PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

CARLOS RENAN SCHICK LOUZADA

USO DE MENSAGENS EM CADEIA APLICADOS NO PROBLEMA DO MULTICAST ATÔMICO GENUÍNO

Porto Alegre
2024

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

USO DE MENSAGENS EM CADEIA APLICADOS NO PROBLEMA DO MULTICAST ATÔMICO GENUÍNO

CARLOS RENAN SCHICK LOUZADA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Fernando Luís Dotti

Porto Alegre
2024

DEDICATÓRIA

Dedico este trabalho aos meus queridos pais Carlos Alberto e Ingrid que não estão mais neste mundo mas que me observam e me conduzem a fazer o certo e o justo neste dias tão sombrios que vivemos. Dedico também a minha família, a minha amada esposa Luciane e ao meu querido filho Leônidas. Só vocês sabem quantos momentos importantes da nossa vida foram subtraídos pela privação necessária aos estudos.

Dedico sobretudo ao Senhor Altíssimo, que sempre me ajudou e me deu forças para continuar...

“Credo in Deum Patrem omnipotentem, Creatorem caeli et terrae, et in Iesum Christum, Filium Eius unicum, Dominum nostrum, qui conceptus est de Spiritu Sancto, natus ex Maria Virgine, passus sub Pontio Pilato, crucifixus, mortuus, et sepultus, descendit ad ínferos, tertia die resurrexit a mortuis, ascendit ad caelos, sedet ad dexteram Dei Patris omnipotentis, inde venturus est iudicare vivos et mortuos. Credo in Spiritum Sanctum, sanctam Ecclesiam catholicam, sanctorum communionem, remissionem peccatorum, carnis resurrectionem, vitam aeternam.”
(Oração do Credo)

AGRADECIMENTOS

Quero agradecer aos bons colegas da PROCERGS que me ajudaram durante este longo processo. Quero agradecer muito aos professores do PPGCC da PUCRS, sobretudo ao professor Dotti, por toda a sua abnegação e condução dos seus alunos na busca pelo conhecimento. O tamanho da minha gratidão só não é maior se comparado com a minha admiração.

USO DE MENSAGENS EM CADEIA APLICADOS NO PROBLEMA DO MULTICAST ATÔMICO GENUÍNO

RESUMO

O multicast atômico provê garantias de entrega e ordem a sub-conjuntos de processos destinatários, sendo um mecanismo fundamental para o provimento de serviços escaláveis com consistência forte. Enquanto muitos algoritmos genuínos de multicast atômico são derivados do algoritmo de Skeen, que usa comunicação de todos para todos processos, temos em outro lado do espectro o protocolo também genuíno de Delporte-Gallet e Fauconnier. Este restringe a direcionalidade da comunicação e é atrativo por sua simplicidade. Sofre porém do efeito comboio. A partir de avaliações de ambos, esta dissertação propõe um protocolo alternativo, com o objetivo de eliminar o efeito comboio de Delporte-Gallet e Fauconnier, contudo abrindo mão da genuinidade em alguns momentos. Este é o primeiro algoritmo multicast atômico parcialmente genuíno que se utiliza de mensagens em cadeia para garantir a ordem global acíclica dentro de um grafo acíclico dirigido. Além disso, uma proposta de aceleração do protocolo é apresentada, de tal modo que, em determinadas cargas de trabalho o algoritmo torna-se totalmente genuíno e com uma alta vazão.

Palavras-Chave: multicast atômico semi-genuíno, grafo acíclico dirigido, relógio vetorial de aresta, TPC-C.

USE OF CHAIN MESSAGES APPLIED TO THE GENUINE ATOMIC MULTICAST PROBLEM

ABSTRACT

Atomic multicast provides delivery and order guarantees to subsets of recipient processes, being a fundamental mechanism for providing scalable services with strong consistency. While many genuine atomic multicast algorithms are derived from Skeen's algorithm, which uses communication of all for all processes, we have on the other side of the spectrum the also genuine protocol of Delporte-Gallet and Fauconnier. This restricts the directionality of communication and is attractive for its simplicity. However, it suffers from the convoy effect. Based on evaluations of both, this dissertation proposes an alternative protocol, with the aim of eliminating the convoy effect of Delporte-Gallet and Fauconnier, however giving up genuineness at times. This is the first genuine partially atomic multicast algorithm that uses chain messages to guarantee global acyclic order within a directed acyclic graph. Furthermore, a proposal to accelerate the protocol is presented, such that, in certain workloads, the algorithm becomes completely genuine and with a high throughput..

Keywords: semi-genuine atomic multicast, directed aciclic graph, edge vector clock, TPC-C.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE ABREVIATURAS

DCC. – Algoritmo de multicast atômico DaisyChainCast

D&F. – Algoritmo de multicast atômico Delporte-Gallet e Fauconnier

EVC. – Relógios Vetoriais de Aresta

GAD. – Grafo Acíclico Dirigido

LAN. – Rede de curta distância

NUMA. – Acesso não Uniforme a Memória

OLTP. – Processamento de Transações Online

TPC-C. – Desempenho de Processamento de Transações C

UNICAST. – Endereçamento de uma mensagem a um único destino

WAN. – Rede de longa distância

SUMÁRIO

1. INTRODUÇÃO

Serviços computacionais online atuais têm requisitos importantes de alta disponibilidade, desempenho e segurança (e.g. Cloud, BigData, registros distribuídos, sistemas bancários, etc.). A alta disponibilidade implica, entre outras coisas, no uso de técnicas de replicação para prover tolerância a falhas. Entre as formas de replicação temos a ativa e a passiva. A replicação ativa é bastante difundida, sendo chamada de Replicação Máquina de Estados (State Machine Replication - SMR) [16, 20].

O desempenho de serviços online pressupõe a alta vazão de operações, com respostas de baixa latência. Alcançar alta vazão em sistemas SMR não é uma tarefa simples, pois além do sistema estar sujeito às velocidades de diferentes réplicas, a necessidade de determinismo entre as réplicas implica no uso de algoritmos de difusão atômica (equivalente ao consenso) para assegurar a mesma ordem de entrada em todas réplicas.

Além de técnicas para aumentar a vazão de cada réplica do sistema, uma forma de aumento de vazão é através do particionamento do estado do serviço. Cada partição (ou *shard*) é mantida de forma replicada, podendo-se empregar diferentes grupos de replicas para diferentes partições. Requisições a partições diferentes podem ser processadas em paralelo. Entretanto, existem requisições que necessitam de acesso a mais de uma partição (cross-shard). Neste caso, requisitos de ordenação entre partições devem ser impostos. Neste contexto, o multicast atômico [8] é uma abstração fundamental, pois processos podem transmitir para diferentes conjunto de destinos (shards), com possíveis sobreposições, mantendo a ordem total acíclica [8].

