Nesse contexto, existem três possibilidades:

1. O processo q já havia recebido e executado a decisão antes da falha do coordenador: q envia "*commit*" ("*abort*") para p, e este prossegue de acordo;

2. O processo q não votou ainda: q pode abortar unilateralmente. Este envia, então, "*abort*" para p, que procede de acordo;

3. O processo q votou "sim" e também não recebeu a decisão do coordenador: q é incerto e, portanto, não pode ajudar p a chegar a uma decisão.

Este algoritmo é descrito no Pseudo Código 1. Um processo que inicia o protocolo de terminação envia mensagens ($DECISION - REQ$) para os demais participantes perguntando se estes possuem conhecimento da decisão global. Do outro lado, se o processo respondente já tiver votado "sim" ou decidido abortar unilateralmente, envia a resposta apropriada. No entanto, se o processo respondente ainda estiver incerto, ele não emite uma resposta imediata, permitindo que o iniciador continue aguardando até que todos os participantes se pronunciem. Caso o *timeout* na espera pela mensagem de decisão seja atingido sem resposta, o processo iniciador é bloqueado. Embora o protocolo reduza a probabilidade de bloqueio, ele não a elimina por completo. Se o iniciador do protocolo de terminação, neste caso p, conseguir se comunicar com outro processo, como q, e as condições 1 ou 2 forem atendidas, p alcançará uma decisão sem ser bloqueado. Caso contrário, se 3 se aplicar a todos os processos com os quais p interagir, ele será bloqueado.

Para que um processo falho possa recuperar-se de uma falha, deve ter memória a respeito de seu estado no momento da falha. Para isso, o *log* TD é utilizado. Observe que um processo tem acesso somente ao seu *log*. Assumindo o uso do protocolo de terminação cooperativa descrito e a utilização do método WAL de manipulação do *log*, as regras de escrita no *log* seguem as seguintes determinações (BERNSTEIN; HADZILACOS; GOODMAN, 1987):

1. O coordenador escreve "*start-2PC*" antes ou depois do envio das mensagens de "*start*";

2. Se o participante votou pela confirmação da transação, o registro "sim" deve preceder o envio da mensagem. Caso contrário, escreve "*abort*" antes ou depois de enviar "não" para o coordenador;

3. Ao decidir confirmar a transação, o coordenador deve registrar "*commit*" antes do envio das mensagens aos participantes. Caso contrário, escreve "*abort*" antes ou depois de enviar tais mensagens;

Pseudo-Código 1 Protocolo de Terminação Cooperativa

```
 1: procedure INITIATOR
 2:     send DECISION-REQ to all processes;
 3:     wait for decision message from any process
 4:         on timeout goto start /* Blocked! */
 5:     end;
 6:     if decision message is COMMIT then
 7:         write a log of local commit to WAL;
 8:     end;
 9:     else /* Decision message is ABORT */ then
10:         write a log of local abort to WAL;
11:     end;
12:     return;
13:
14: procedure RESPONDER
15:     wait for DECISION-REQ from any process p;
16:     if responder has not voted YES or has decided to abort then
17:         send ABORT to p;
18:     end;
19:     else if responder has decided to commit then
20:         send COMMIT to p;
21:     end;
22:     else /* responder is in its uncertainty period */ then
23:         skip; /* Do nothing */
24:     end;
25:     return;
```

Fonte: BERNSTEIN; HADZILACOS; GOODMAN (1987, adaptado da figura 7.4, p. 233)

4. Após receber "*commit*" ("*abort*"), o participante registra "*commit*" ("*abort*").

Acompanhado da escrita de "*start-2PC*", o coordenador registra a lista de identificadores dos participantes, pois em caso de falha do primeiro, os últimos devem se conhecer. Em um cenário de falha, esse registro no *log* de um processo indicará que trata-se do coordenador, distinção importante para o processo de recuperação de falhas. Observe que os registros de "*commit*" são mais críticos que os de "*abort*", pois o protocolo WAL exige que operações de *escrita* sejam registradas antes de serem executadas.

Aplicando tais regras ao protocolo 2PC, quando um processo p se recupera de uma falha, o seguimento da execução da transação depende da interpretação dos registros no *log* TD:

1. Se o *log* TD contém "*start-2PC*", então p é o coordenador:

   (a) O *log* TD contém um registro de "*commit*" ou "*abort*", então o coordenador decidiu antes de falhar e pode dar prosseguimento com esta operação;

   (b) Caso contrário, se ambos registros não forem encontrados, o coordenador pode decidir "*abort*" unilateralmente ao inserir esta ação no *log*.

2. Se o *log* TD não contém "*start-2PC*", então p é um participante:

(a) O *log* TD contém um registro de "*commit*" ou "*abort*". Então, o participante falhou após receber a decisão do coordenador e pode prosseguir de acordo;

(b) O *log* TD não contém um registro "sim", de forma que, o participante não votou, ou votou "não" (mas não registrou "*abort*" no *log*). O participante pode, portanto, abortar unilateralmente inserindo um registro de "*abort*" no *log* de transações;

(c) Existe um registro "sim", mas o *log* não possui a decisão do coordenador registrada. Nesse caso, o participante está no seu período de incerteza e pode tentar atingir uma decisão através do protocolo de terminação.

O Pseudo Código 2 apresenta os protocolos locais executados pelos processos coordenador e participantes acrescido de ações de *timeout* e registros no *log* TD, alterações necessárias para fazer com que o protocolo satisfaça a Condição 5.

O protocolo local do processo coordenador tem início com o envio das mensagens de "*start*" para os participantes, seguido pelo registro de "*start-2PC*" no *log*. Este registro poderia, alternativamente, ter sido realizado antes do envio das mensagens. A seguir, o coordenador aguarda os votos dos participantes durante um tempo determinado por um *timeout*. Dessa forma, o coordenador pode realizar uma transição de *timeout* caso os votos não sejam recebidos dentro do prazo estabelecido. Neste caso, registra a decisão de "*abort*" em seu *log* local e envia sua decisão para aqueles processos que votaram "sim". Quando todos os votos são "sim", o coordenador deve registrar a decisão de "*commit*" em seu *log* antes de enviar um *broadcast* com a decisão global para os demais processos. Se houver pelo menos um voto "não", o coordenador segue o mesmo procedimento de abortar a transação descrito para o caso em que realiza uma transição de *timeout*, podendo registrar a decisão de *abort* em seu *log* antes ou depois de enviar as mensagens. Esse comportamento não prejudicará a detecção de falhas, pois, mesmo que o coordenador falhe antes de registrar sua decisão desfavorável, o protocolo executado pelos participantes também possui um *timeout* para o recebimento da decisão do coordenador.

No protocolo executado pelos participantes, o processo inicia aguardando o recebimento da mensagem de "*start*". Caso não receba essa mensagem dentro do período esperado, o participante registra localmente em seu *log* a decisão de *abort* e encerra sua execução, abortando unilateralmente. Ao receber com sucesso a mensagem de "*start*", o participante pode então enviar seu voto. Quando o participante vota "sim", registra seu voto em seu *log* antes de enviar sua decisão para o coordenador. Em seguida, aguarda a decisão do coordenador. Se esta não for recebida dentro do período de *timeout*, o protocolo de terminação cooperativa é invocado. O participante, por ser incerto, não é capaz de tomar uma decisão unilateral. Por causa disso, numa tentativa de evitar o seu bloqueio, deve invocar o protocolo de terminação. Se a mensagem de decisão for recebida, o processo procede com esta ação. Por outro lado, se o participante votar "não", aborta

unilateralmente ao registrar sua decisão antes ou depois de enviar seu voto negativo para o coordenador.

3.6 FALHAS DE COMUNICAÇÃO

Os processos envolvidos na execução de uma transação distribuída são interconectados por meio de *links* de comunicação bidirecionais, que também estão sujeitos a falhas. Essas falhas ocorrem por diversas razões, como mensagens corrompidas por ruídos no canal, perda total de mensagens devido a interrupções temporárias nos enlaces ou indisponibilidade de nós que causam a divisão da rede em diferentes grupos. Consideram-se falhas de comunicação causadas pela partição da rede em dois ou mais grupos disjuntos, nos quais conjuntos de nós operacionais que pertencem a grupos distintos tornam-se incapazes de trocar mensagens (SKEEN, 1982).

Como consequência de falhas de comunicação, mensagens enviadas podem não chegar ao destinatário, impactando a execução do protocolo. Quando esse problema ocorre devido a partições na rede, dois cenários podem ser considerados. No primeiro, cada processo detecta a partição por meio de um *timeout* e realiza uma transição apropriada, modelando a partição como uma transição global de estado que apaga todas as mensagens pendentes. Essa abordagem estabelece uma analogia entre partições e falhas simultâneas de nós, pois, em ambos os casos, a ausência de mensagens leva à ativação de um *timeout* e à adoção de medidas de recuperação. No segundo cenário, as mensagens são simplesmente perdidas, sem qualquer mecanismo adicional para sua detecção ou retransmissão. A primeira abordagem exige suporte explícito da rede para a sinalização de falhas e, portanto, não é comumente implementada, pois introduz uma complexidade semelhante à dos próprios protocolos de *commit*, apenas deslocando o problema do compromisso atômico para outra parte do sistema. Já a segunda abordagem, de caráter mais realista, reflete com maior precisão as limitações práticas de sistemas distribuídos (BERNSTEIN; HADZILACOS; GOODMAN, 1987).

Ao longo desta seção, abordam-se os desafios diante de partições de rede, considerando inicialmente o caso mais simples de um sistema com dois processos e avançando para cenários mais complexos envolvendo múltiplos nós. Para isso, ambas as abordagens são consideradas. Verifica-se que protocolos resilientes existem apenas para partições simples, sob a abordagem otimista. Em um ambiente de rede que adota a abordagem pessimista, não existem protocolos não bloqueantes, nem mesmo para partições simples (SKEEN, 1982).

3.6.1 Sistema com dois processos

Existe uma correspondência entre a detecção de uma falha de nó, que resulta em uma transição de falha sob recuperação independente, e a detecção de uma falha de

comunicação devido a partição da rede, que leva à realização de uma transição de *timeout*. Para ilustrar tal correspondência, o seguinte é verdadeiro quando um processo operacional, p, detecta a falha de nó ocorrida em q através da expiração do *timeout*:

1. A última mensagem enviada por p não foi recebida (q falhou antes de recebê-la);
2. A comunicação com q é impossível, pois está indisponível em razão de sua falha;
3. O processo q tomará uma decisão através da recuperação independente.