A abstração de multicast atômico [9] atende a grande parte destes requisitos [18], sendo um bloco importante para o provimento de sistemas particionados de alta disponibilidade e com consistência forte. Para que sejam escaláveis, no entanto, protocolos de multicast atômico devem ser preferencialmente *genuínos* [14]. De maneira informal o multicast é dito genuíno se, durante uma execução, para a entrega ordenada somente os processos destinatários e origens envolvidos participam da comunicação. Entre os protocolos de multicast genuíno, apesar de não tolerante a falhas, o protocolo de Skeen [3] tem papel importante. Ele usa timestamps lógicos para atribuir timestamps finais às mensagens, tendo inspirado variantes tolerantes a falhas. O protocolo de Skeen supõe a conectividade entre processos como um grafo dirigido completo, isto é, onde todo processo pode se comunicar diretamente com qualquer outro processo.

No que tange esta topologia de conectividade entre processos, em outro lado do espectro temos o protocolo proposto em [10], chamado doravante simplesmente de D&F, que impõe uma ordem total aos processos e permite comunicação somente no sentido de um processo de menor ordem para um de maior ordem. Isto resulta em um grafo acíclico dirigido para a disseminação de mensagens. Entretanto, para garantir a ordem

global acíclica, o último processo destinatário de uma mensagem deve sincronizar com os demais destinatários através de uma mensagem de final.

O protocolo D&F sofre do efeito comboio devido à sincronização dos processos para finalizar a entrega de cada mensagem exibindo baixa vazão e alta latência em diferentes situações. Conforme observado em [1] protocolos de multicast atômico sofrem do chamado efeito comboio (*convoy*), que significa que a entrega de mensagens locais pode ser atrasada devido à necessidade de entrega de mensagens globais.

1.1 Motivação

Enquanto o protocolo de Skeen é alvo de desdobramentos na literatura, principalmente no sentido de criar variantes tolerantes a falhas, nesta dissertação questionamos sobre a possibilidade de melhoria de desempenho do protocolo D&F e investigamos a possibilidade de eliminar a sincronização de processos em D&F. A sincronização que se busca eliminar é central para manter a ordem total acíclica de mensagens: como cada processo destinatário de uma mensagem m espera até que o último processo confirme a entrega de m, qualquer mensagem m' posterior, envolvendo quaisquer subconjuntos de processos comuns com $m.dst$ será entregue após m em todos estes processos.

1.2 Objetivo

Dada a motivação, propomos então um protocolo que chamamos DCC (Daisy Chain Cast). DCC elimina a necessidade de sincronização de D&F. Entretanto, para isso abre mão de genuinidade em alguns casos, sendo um protocolo semi-genuíno. Além disso, D&F faz uso de informação de causalidade junto às mensagens para garantir a ordem global acíclica. Além da proposta do protocolo, discutimos resultados de desempenho de um protótipo de DCC ao lado de resultados de implementações próprias de Skeen e D&F na mesma plataforma e com as mesmas cargas de trabalho.

1.3 Organização do documento

Devido às especificidades do tema, o trabalho se organiza da seguinte forma. No segundo capítulo se coloca os fundamentos mais importantes sobre o assunto, tendo como o objetivo fornecer a sustentação necessária ao leitor perante a proposta. Para tanto, se abordam os principais modelos de falhas em sistemas distribuídos se apresentam os conceitos de replicação máquina de estados e o consenso. Em uma escala cres-

cente de conteúdo, se disserta sobre a importância da difusão atômica e a sua generalização (multicast atômico), cada um com as suas propriedades inerentes.

Para ilustrar as primitivas de difusão e multicast atômico, no terceiro capítulo, alguns algoritmos são discutidos em maiores detalhes com atenção especial ao Skeen e D&F que representam a base teórica comparativa da discussão. Os dois algoritmos, vale observar, foram projetados para trabalhar em um ambiente livre de falhas. Várias sofisticações e otimizações do algoritmo Skeen surgiram nesse meio tempo trazendo consigo o suporte a tolerância a falhas do tipo crash. Protocolos como o RamCast e White-Box são exemplos dessa evolução. Por contraste, o ByzCast é único até então protocolo tolerante a falhas do tipo bizantino que se tem conhecimento.

O quarto capítulo é o cerne da dissertação, começando pela intuição do protocolo DCC até chegarmos à ideia central do seu funcionamento. A seguir discorre-se sobre questões estruturantes do DCC e o algoritmo propriamente. Uma discussão envolvendo a corretude ajuda a compreender porque as propriedades de progresso e corretude estão mantidas.

O quinto capítulo apresenta as variações do protocolo DCC prevendo a tolerância a falhas dos tipos colapso e bizantino. Novamente é posta uma discussão a acerca da corretude e progresso destes algoritmos.

No sexto capítulo são as apresentados os detalhes sobre a metodologia do experimento. No sétimo, os resultados obtidos pelo protocolo DCC e a discussão da representatividade destes números frente aos seus antecessores.

Por fim, no oitavo capítulo são apresentadas as considerações finais a cerca da dissertação e os trabalhos futuros.

2. FUNDAMENTAÇÃO TEÓRICA

Para que seja possível um melhor entendimento daquilo que será apresentado na proposta, é necessário detalhar alguns conceitos em sistemas distribuídos. Deste modo, este capítulo apresenta os principais conceitos teóricos que serão empregados neste trabalho. De forma sequencial e crescente são apresentados os modelos de falha em 2.1, a replicação máquina de estado em 2.3, o problema do consenso em 2.4, os diferentes tipos difusão atômica em 2.5, o multicast atômico em 2.6 e a importante propriedade da genuinidade em 2.7.

2.1 Modelos de Falhas

Sempre que se considera algoritmos ou protocolos distribuídos, o posicionamento a respeito da tolerância a falhas do sistema é inerente (ou obrigatório). Para tal, faz-se necessário definir sobre os tipos de falhas de interesse. A literatura conta com algumas classes. Neste texto, mencionaremos as classes de falhas de maior interesse.

2.1.1 Colapso

Em termos mais simples, quando um processo pára de executar passos, esse mesmo processo potencialmente estará em um estado de falha. Neste estado de falha o processo cessa o seu funcionamento para sempre, portanto ele não deverá mais realizar qualquer atividade incluindo enviar, transmitir ou receber qualquer mensagem [8].

Dentre as categorias de *crash* é possível apresentar as seguintes: *Crash-Stop* e *Crash-Recovery*. Na falha do tipo *crash-stop* se assume que o processo executa o algoritmo de forma correta mas em determinado momento o processo colapsa e, deste modo, não consegue mais dar qualquer passo. Assume-se que o processo permanece fora do sistema, não sendo reiniciado. De forma análoga a falha do tipo *crash-recovery* também se assume que um processo executa o algoritmo de forma correta e ele falha, todavia ele se recupera, retornando ao sistema. Caso o processo permaneça indefinidamente colapsando e se recuperando, o processo é dito como falho.