Exatamente as mesmas condições são válidas quando p é notificado de que sua mensagem não foi entregue para q. Assim, os dois modelos são isomorfos, o que faz com que as soluções apresentadas na seção anterior impliquem na solução para o problema em questão, conhecido como o **problema da partição** (SKEEN, 1982). Como consequência do Teorema 1, o seguinte teorema foi formulado (Teorema 4.4, SKEEN, 1982), que também pode ser generalizado para o caso de múltiplos nós.

Teorema 2. Não existem protocolos resilientes a uma partição de rede quando mensagens são perdidas.

Para poder assinalar as transições de *tiemout* e de mensagens não entregues, deve ser considerado o caso otimista em que os nós remetentes são notificados sobre mensagens perdidas. As regras 1 e 2 podem ser generalizadas para o contexto de falhas de comunicação, resultando no protocolo retratado pela Figura 17. Tais regras sempre produzem protocolos que operam com dois processos resilientes a partição simples (Teorema 4.5, SKEEN, 1982).

Regra 3. Para cada estado e_i: Se $C(e_i)$ contém um estado de *commit* (*abort*), logo associe uma transição de *timeout* de e_i para um estado de *commit* (*abort*).

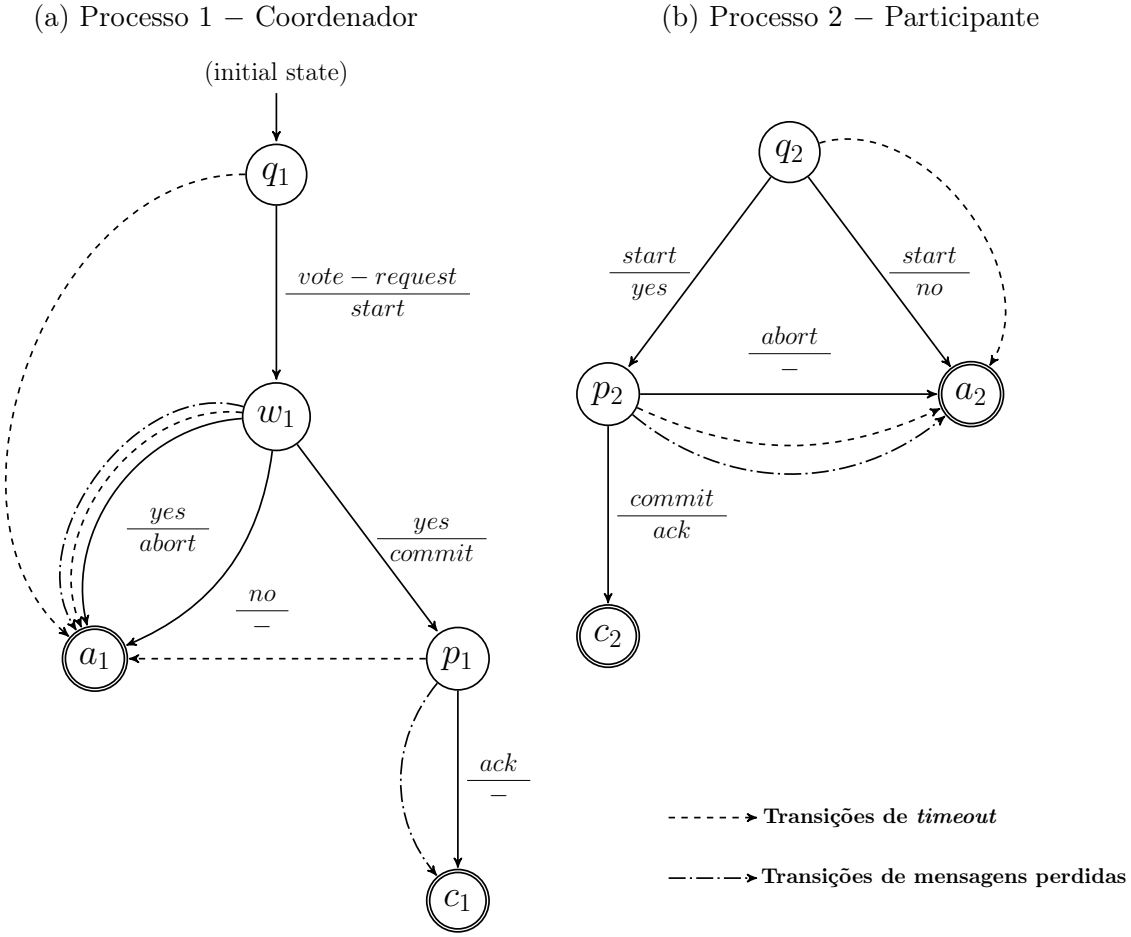
Regra 4. Para cada estado e_j: se $e_i \in S(e_j)$, e e_i possui uma transição de *timeout* para um estado de *commit*, então associe uma transição de mensagem perdida de e_i para um estado de *commit*. Da mesma forma, se e_i possui uma transição de *timeout* para um estado de *abort*, associe uma transição de mensagem perdida de e_i para um estado de *abort*.

3.6.2 Sistema com múltiplos nós

As regras 3 e 4 podem ser estendidas para protocolos com múltiplos nós que são submetidos a uma partição simples, resultando no corolário a seguir (Corolário 4.6, SKEEN, 1982).