De acordo com [4] é possível um processo falhar e parar de enviar mensagens e depois se recuperar. Essa característica pode ser vista como uma falha de omissão, mas com uma exceção. Esse mesmo processo que colapsa pode sofrer uma perda total do seu estado interno, uma espécie de amnésia do processo. Essa amnésia pode vir a prejudicar a modelagem do algoritmo porque, após a recuperação, o processo pode enviar novas

mensagens que venham a contradizer as mensagens anteriores a falha. Para tratar este tipo de problema, isto é, a memória ser volátil e não persistente, um armazenamento estável (também chamado de log) pode ser usado para armazenamento e recuperação destas operações.

2.1.2 Omissão

Em poucas palavras, a falha de omissão acontece quando o processo se omite em realizar determinada tarefa em um sistema distribuído. Por exemplo, é dito como omisso um processo que deveria transmitir ou entregar uma mensagem de acordo com o seu algoritmo e não o faz. As causas mais comuns podem estar relacionadas ao congestionamento de rede até situações mais graves como estouro de buffer entre outros [4].

2.1.3 Bizantino

A falha bizantina, diferentemente das demais, é aquela falha arbitrária cujo processo se desvia de qualquer possibilidade prevista no algoritmo, podendo ser de natureza maliciosa ou não (falhas elétricas ou de software). O comportamento de uma falha arbitrária é bastante genérico e o mais difícil entre os modelos de falha a ser detectado e tratado, na medida que, nenhuma premissa sobre o comportamento de um processo defeituoso é estabelecido. Isto é, processos bizantinos podem gerar dados arbitrários, permitindo qualquer tipo de saída ou transmissão de mensagem [4].

Vale observar que, uma falha arbitrária não é necessariamente de natureza intencional ou maliciosa. A falha bizantina pode, por exemplo, ser causada por um erro de implementação, pode estar presente na linguagem de programação ou, simplesmente, embutida no próprio compilador. Este erro, por assim dizer, pode fazer com que o processo se desvie do algoritmo que supostamente deveria executar. Falhas que são acionadas por erros benignos podem ser detectadas e seus efeitos colaterais eliminados através da verificação dupla dos resultados e, a colocação de redundância no próprio processo ou a partir de outros processos [4]. Falhas arbitrárias não maliciosas geralmente aparecem de forma aleatória ou seguem uma certa distribuição uniforme. Por outro lado, as falhas arbitrárias maliciosas são aquelas que assumem a existência de um atacante que coordena as ações dos processos com falha. A partir do controle do nodo, não há limites propriamente com relação ao que um atacante possa fazer do ponto de vista do processo defeituoso [4]. Como exemplo de atividades maliciosas podemos citar o acesso e a leitura

das mensagens nos canais de comunicação, a inserção e a própria alteração no conteúdo destas mensagens, entre outras atividades.

As falhas arbitrárias tipicamente são as mais custosas de serem tratadas e, na maioria dos casos, essa opção somente é aceitável quando falhas desconhecidas e imprevisíveis possam acontecer ou, quando um sistema é vulnerável a ataques cujo adversário possa deliberadamente tentar impedir a operação correta do sistema [4].

2.2 Difusão Confiável

A difusão confiável, é um conceito crítico em sistemas distribuídos, onde múltiplos processos necessitam de uma comunicação eficaz e confiável. A essência desse conceito reside em garantir que as mensagens enviadas por um processo sejam recebidas corretamente por todos os outros processos participantes, apesar dos desafios inerentes aos sistemas distribuídos, como latência de rede, falhas de hardware e software, e problemas de sincronização [4].

Em sistemas distribuídos, a comunicação entre processos é frequentemente sujeita a falhas e inconsistências. Por exemplo, uma mensagem enviada pode ser perdida ou chegar fora de ordem. A difusão confiável aborda esses problemas fornecendo garantias de que, se um processo envia uma mensagem, todos os processos corretos (aqueles que não falharam) eventualmente entregam a mensagem [4].

Aprofundando o conceito de difusão confiável, existem variações que diferem de acordo com a confiabilidade da disseminação. Por exemplo, a difusão de melhor esforço garante que todos os processos corretos entreguem o mesmo conjunto de mensagens se os remetentes estiverem corretos. Formas mais robustas como a difusão confiável uniforme garantem essa propriedade mesmo se os remetentes falharem durante a transmissão de suas mensagens. Variações ainda mais robustas como difusão atômica garantem a ordem total entre os processos. As falhas arbitrárias e os processos bizantinos também estão contemplados, tendo variações específicas que garantam a consistência como nas difusões bizantinas consistentes e confiáveis [4].

Para implementar a difusão confiável, diversos mecanismos e protocolos são usados. Um dos métodos comuns é a retransmissão de mensagens, onde as mensagens são enviadas repetidamente até que o envio seja confirmado por todos os destinatários. Os protocolos podem usar reconhecimentos (ACKs) para confirmar o recebimento das mensagens [4].

Em resumo, a difusão atômica desempenha um papel vital na garantia de comunicações confiáveis e consistentes em ambientes onde falhas e incertezas são a norma. Ele permite que sistemas distribuídos funcionem de maneira eficiente e confiável, apesar dos desafios inerentes à comunicação.

2.3 Replicação Máquina de Estados

Um conceito importante em sistemas distribuídos é a replicação máquina de estados (State Machine Replication ou SMR), capaz de prover a tolerância a falhas. De forma breve, cria-se um serviço de alta disponibilidade implantando-se cópias desse serviço em diferentes máquinas, ditas réplicas, de tal forma que falhas em uma minoria de réplicas possam ser toleradas caso aconteçam. Este conceito permite a continuidade do serviço mesmo que uma falha em um subconjunto de máquinas possa ocorrer [4].

Para a replicação ser efetiva, as diferentes réplicas precisam ser mantidas em um estado consistente, ou seja, os valores não podem divergir entre elas. Se as réplicas trabalharem de forma determinística, um modo simples de manter a consistência é garantir que todas as réplicas iniciem no mesmo estado e recebam, na mesma ordem, o mesmo conjunto de requisições. Para tal, uma primitiva chamada de Difusão Atômica atende essa necessidade, de modo que, um cliente possa realizar uma difusão com ordem total ao conjunto de réplicas da SMR [4]. Uma vez que cada réplica mantiver o mesmo estado, todas as respostas deverão ser iguais, independentemente da máquina onde a requisição for atendida. Para resumir, o uso da difusão atômica garante que o objeto esteja altamente disponível, mas sugere uma entidade lógica única, acessada de maneira sequencial e livre de falhas, onde as operações atuam atomicamente em cada estado da SMR.

2.4 Consenso

O problema do Consenso instancia-se quando um conjunto de processos distribuídos precisam entrar em acordo sobre a adoção de valores, propostos entre eles. A característica marcante é que todos os processos envolvidos precisam decidir por (adotar) um único mesmo valor proposto. O consenso é um módulo fundamental em diversos algoritmos distribuídos, principalmente em sistemas tolerantes a falhas distribuídos. Na difusão atômica o consenso estabelece a mesma ordem das mensagens para todos processos; no problema da efetivação de transações distribuídas o consenso leva ao resultado único de efetivação ou aborto da transação em todas as bases distribuídas; na comunicação em grupos de processos, o consenso é utilizado para implementar a comunicação com sincronia de visão (*view-synchronous communication*), entre outros. O consenso pode ser definido por quatro propriedades elementares [4]:

- **Terminação**: todos os processos corretos eventualmente decidem por um valor.
- **Validade**: se o processo decide v, então v foi proposto por algum processo;

- **Integridade**: nenhum processo decides duas vezes;
- **Acordo**: não existe quaisquer dois processos que decidam de forma diferente;

Múltiplas variações de consenso estão disponíveis na literatura, conforme o modelo de sistema. Uma distinção básica nas propriedades é a uniformidade, ditando se uma propriedade aplica-se a todos processos (uniforme) ou somente aos corretos (regular). Algoritmos para fail-stop, fail-noisy, fail-recovery, e para falhas bizantinas são apresentados em [4] por exemplo. Para cada caso, os algoritmos podem se basear em diferentes estratégias tais como: diferentes detectores de defeitos; suposições com relação a maioria de participantes e uso de quóruns; algoritmos randomizados; entre outros [4].

2.5 Difusão Atômica

A difusão ou broadcast, em geral, é aplicada na disseminação da informação entre um conjunto de processos quaisquer e se diferencia conforme as maneiras como essa disseminação é feita do ponto de vista da confiabilidade e da ordem entre as mensagens.

A difusão atômica, também chamada de difusão em ordem total[1], é importante sempre que um conjunto de processos deseja ter a mesma sequencia de entradas, na mesma ordem. Isto é crucial para manter consistência forte em sistemas distribuídos. A difusão atômica é equivalente ao problema do consenso. Este problema parece simples a princípio, entretanto em sistemas assíncronos e na possibilidade de uma falha por colapso tais algoritmos não garantem terminação [11].

O problema do broadcast atômico é definido em termos das primitivas TO-broadcast(m) e TO-deliver(m). Onde *broadcast* é o ato do envio, pelo originador, a todos processos destinatários e *deliver* é o ato de entrega no destinatário da mensagem m para o usuário do broadcast atômico, cumprindo então a semântica do broadcast atômico. Esta semântica é dada pelas seguintes propriedades [8]:

- **Validade**: Se um processo correto faz um broadcast de m, então todos os processos corretos entregam m;
- **Acordo Uniforme**: Se um processo entrega a mensagem m, então todos os processos corretos entregam m;
- **Integridade Uniforme**: Para toda mensagem m, todo processo entrega m não mais que uma vez, e somente se m foi previamente difundida pelo seu originador;

[1] As expressões Total Order Broadcast e Atomic Broadcast têm mesmo significado.

- **Ordem Total Uniforme**: Se ambos os processos *p* e *q* entregam as mensagens *m* e *m'*, então *p* entrega *m* antes de *m'*, se e somente se o processo *q* entrega *m* antes de *m'*.

A expressão *processo correto* se refere a um processo que não falha na execução em consideração. Analogamente, um *processo falho* é um processo que falha em algum ponto da execução em consideração. Propriedades uniformes são as garantidas igualmente para processos falhos e corretos. Por exemplo, no acordo uniforme: se um processo (qualquer - correto ou falho) entrega uma mensagem, todo processo correto entregará.

As primeiras duas propriedades são ditas de vivacidade (ou liveness), as últimas duas são ditas propriedades de corretude (ou safety). Propriedades de liveness devem se tornar verdade em algum ponto no funcionamento do sistema (do inglês eventually). Propriedades de safety devem ser verdade em todo estado do sistema.

Quando consideramos falhas bizantinas, as propriedades de uniformidade não podem ser garantidas tal como descritas, pois não se pode garantir nenhum comportamento do processo falho - devido à natureza da falha. Adaptações a estas propriedades existem na literatura.

2.6 Multicast Atômico

Dado que, no broadcast toda mensagem é enviada a todos os processos do sistema, no multicast mensagens são enviadas a subconjuntos destes processos. Em [8] estes subconjuntos são chamados grupos. Assim, no multicast existem $M >= 1$ grupos de processos $B = \{G_1, G_2, \ldots, G_M\}$, onde G_i é um grupo não vazio e a intersecção entre grupos pode ser não vazia. O grupo de destinatários de uma mensagem *m* é chamado *m.dst*, ou seja, *m* será entregue em todos os processos que fazem parte do conjunto *m.dst*. Assim, o broadcast atômico é um caso especial do multicast atômico onde um grupo contém todos os processos.

Antes de definir o multicast atômico, se apresenta o conceito do multicast confiável (reliable), definido pelas seguintes propriedades.

- **Validade**: Se um processo correto faz um multicast de *m*, então todos os processos corretos que pertençam ao grupo *m.dst* entregam *m*;
- **Acordo**: Se um processo correto entrega a mensagem *m*, então todos os processos corretos em *m.dst* entregam *m*;
- **Integridade**: Para toda mensagem *m*, todo processo $p \in m.dst$ entrega *m* não mais que uma vez, e somente se *m* foi previamente multicast pelo seu originador.

O multicast atômico pode ser visto como uma especialização do multicast confiável com restrições fortes à ordem na entrega das mensagens. Deste modo é possível um comportamento que leve a intersecção de grupos multicast quando da necessidade de transmissão de mensagens para múltiplos grupos [8]. Há vários tipos de multicast atômicos que diferem neste quesito, ou seja, a capacidade de atender a determinados requisitos na entrega, com três tipos predominantes.

O primeiro tipo é o multicast atômico com ordem total local. Neste paradigma se aplicam restrições mais fracas com relação a ordem das mensagens, uma vez que se requer que a ordem total seja aplicada somente para mensagens multicast do mesmo grupo [8]. Necessita que a seguinte propriedade adicional esteja presente:

- **Ordem Total Local**: Se ambos os processos corretos *p* e *q* entregam as mensagens *m* e *m'* e *m.dst = m'.dst*, então *p* entrega *m* antes de *m'*, se e somente se o processo *q* entrega *m* antes de *m'*.

O segundo tipo é o multicast atômico com ordem total de pares. Diferentemente do anterior, se aplicam restrições mais fortes com relação a ordem das mensagens, dado que, se requer que a ordem total seja aplicada para todas as mensagens entregues na intersecção de dois grupos [8]. Se comparada com o multicast confiável, necessita da seguinte propriedade:

- **Ordem Total de Pares**: Se dois processos corretos *p* e *q* entregam as mensagens *m* e *m'*, então *p* entrega *m* antes *m'*, se e somente se *q* entrega *m* antes de *m'*.

O multicast atômico com ordem total de pares pode apresentar resultados inesperados caso haja a intersecção de três grupos de destino ou mais na transmissão [8]. Por exemplo, considere três processos *pi*, *pj*, *pk*, e três mensagens *m1*, *m2*, *m3* que respectivamente são enviadas por três grupos sobrepostos *G1 = {pi,pj}*, *G2 = {pj,pk}* e *G3 = {pk,pi}*. A figura 2.1 ilustra esta situação.