Corolário 1. Existem protocolos operando com múltiplos nós que são resilientes a **uma partição simples** quando mensagens não entregues são retornadas ao remetente.

Figura 17 – Protocolos locais do algoritmo *Two-Phase Commit* estendido com mensagens de reconhecimento, acrescido de transições de *timeout* e de mensagens perdidas



Fonte: SKEEN (1982, figura 4.4, p. 81)

Na verdade, a resiliência a partições simples tem relação direta com a abordagem utilizada pela rede. Sabendo que a resiliência é garantida quando a abordagem otimista é utilizada, somado ao Teorema 2, o corolário a seguir pode ser derivado (Corolário 4.7, SKEEN, 1982), que encerra a discussão a respeito da resiliência a essa subclasse de falhas.

Corolário 2. Saber quais mensagens não foram entregues no momento da falha de comunicação é necessário e suficiente para a recuperação de partições simples.

Diante da última subclasse de falhas, partições múltiplas, se a abordagem otimista é utilizada, considera-se que transições de *timeout* e de mensagens perdidas não são afetadas pela ocorrência de partições simples adicionais. Contudo, mesmo nesse modelo extremamente otimista, os resultados demonstram limitações significativas (Teorema 4.8, SKEEN, 1982):

Teorema 3. Não existe nenhum protocolo resiliente a múltiplas partições.

Conclui-se que, para o caso de uma partição simples, a abordagem otimista permite a construção de protocolos resilientes a falhas de comunicação tanto para um sistema com dois processos, quanto com múltiplos nós. Em contrapartida, a abordagem pessimista inviabiliza a resiliência a essa subclasse de falhas, destacando o impacto da perda definitiva de mensagens.

Para o caso de múltiplas partições, mesmo a adoção da abordagem otimista não é suficiente para garantir a resiliência a falhas de comunicação. O Teorema 3 evidencia a inexistência de qualquer protocolo capaz de lidar de forma eficaz com essa subclasse de falhas, indicando que as restrições impostas por múltiplas partições vão além das limitações específicas de uma abordagem otimista ou pessimista.

3.7 RESUMO

Este capítulo abordou as falhas de nós e de comunicação que afetam protocolos de *commit*. As falhas de nós ocorrem quando um processo se torna inativo devido a uma falha. A recuperação independente foi apresentada como uma solução para garantir resiliência a falhas de um único nó em protocolos que não possuem estados cujos conjuntos de concorrência são inconsistentes. No entanto, essa estratégia não é suficiente para falhas múltiplas de nós, como demonstrado no Teorema 1, e, nesses casos, o protocolo de terminação cooperativa é adotado para evitar bloqueios. Quanto às falhas de comunicação, discutiu-se o impacto das partições de rede, que podem dividir a rede em grupos incapazes de se comunicar. A abordagem otimista, em que processos remetentes são notificados de falhas de entrega, oferece resiliência a partições simples, mas não a múltiplas. Já a abordagem pessimista, que não permite protocolos resilientes a partições, resulta em falhas graves quando há perda definitiva de mensagens. Conclui-se que não existem protocolos de *commit* resilientes a partições múltiplas.

Pseudo-Código 2 O Protocolo *Two-Phase Commit*

```
 1: procedure COORDINATOR
 2:     send START to all Participants;
 3:     write a log of start-2PC to WAL;
 4:     wait for vote messages (YES or NO) from all Participants
 5:         on timeout then
 6:             let P_i be the processes from which YES was received;
 7:             write a log of local abort to WAL;
 8:             send ABORT to all processes in P_i;
 9:             return;
10:         end;
11:     end;
12:     if all votes were YES and Coordinator voted YES then
13:         write a log of local commit to WAL;
14:         send COMMIT to all Participants;
15:     end;
16:     else /* There is at least one vote NO */ then
17:         let P_i be the processes from which YES was received;
18:         write a log of local abort to WAL;
19:         send ABORT to all processes in P_i;
20:     end;
21:     return;
22:
23: procedure PARTICIPANT(i)
24:     wait for START from Coordinator
25:         on timeout then
26:             write a log of local abort to WAL;
27:             return;
28:         end;
29:     end;
30:     if participant's vote is YES then
31:         write a yes record in WAL log;
32:         send YES to Coordinator;
33:         wait for decision message (COMMIT or ABORT) from Coordinator
34:             on timeout then
35:                 initiate termination protocol;
36:             end;
37:         end;
38:         if decision message is COMMIT then
39:             write a log of local commit to WAL;
40:         end;
41:         else /* decision message is ABORT */ then
42:             write a log of local abort to WAL;
43:         end;
44:     end;
45:     else /* Participant's vote is NO */ then
46:         write a log of local abort to WAL;
47:         send NO to Coordinator;
48:     end;
49:     return;
```

Fonte: BERNSTEIN; HADZILACOS; GOODMAN (1987, adaptado da figura 7.3, p. 232)

4 PROTOCOLOS DE COMMIT ATÔMICO NÃO BLOQUEANTES

Embora o 2PC resolva o problema do compromisso atômico, ele permite que processos corretos fiquem bloqueados devido a falhas em outros processos. O objetivo deste capítulo é apresentar protocolos não bloqueantes como uma alternativa que elimina essa possibilidade. Ao longo da discussão, exploram-se as condições que devem ser atendidas para que um protocolo seja considerado não bloqueante, conforme descrito pelo Teorema Não Bloqueante (SKEEN, 1982). Como aplicação desse teorema, verifica-se como a adição de estados de *buffer* pode transformar protocolos bloqueantes em não bloqueantes. Essa técnica é ilustrada com a aplicação ao protocolo *Two-Phase Commit*, levando à concepção do protocolo *Three-Phase Commit*.