- $pi : ...TOMdeliver(m_3)...TOMdeliver(m_1)...$
- $pj : ...TOMdeliver(m_1)...TOMdeliver(m_2)...$
- $pk : ...TOMdeliver(m_2)...TOMdeliver(m_3)...$

Para contornar este problema surgiu o multicast atômico com ordem total global, uma sofisticação capaz de equacionar este problema a partir da não tolerância de ciclos. Para tal, requer uma propriedade ainda mais forte:

- **Ordem Total Global**: Seja a relação $<$ definida como $m < m'$ se e somente se algum processo entrega m e então m'. A relação $<$ é acíclica.

Esta propriedade diz que as observações de ordem entre mensagens, em cada processo, se consideradas em conjunto, não podem formar um ciclo.

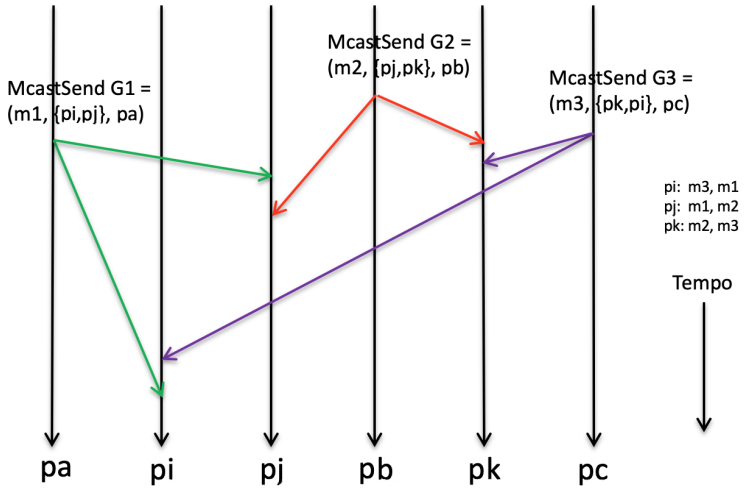


Figura 2.1 – Diagrama de tempo que demonstra o problema do ciclo.

2.7 Multicast e Genuinidade

De acordo com [8], o multicast atômico é uma generalização da difusão atômica de um para vários grupos de destinatários. Uma abordagem simplista para implementar o multicast atômico a partir da difusão atômica seria difundir as mensagens a todos processos e somente os pertencentes a $m.dst$ entregariam m. Entretanto, essa estratégia acabaria por enviar m para todos os participantes do sistema, inclusive àqueles não fazem parte do destino. Em outras palavras, todos os processos no sistema estariam direta ou indiretamente envolvidos e a implementação seria tão cara quanto a implementação de um algoritmo de difusão atômica, restringindo a escalabilidade.

Um algoritmo de multicast atômico genuíno é um algoritmo que resolve o problema do multicast atômico em que apenas o remetente da mensagem m e os processos destinatários $m.dst$ estão envolvidos no multicast atômico da mensagem m. Evidentemente que o problema da não genuinidade está diretamente relacionado a escalabilidade. Isto é, quanto maior o número de participantes operando um algoritmo multicast atômico não genuíno, mais agravado o cenário estará, na medida que mais mensagens proporcionalmente precisarão ser trafegadas, descartadas e mais processos se envolverão nestas ações.

Segundo [14], um algoritmo multicast atômico genuíno A é aquele algoritmo que para cada execução R de A, a seguinte propriedade adicional é satisfeita [14]:

- **Minimalidade**: Para qualquer processo $p \in \Omega$, a menos que uma mensagem m seja R-multicast em R e $p \in m.dst \cup m.src$, p não participa deste multicast.

Esta propriedade da minimalidade reflete diretamente na escalabilidade de um algoritmo multicast atômico genuíno. Não menos importante é observar que esta propri-

edade deve ser aplicada somente ao conjunto Ω de processos. Deste modo elementos de conectividade que, por ventura, estejam entre os processos envolvidos no multicast não prejudicam a genuinidade do algoritmo [14].

2.8 Latência vs Vazão

Se formos observar, a grande maioria dos protocolos voltados a sistemas distribuídos são pensados e arquitetados tendo como um dos elementos centrais a redução da latência no processo de difusão das mensagens [13]. A latência, em poucas palavras, corresponde ao tempo necessário para a difusão de uma mensagem sem contenção. De acordo com modelo de Lynch [13], a latência é capturada pelo números rodadas ou passos, onde no início de cada rodada um processo *p* pode mandar uma mensagem para um ou mais processos. No final da rodada, o processo recebe a mensagem enviada por outro processo. A vazão, por outro lado, mede o número de mensagens que se consegue fazer chegar a um ou vários destinatários, conforme o protocolo em questão, por unidade de tempo [13].

Em alguns casos, como por exemplo, aqueles cujo o ambientes possui uma alta carga de trabalho, a vazão pode inclusive ser mais importante que a latência. Em casos extremos associados a baixa vazão e a alta carga de trabalho, é possível o enfileiramento indefinido de mensagens causar o não progresso. Portanto, comparativamente não é incomum encontrar protocolos com boas taxas de latência, mas com uma vazão abaixo do esperado.

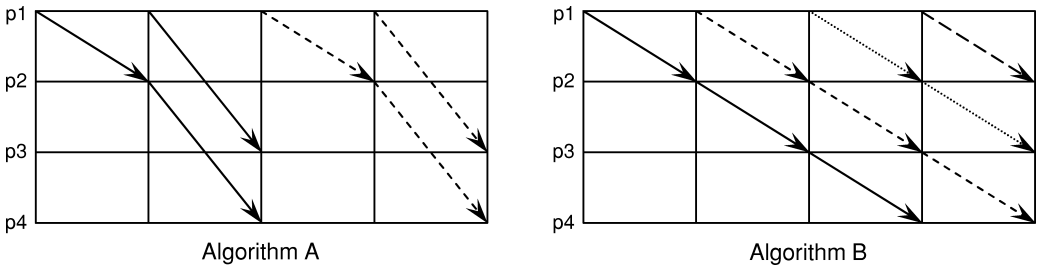


Figura 2.2 – Latência vs vazão retirado de [13].

A figura 2.2 procura demonstrar esse fenômeno. Por exemplo no algoritmo *A* (esquerda), o processo *p1* envia a mensagem ao *p2*. No próximo passo *p2* encaminha para *p4* e, simultaneamente, *p1* envia para *p3*. No algoritmo *B* (direita), o processo *p1* envia a mensagem para *p2*, que encaminha para *p3*, que encaminha para *p4*. Do ponto de vista de latência, o algoritmo *A* é o vencedor na medida que ele apresenta 2 passos de comunicação e o algoritmo *B* apresenta 3. Todavia se observarmos a vazão, o algoritmo *B* será o vencedor pois ele consegue enviar um nova difusão a cada 1 unidade de tempo ao passo que no algoritmo *A* será em 2 unidades de tempo.