4.1 PROTOCOLOS NÃO BLOQUEANTES

Um protocolo é dito não bloqueante diante de *falhas de nós* se nunca exigir que processos corretos, i.e., aqueles que nunca ficaram indisponíveis em razão de falhas, sejam bloqueados aguardando a recuperação de um processo falho (SKEEN, 1982). Protocolos não bloqueantes existem somente sob a premissa da não existência de falhas de nós totais e de comunicação. Apesar de não haver diferença na complexidade de mensagens entre protocolos bloqueantes e não bloqueantes, os últimos são mais custosos em tempo quando comparados aos primeiros (DWORK; SKEEN, 1983). Isso ocorre pois protocolos não bloqueantes invocam sub protocolos com objetivo de minimizar situações de bloqueio, como algoritmos para eleger um novo líder caso o coordenador fique indisponível; algoritmos para determinar o último processo a falhar; dentre outros. A próxima seção apresenta o teorema fundamental que possibilitará a análise formal das condições necessárias e suficientes para a existência de protocolos não bloqueantes.

4.2 TEOREMA NÃO BLOQUEANTE

Quando existe a presença de processos falhos na execução de um protocolo de *commit* atômico, processos operacionais devem chegar a um consenso sobre a realização da transação ao analisar seus estados locais. Para esclarecer essa afirmação, considere um cenário em que apenas um nó sobreviveu a falhas. Para ser capaz de inferir o estado da transação sem haver a necessidade de ser bloqueado, este deve analisar seu *conjunto de concorrência local* e aplicar restrições sobre o mesmo com o objetivo de avaliar se é possível prosseguir unilateralmente ou não (SKEEN, 1981):

- Para ser capaz de abortar, não pode haver nenhum estado de *commit* em seu conjunto de concorrência;

- Para ser capaz de confirmar, o estado do processo em questão deve ser confirmável e não pode haver nenhum estado de *abort* em seu conjunto de concorrência.

Dada tais restrições sobre o conjunto de concorrência, é possível definir os dois casos em que uma *situação de bloqueio* pode ocorrer:

- O processo possui conjunto de concorrência inconsistente (contém ambos estados de *commit* e *abort*);
- O processo está em um estado não confirmável e seu conjunto de concorrência contém um estado de *commit*.

No cenário da primeira restrição, o processo em questão não consegue determinar com certeza se a transação deve ser confirmada ou abortada analisando somente seu conjunto de concorrência local, gerando um bloqueio. A segunda restrição gera um impasse pois, pela natureza do estado que o processo ocupa, o processo não conhece o estado dos demais participantes (é um estado não confirmável, i.e., não consegue inferir se os demais votaram "sim"); ao mesmo tempo que não pode abortar unilateralmente pois existe pelo menos um processo que votou "sim" (seu conjunto de concorrência contém um estado de *commit*). Um protocolo no qual ambas as situações descritas pelas restrições não ocorram impediria que processos fossem bloqueados. Com isso em mente, o teorema a seguir foi elaborado (SKEEN, 1981):

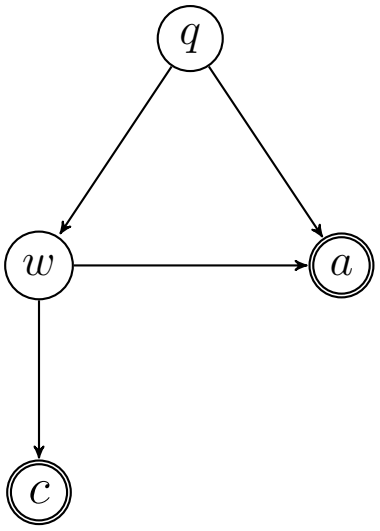
Teorema 4. (Teorema Fundamental Não Bloqueante)
Um protocolo é não bloqueante se e somente se satisfaz as condições abaixo (para cada processo):

1. Não existe nenhum estado local tal qual seu conjunto de concorrência possui ambos os estados de *commit* e *abort*;
2. Não existe nenhum estado não confirmável do qual o seu conjunto de concorrência contém um estado de *commit*.

Este teorema fornece o ferramental para a construção de protocolos não bloqueantes. Para demonstrar isso, considere o protocolo *Two-Phase Commit* canônico que representa o protocolo local dos processos participantes, ilustrado pela Figura 18. Observe que o conjunto de concorrência dos estados deste protocolo canônico estão definidos na imagem. Protocolos de *commit*, quando centralizados, são síncronos dentro de uma transição de estado. Em cada fase destes protocolos, o coordenador envia a mesma mensagem para cada participante, e aguarda uma resposta de cada um. Como consequência, é garantido que os processos progridam aproximadamente na mesma taxa. Por exemplo, se tal característica não fosse válida, o participante não poderia abortar unilateralmente diante do não

recebimento da decisão do coordenador na etapa 2, pois qual seria a garantia de que outro processo não registrou um voto "sim"? Essa característica de protocolos centralizados se traduz na seguinte propriedade.