3. TRABALHOS RELACIONADOS

No campo da pesquisa e daquilo que se considera os principais algoritmos de multicast atômico, faz-se necessário citar aqueles que constituem a base algorítmica daqueles que hoje são considerados o estado da arte no tema. Então começaremos apresentando os algoritmos clássicos destinadas para um ambiente livre de falhas em 3.1. Logo após são apresentados os algoritmos para ambientes tolerantes a falha do tipo crash em 3.2 e, por último, para ambientes tolerantes a falha do tipo bizantino em 3.3.

3.1 Algoritmos de Multicast Atômico - sem tolerância a falhas

Dois algoritmos sem tolerância a falhas merecem uma atenção especial não apenas do ponto de vista histórico, mas do ponto de vista experimental, uma vez que, comparamos os resultados destes algoritmos com o DCC.

3.1.1 Skeen

Este algoritmo pressupõe a possibilidade de envio de mensagens de todos para todos processos, e implementa as propriedades de ordem do multicast através do uso de selos temporais (chamaremos *timestamps*). Cada processo mantém um relógio lógico. Este avança quando seu processo propõe um *timestamp* a uma mensagem, ou quando um *timestamp* final é associado a uma mensagem. Os *timestamps* são únicos, usando, por exemplo, o identificador do processo como uma parte do mesmo para desambiguá-los.

Ao fazer $multicast(m)$, m é enviada a todos destinos em $m.dst$. Cada um destes então propõe à mensagem um *timestamp* local e envia m com este *timestamp* a todos processos em $m.dst$. Quando um processo recebeu todos os *timestamps* de uma mensagem, ele computa o *timestamp* final como o mais alto dos *timestamps* propostos. Dado que todos tem os mesmos *timestamps* e a função de escolha é determinística, todos associam o mesmo *timestamp* a m. Uma mensagem m pode ser entregue por um processo se todas mensagens com *timestamp* proposto menor que o *timestamp* final de m tenham já sido entregues ou recebido um *timestamp* final maior que o de m. Este algoritmo 3.1 usa dois passos de comunicação. Para ordenar m, são necessárias $|m.dst|^2$ mensagens. Pode existir um efeito de enfileiramento: quando um processo propõe um timestamp t a uma mensagem m, ele deve antes resolver todas as mensagens para as quais propôs timestamps menores que t. Diferentes mensagens envolvem diferentes processos, com diferentes atrasos de comunicação para a resolução de timestamps.

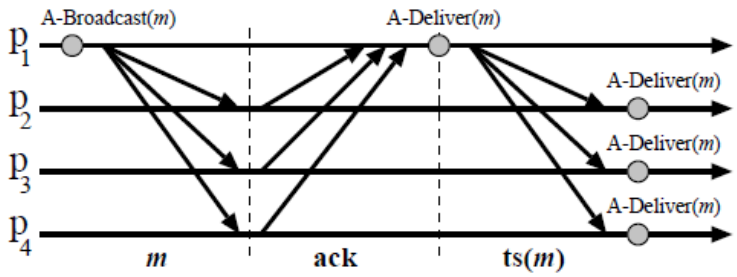


Figura 3.1 – Diagrama de tempo do protocolo Skeen, retirado de [23].

O descrito acima dá conta da ordenação de mensagens enviadas a diferentes conjuntos de processos, mas não da tolerância a falhas. Uma técnica para tornar o protocolo Skeen tolerante a falhas é fazer com que cada processo seja implementado por um grupo de réplicas coordenados por um protocolo de replicação como o Paxos. Cada processo do Skeen torna-se tolerante a falhas, como uma caixa preta. O protocolo Skeen em si permanece inalterado, realizando suas duas ações principais (computar o timestamp local e avançar o relógio lógico com base no timestamp final) da mesma forma.

1: **Definições**
2: $clock \leftarrow 0$ *{relógios lógicos de p}*
3: $local[] \leftarrow \emptyset$ *{map de mensagens com timestamp local em p }*
4: $final[] \leftarrow \emptyset$ *{map de mensagens com timestamp final em p}*
5: $del \leftarrow \emptyset$ *{ set de mensagens entregues}*
6: **multicast** m:
7: $send(START, m)$ *to* $m.dst$
8: **when** receive (START, m):
9: $clock \leftarrow clock + 1$
10: $local[m] \leftarrow (clock, P_x)$
11: $send(LOCAL\text{-}TS, m, local[m])$ *to* $m.dst$
12: **when** receive (LOCAL-TS, m, ts) from all partitions m.dst:
13: $final[m] \leftarrow$ *maximum ts receive for* m
14: $clock \leftarrow max(clock, final[m])$
15: $tryDeliver()$
16: **tryDeliver():**
17: **for each** $m \in final \setminus del$: *in* $final[m]$ *order* **do**
18: **if** $\forall m' \in local \setminus del : (m' \in final \wedge final[m] < final[m']) \vee (final[m] < local[m'])$ **then**
19: $del \leftarrow del \cup m$
20: $deliver(m)$

Algoritmo 3.1 – Skeen - retirado de [19]

3.1.2 Delporte e Fauconnier (D&F)

Este algoritmo 3.2, apresentado em [10] e chamado aqui de D&F, assume uma ordem total entre os processos, onde um processo de ordem menor pode enviar mensagens para processos de ordem maior - exceto uma mensagem END enviada pelo último processo a todos destinatários anteriores.

Para enviar uma mensagem m para um conjunto de processos $m.dst$, escolhe-se o processo mais baixo[1] na ordem e envia-se m a ele. Ao receber m, um processo repassa ao próximo processo maior em $m.dst$ conforme, e aguarda uma mensagem END. Este comportamento segue recursivamente pela ordem de processos destinatários até atingir o último, que então manda uma mensagem END a todos os destinatários anteriores. Neste ponto, cada destinatário estava aguardando a mensagem END e ao recebê-la passa tratar a próxima mensagem. O algoritmo 3.2 descreve de forma sintética esse funcionamento .

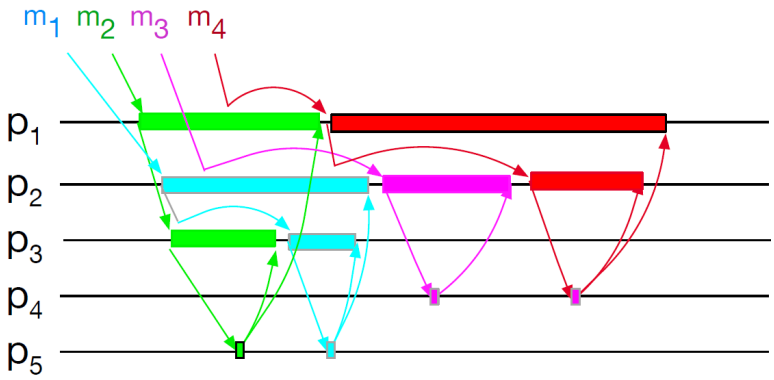


Figura 3.2 – Troca de mensagens usando D&F. P_1 tem a menor ordem e P_5 a maior.

Devido à espera por END em cada processo destinatário, este protocolo garante as propriedades de ordem do multicast atômico. Considerando a mensagem recebida no nodo mais baixo na ordem (ingresso na topologia), ele tem número de passos e número de total de mensagens necessárias para completar a entrega respectivamente proporcionais a $|m.dst|$ e $2 \times (|m.dst| - 1)$. Outra característica é que a sincronização imposta na espera por END pode gerar enfileiramento de mensagens a tratar pelos processos (efeito comboio). A Figura 3.2 exemplifica este efeito quando quatro mensagens concorrentes são encaminhadas a sub-conjuntos de processos sobrepostos.

[1] Usa-se também 'mais baixo'/'mais alto' para denotar um processo de menor / maior ordem em relação a outro.

```
1: multicast m:
2:    send(m) to min(m.dst)
3: when receive (m) from q :
4:    deliver(m)
5:    if p = max(m.dst) then
6:       send(m.END) to m.dst − p
7:    else
8:       send(m) to next(m.dst)
9:       wait until receive(m.END)
```

Algoritmo 3.2 – D&F - retirado de [10]

3.2 Algoritmos de Multicast Atômico - tolerantes a falhas crash

Uma abordagem comum para construção de algoritmos de multicast tolerantes a falhas é fazer com que cada processo seja tolerante a falhas. Cada processo então torna-se um sistema replicado que pode tolerar um determinado tipo de falha de interesse - a este sistema chamamos aqui de nodo. A tolerância a falhas de cada nodo é implementada com um algoritmo de broadcast atômico (ou consenso) entre um grupo de réplicas.

A disseminação das mensagens entre estes nodos é realizada de forma a respeitar as propriedades do multicast atômico. Uma das primeiras utilizações desta abordagem foi com o algoritmo de Skeen [22], relatado a seguir. Outros casos são relatados na sequência. Esta abordagem é importante também por ter sido utilizada no algoritmo Byzcast [6], o primeiro algoritmo de multicast atômico tolerante a falhas bizantinas.

3.2.1 White-Box

Conforme posto em 3.1.1, o clássico algoritmo Skeen pode ser reescrito de tal maneira a tolerar falhas, para isso, basta que se mantenha a cerne do processo de disseminação inalterado (computar o timestamp local e avançar o vetor de relógio a partir do timestamp global) e se utilize de uma abordagem de caixa preta na replicação (Paxos). Todavia, ao se combinar os dois algoritmos sem qualquer tipo de refinamento, isso acarretaria em no mínimo mais duas fases no protocolo, trazendo maior latência ao processo de disseminação [12]. Para contornar a situação, o termo White-Box (WBCast) surge em decorrência das otimizações no protocolo para torná-lo mais eficiente em virtude desta combinação.

A ideia central do protocolo White-Box para alcançar a tolerância a falhas e, ao mesmo tempo, reduzir latência é simples: não enviar timestamps locais para os líderes de cada grupo diretamente, mas encaminhá-los através de um quorum de processos em cada

grupo destino para garantir a durabilidade [12]. Vale lembrar que o White-Box segue o modelo de replicação passiva, onde o processo líder de cada grupo computa o timestamp e decide quando entregar a mensagem. Os seguidores (followers) seguem a decisão do líder [12].

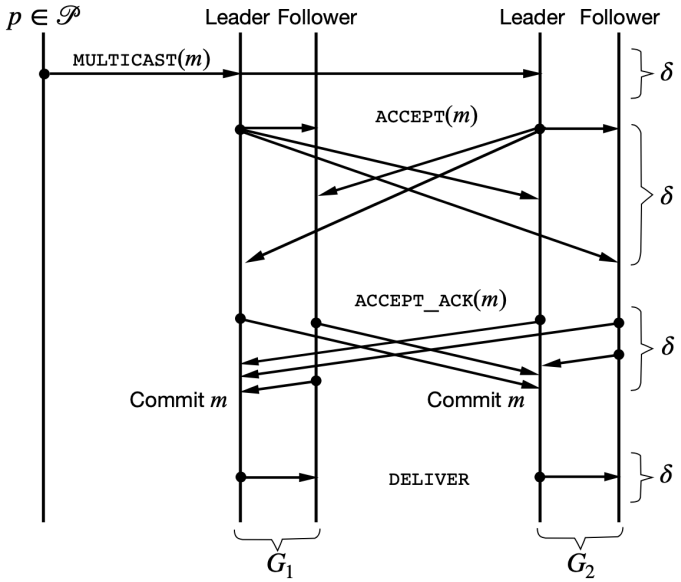


Figura 3.3 – Diagrama de tempo do protocolo White-Box, retirado de [10].

O diagrama de tempo da Figura 3.3 sintetiza o funcionamento do protocolo White-Box. Na medida que o cliente faz multicast de uma mensagem para os líderes de cada grupo *G*1 e *G*2, cada líder calcula o timestamp local (baseado no seu relógio) e encaminha uma mensagem de *ACCEPT* para todos os processos destino de cada grupo (inclusive os líderes). Todos os processos dos grupos destino aceitam a mensagem, calculam o maior timestamp local e devolvem para o líder uma mensagem de reconhecimento *ACCEPT_ACK*. Uma vez formado um quorum de mensagens (2f+1) *ACCEPT_ACK* para cada grupo destino, o timestamp global é calculado no líder (a partir do valor máximo de cada timestamp local do grupo) e, a mensagem é dada como comprometida [2](*COMMITTED*).

O líder entrega uma ou mais mensagens comprometidas como o algoritmo Skeen, ou seja, tendo como critério a ordem do timestamp global (ascendente). No final do protocolo, o processo líder envia as informações sobre a entrega de cada mensagem, a partir de uma mensagem de *DELIVER*, para todos os membros do grupo. Cada membro recebe a mensagem *DELIVER*, atualiza seu relógio, seu timestamp local e global, para então, entregar a mensagem de dados para a aplicação.

[2]A palavra comprometida faz menção a resolvida ou efetivada.

3.2.2 RamCast

Um exemplo de algoritmo de multicast atômico contrastante ao paradigma corrente de troca de mensagens é o RamCast [17]. O RamCast é o primeiro protocolo de multicast atômico para o modelo de memória compartilhada que se utiliza de RDMA (*Remote Direct Memory Access*) para a transmissão. A tecnologia RDMA estende as primitivas de comunicação *send* e *receive* com operações de leitura e escrita na memória compartilhada. Na sua essência, o RDMA permite que um nodo possa ler e escrever na memória de outro nodo sem que essa operação passe pelo processador ou a pilha de rede do sistema operacional. Como não poderia ser diferente, os ganhos de performance são consideráveis, todavia requer um rearranjo levando-se em consideração não apenas as idiossincrasias da arquitetura mas, principalmente, características do protocolo RDMA na transmissão.