Figura 18 – Protocolo *Two-Phase Commit* canônico dos participantes, incluindo a definição dos conjuntos de concorrência de seus estados.



Conjunto de concorrência
$C(q) = \{q, w\}$
$C(w) = \{w, a, c\}$
$C(a) = \{w, a\}$
$C(c) = \{c\}$

Fonte: SKEEN (1981, adaptado da figura 5, p. 139)

Propriedade 1 (Síncrono dentro de uma transição de estado). Um protocolo é síncrono dentro de uma transição de estado se um nó nunca está a frente de outro nó por mais de uma transição de estado durante a sua execução.

Já que o protocolo 2PC é síncrono dentro de uma transição de estado, o conjunto de concorrência de um dado estado local só pode conter ele mesmo, referenciando-se aos demais processos que podem estar ocupando este mesmo estado, e seus estados imediatamente alcançáveis, uma vez que os processos nunca se distanciam em mais de uma transição de estado. Ao aplicar a Propriedade 1 ao teorema, surge o seguinte lema (SKEEN, 1981):

Lema 1. Um protocolo que é síncrono dentro de uma transição de estado é não bloqueante se e somente se:

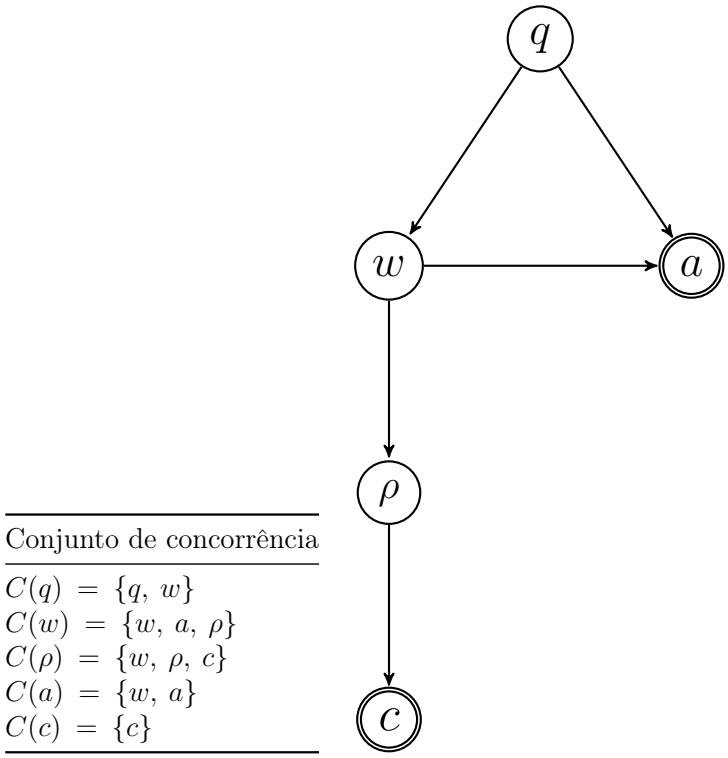
1. Não contém nenhum estado local adjacente a ambos estados de *commit* e *abort* ao mesmo tempo;

2. Não contém nenhum estado não confirmável que é adjacente a um estado de *commit*.

Ao observar o conjunto de concorrência de w definido na Figura 18, nota-se que este é adjacente a ambos estados de *commit* e *abort*; além disso, é um estado não confirmável

adjacente a um estado de *commit*, violando ambas restrições do Lema 1, respectivamente. Ao introduzir um estado de *buffer* (ρ) entre o estado de espera e o de *commit*, o estado w passa a ser adjacente ao estado ρ; e este, por sua vez, é um estado confirmável pois deriva da transição em que o voto é "sim". Portanto, ambas as restrições do lema são atendidas e o protocolo torna-se não bloqueante. O protocolo resultante é retratado na Figura 19, bem como o conjunto de concorrência de seus estados, ilustrando o algoritmo canônico da família de protocolos conhecida como *Three-Phase Commit*. Observe que, com a adição do estado de *buffer*, a inconsistência do conjunto de concorrência de w é resolvida, pois passa a conter ρ, ao invés de c.

Figura 19 – Protocolo *Three-Phase Commit* canônico dos participantes, incluindo a definição dos conjuntos de concorrência de seus estados.



Conjunto de concorrência
$C(q) = \{q, w\}$
$C(w) = \{w, a, \rho\}$
$C(\rho) = \{w, \rho, c\}$
$C(a) = \{w, a\}$
$C(c) = \{c\}$

Fonte: SKEEN (1981, adaptado da figura 6, p. 139)

Não existe restrição a respeito da quantidade de estados de *buffer* a serem adicionados, porém, em muitos casos, a adição de apenas um já é suficiente para torná-lo não bloqueante (SKEEN, 1981).

O Lema 1 é um resultado forte que, porém, só pode ser aplicado a protocolos de *commit* que satisfazem a Propriedade 1. Em (SKEEN, 1982) este resultado foi generalizado para protocolos "menos síncronos".

4.3 RESUMO

Ao garantir que processos corretos não sejam bloqueados devido a falhas, os protocolos não bloqueantes aumentam a disponibilidade de sistemas distribuídos. A existência de protocolos não bloqueantes foi formalizada pelo Teorema Não Bloqueante. A análise do protocolo *Two-Phase Commit* e a introdução do *Three-Phase Commit* ilustram como o teorema pode ser aplicado na prática para transformar um protocolo bloqueante em um protocolo não bloqueante através da adição de um estado de *buffer*.

5 CONCLUSÕES E TRABALHOS FUTUROS

O problema do compromisso atômico surge no contexto de sistemas de bancos de dados distribuídos, nos quais múltiplos processos devem decidir, de forma consistente, entre confirmar (*commit*) ou abortar (*abort*) uma transação distribuída. Nesse contexto, este trabalho teve como objetivo analisar a resiliência de protocolos de *commit* atômico centralizados diante de determinadas classes de falhas. As classes de falhas analisadas foram:

- **Falhas de nós:** ocorrem quando um processo se torna inativo em razão de uma falha.
- **Falhas de comunicação:** provocadas pela partição da comunicação, interrompendo a interação entre processos corretos.

Para contextualizar o estudo em um cenário concreto, o protocolo *Two-Phase Commit* foi utilizado como referência para a discussão.

Falhas de nós ocorrem quando este fica indisponível em razão de uma falha. Nesse contexto, existem duas subclasses de falhas: falhas de um único nó; e falhas de múltiplos nós (de forma concorrente). Verificou-se que, para que um algoritmo seja resiliente a falhas de um único nó sob recuperação independente, é necessário que não haja processos cujos conjuntos de concorrência sejam inconsistentes. Esta estratégia permite que processos tomem decisões sem a necessidade de comunicação entre si, baseando-se somente em informações locais. O protocolo 2PC, na sua forma mais simples, não é resiliente a falhas de um único nó pois o coordenador avança para o estado de *commit* antes que os participantes tenham reconhecido a confirmação da transação. Ainda que a recuperação independente seja atrativa, não pode garantir resiliência a falhas concorrentes de múltiplos nós, ressaltando a necessidade de comunicação entre processos determinada por protocolos de terminação. Em razão disso, foi apresentado um protocolo de terminação cooperativa que pode ser utilizado em conjunto com o 2PC na tentativa de resolver situações de bloqueio. O protocolo de terminação, no entanto, pode não ser capaz de prevenir o bloqueio de processos. Em razão disso, verificou-se que não é possível garantir soluções não bloqueantes para falhas concorrentes.

Falhas de comunicação, por sua vez, ocorrem quando dois processos corretos não conseguem comunicar-se em razão do particionamento da rede, fazendo com que mensagens sejam perdidas. Dentro dessa classificação, existem duas subclasses: uma ocorrência de partição simples; e múltiplas partições. Verificou-se que, se a rede não adotar uma abordagem em que processos remetentes e destinatários sejam notificados de mensagens

perdidas, não existem protocolos resilientes a falhas de comunicação, seja para uma partição simples, ou para múltiplas partições, evidenciando a inviabilidade de uma solução não bloqueante para falhas de comunicação. Medidas de redundância, como o uso de múltiplos canais de comunicação ou retransmissões periódicas, podem ser adotadas para minimizar os impactos dessas falhas.

Constatou-se, então, que soluções não bloqueantes para o problema do compromisso atômico são viáveis somente sob a premissa da inexistência de falhas de nós totais e falhas de comunicação. Com o objetivo de formalizar as condições necessárias e suficientes para a existência de protocolos não bloqueantes, bem como derivar estratégias para transformar protocolos bloqueantes em não bloqueantes, o Teorema Não Bloqueante (Teorema 4) foi apresentado. A partir deste, uma estratégia que baseia-se na adição de estado(s) de *buffer* foi demonstrada utilizando o *Two-Phase Commit*, explicando a concepção do protocolo *Three-Phase Commit*.

Concluindo, este trabalho examinou a resiliência de protocolos de *commit* atômico centralizados diante de falhas de nós e de comunicação, destacando limitações fundamentais desses protocolos em cenários envolvendo tais falhas. Como trabalhos futuros, há o desejo de explorar como o avanço das redes 5G pode impactar protocolos de *commit* atômico em ambientes móveis. Embora o 2PC seja amplamente aplicável em redes fixas porque situações de bloqueio são raras, sua aplicabilidade em ambientes móveis é limitada, pois o bloqueio é, na verdade, parte do comportamento normal do sistema devido a falhas frequentes. Os protocolos não bloqueantes não são adequados nesse cenário, pois causam uma sobrecarga de tempo considerável, sendo aproximadamente 50% mais custosos quando comparados com protocolos bloqueantes (DWORK; SKEEN, 1983). Portanto, a adaptação ou o desenvolvimento de novos protocolos de *commit* atômico, capazes de equilibrar a preservação das propriedades ACID com a resiliência necessária em redes móveis, é uma linha de pesquisa promissora a ser aprofundada.

REFERÊNCIAS

ABDALLAH, M.; GUERRAOUI, R.; PUCHERAL, P. One-phase commit: does it make sense? In: **Proceedings 1998 International Conference on Parallel and Distributed Systems**. [S.l.: s.n.], 1998. p. 182–192.

BERNSTEIN, P. A.; HADZILACOS, V.; GOODMAN, N. **Concurrency control and recovery in database systems**. Addison-Wesley Longman Publishing Co., 1987. v. 370. ISBN 0-201-10715-5. Disponível em: http://www.sigmod.org/publications/dblp/db/books/dbtext/bernstein87.html. Acesso em: 13 jul.2024.

CHORAFAS, D. N. Transaction locks, two-phase commit and deadlocks. In: _____. **Transaction Management: Managing Complex Transactions and Sharing Distributed Databases**. Londres: Palgrave Macmillan, 1998. cap. 11, p. 215–237. ISBN 978-0-230-37653-3. Disponível em: https://link.springer.com/chapter/10.1057/9780230376533_11. Acesso em: 23 nov. 2024.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. **Distributed Systems: Concepts and Design**. 5. ed. [S.l.]: Addison-Wesley, 2011. ISBN 1558606238.

DWORK, C.; SKEEN, D. The inherent cost of nonblocking commitment. In: **PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing**. Nova York: Association for Computing Machinery, 1983. (PODC '83), p. 1–11. ISBN 0897911105. Disponível em: https://dl.acm.org/doi/10.1145/800221.806705. Acesso em: 19 jan. 2025.

ELBAGIR, F. A.; KHALID, A.; KHANFAR, K. A survey of commit protocols in distributed real time database systems. **International Journal of Emerging Trends & Technology in Computer Science**, v. 31, n. 2, p. 61–66, 2016. Acesso em: 4 mar. 2025.

GRAY, J. Notes on data base operating systems. In: **Operating Systems: An Advanced Course**. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 1978. v. 60, p. 393–481. ISBN 3-540-08755-9. Disponível em: https://link.springer.com/chapter/10.1007/3-540-08755-9_9. Acesso em: 25 nov. 2024.

GRAY, J.; REUTER, A. **Transaction Processing: Concepts and Techniques**. 1. ed. San Francisco, California: Morgan Kaufmann Publishers Inc., 1992. 1128 p. ISBN 9781558601901; 9780080519555. Acesso em: 20 out. 2024.

GRAY, J.; TRAIGER, I.; GALTIERI, C. A. Transactions and consistency in distributed database systems. **ACM computing surveys (CSUR)**, Association for Computing Machinery, v. 7, n. 3, p. 323–342, 1982. ISSN 0362-5915. Disponível em: https://doi.org/10.1145/319732.319734. Acesso em: 28 jan. 2024.

HADZILACOS, V. On the relationship between the atomic commitment and consensus problems. In: **Fault-Tolerant Distributed Computing**. Nova York: Springer-Verlag Berlin Heidelberg, 1990. p. 201–208. ISBN 978-0-387-34812-4. Disponível em: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=44edea16c2cf5e6fbeb9a603f9f788f04e1d7eea. Acesso em: 2 nov. 2024.

HAERDER, T.; REUTER, A. Principles of transaction-oriented database recovery. **ACM computing surveys (CSUR)**, Association for Computing Machinery, Nova York, v. 15, n. 4, p. 287–317, 1983. Disponível em: https://dl.acm.org/doi/pdf/10.1145/289.291. Acesso em: 2 nov. 2024.

LAMPSON, B.; STURGIS, H. Crash recovery in a distributed data storage system. **Unpublished technical report, Xerox Palo Alto Research Center**, Xerox Palo Alto Research Center, Palo Alto, California, 1976. Disponível em: https://web.cs.wpi.edu/~cs502/cisco11/Papers/LampsonSturgis_Crash%20recovery_later.pdf. Acesso em: 13 jul. 2024.

LINDSAY, B. G. **Notes on Distributed Databases**. [S.l.], 1979. Disponível em: https://dominoweb.draco.res.ibm.com/reports/RJ2571.pdf. Acesso em: 27 dez. 2024.

MOHAN, C.; LINDSAY, B.; OBERMARCK, R. Transaction management in the r* distributed database management system. **ACM Transactions on Computer Systems (TOCS)**, Association for Computing Machinery, Nova York, v. 11, n. 4, p. 378–396, 1986. Disponível em: https://dl.acm.org/doi/10.1145/7239.7266. Acesso em: 4 mar. 2025.

SKEEN, D. Nonblocking commit protocols. In: **SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data**. Nova York: Association for Computing Machinery, 1981. p. 133–142. ISBN 0897910400. Disponível em: https://dl.acm.org/doi/pdf/10.1145/582318.582339. Acesso em: 13 jul. 2024.

SKEEN, D. **Crash Recovery in a Distributed Database System**. Tese (Doutorado) — EECS Department, University of California, Berkeley, 1982. Disponível em: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1982/7550.html. Acesso em: 25 nov. 2024.

SPECIFICATION, C. **Distributed Transaction Processing (DTP): The XA Specification**. [S.l.]: X/Open, 1991. ISBN 1 872630 24 3. Acesso em: 4 mar. 2025.

TANENBAUM, A. S.; STEEN, M. V. **Distributed Systems: Principles and Paradigms**. 2. ed. [S.l.]: CreateSpace Independent Publishing Platform, 2017. ISBN 1558606238.

ÖZSU, M. T.; VALDURIEZ, P. **Principles of Distributed Database Systems**. 3. ed. [S.l.]: Springer-Verlag Berlin Heidelberg, 1999.