O protocolo RamCast é construído levando-se em consideração dois aspectos fundamentais: O algoritmo Skeen e o protocolo de consenso Paxos, contudo, operando com um modelo de memória protegido. Com relação ao Skeen, o RamCast se comporta de forma similar ao algoritmo visto anteriormente na seção 3.1.1. Isto é, o processo destino da mensagem multicast primeiro define um possível valor de timestamp para a mensagem e, eventualmente, entra em um acordo com o valor final do timestamp. Já o processo que entrega a mensagem se pauta pelo timestamp final da mensagem para ordenar corretamente a entrega. No Paxos sem a memória protegida (tradicional) para ordenar a mensagem m, o líder propõe m em uma instância de consenso. Em situações normais, os seguidores aceitam o valor proposto e respondem para o líder. Ao passo que, no Paxos com memória protegida, os seguidores devem permitir o acesso exclusivo de escrita da sua memória ao líder, de tal sorte que, ele possa propor um valor m (via RDMA) na memória dos seguidores formando um quorum. Em caso de falha do processo líder, um novo líder tomará o seu lugar, revogando as permissões de escrita do líder anterior.

Em resumo, a execução normal do Ramcast possui o seguinte regramento. Um cliente escreve uma mensagem multicast m na memória de todos os processos destino. O líder de cada grupo de destino propõe e escreve um timestamp para a mensagem m na memória dos seguidores e para os outros líderes. Os líderes propagam o timestamp escrito por outros líderes e os seguidores reconhecem o timestamp proposto. A mensagem m é entregue de forma ordenada.

3.2.3 ft-abcast

Ft-abcast é um algoritmo de multicast atômico tolerante a falhas do tipo *crash-recovery* capaz de garantir a ordem total global na entrega das mensagens [10]. Seguindo os mesmos princípios norteadores dos protocolos de multicast atômico vistos em 2.6 e 2.7, a maior contribuição deste algoritmo dita sobre como foi equacionado o problema da ordem total global. Para tal, uma ideia metódica foi proposta que faz com que cada grupo envolvido no multicast atômico espere até que o último grupo entregue a mensagem *m*. De forma mais precisa, o primeiro grupo decide a ordem da entrega da mensagem *m*, então ele transmite a mesma mensagem para o próximo grupo de destino e espera uma mensagem de controle END [10]. Os demais grupos procedem da mesma maneira, ou seja, eles decidem a ordem da mensagem *m* e a encaminham para o próximo grupo de forma sistemática, até que recebam uma mensagem de controle END, vinda do último grupo [10]. Do ponto de vista do último grupo multicast de destino, depois de decidir a ordem na entrega da mensagem *m*, o grupo deverá enviar uma mensagem de controle para todos os demais grupos sinalizando que os mesmos deverão cessar a espera e continuar o processo de multicast atômico para a próxima mensagem (*m'*). Precisamente esse é o mecanismo que impede a mensagem *m'* ser processada antes que todos os grupos multicast tenham entregue de forma ordenada a mensagem *m*, deste modo, evitando a possibilidade de ciclos.

No ft-abcast, não diferente dos demais algoritmos pertencentes a mesma categoria de multicast atômico, cada grupo de processos executa uma instância de consenso que garante, entre outras coisas, o estado de máquina replicado entre todos os processos do mesmo grupo. Então, cada processo *p* dentro de um grupo *g* deve concordar com a ordem de entrega de cada mensagem multicast *m* para o grupo. O algoritmo 3.3 descreve de forma sintética esse funcionamento.

3.3 Algoritmos de Multicast Atômico - tolerantes a falhas bizantinas

O desenvolvimento de algoritmos de multicast atômico tolerantes a falhas bizantinas foi pouco explorado na literatura.O ByzCast foi o primeiro algoritmo de multicast atômico tolerante a falhas bizantinas [6] capaz de propor uma solução para o problema. Em linhas gerais, segue o mesmo estratagema dos anteriores contudo com uma abordagem bizantina.

```
1:  Inicialization
2:     Rec ← ∅                                   {set of received messages }
3:     Del ← ∅                                   {set of delivered messages }
4:  when multicast m:
5:     send(m) to min(m.dst)
6:  when receive (m) from the first time
7:     Rec ← < Rec; m >
8:     if min(m.dst) = g(p) then
9:        send(m) to g(p)
10: when True
11:    Rec ← Rec − Del
12:    if Rec ≠ ∅ then
13:       m0 ← first(Rec)                        {choose a message to deliver }
14:       propose(m0, g(p))                      {consensus }
15:       decide(m, g(p))
16:       GA − deliver(m)
17:       Del ← Del ∪ (m)
18:       if g(p) = max(m.dst) then
19:          send(m.END) to m.dst − g(p)
20:       else
21:          send(m) to next(m.dst, g(p))
22:          wait until receive(m.END)
```

Algoritmo 3.3 – ft-abcast - retirado de [10]

3.3.1 ByzCast

O protocolo ByzCast foi o primeiro protocolo surgido na academia a preencher o vácuo de um multicast atômico tolerante a falhas bizantinas (BFT). Sua aplicabilidade é bastante ampla, podendo ser implementado em toda e qualquer aplicação que necessite de requisitos de multicast atômico em ambientes hostis ou não confiáveis. Blockchain é um bom exemplo, onde os dados podem ser particionados (shards) entre vários grupos de processos.

As propriedades do Byzcast são as mesmas dos protocolos de multicast atômico vistas em 2.6, a saber: Validade, Integridade, Acordo, Ordem Total Local e Global.

De acordo com [6], o ByzCast tem duas premissas fundamentais: Primeiro, a reutilização do arcabouço de software já disponível para trabalhar com o cenário bizantino (BFT-SMaRt). Segundo, a percepção que os protocolos de multicast atômicos devem prover um desempenho escalável. Assim, a busca por um algoritmo genuíno, isto é, que satisfaça a propriedade de minimalidade, deve ser sempre perseguida. Nesse contexto, o protocolo ByzCast pode ser classificado como parcialmente genuíno, na medida que, em situações triviais onde as mensagens são transmitidas para um único grupo de processos,

apenas a coordenação entre o remetente e o grupo de destino faz-se necessária. Todavia, quando se necessita de mensagens endereçadas a vários grupos de processos, diferentemente de antes, o envolvimento de processos que não fazem parte do destino pode se fazer necessário [6]. Precisamente esta característica torna o protocolo parcialmente genuíno.

Para garantir a ordem total global, a disseminação no ByzCast dá-se através de uma árvore lógica cujos nodos são destinos do multicast. Cada nodo desta árvore é implementado como um grupo de processos e, em cada grupo, uma instância de difusão atômica é executada [6]. Esta difusão atômica segue ordem FIFO. **Ordem FIFO**: Se um processo correto *p* faz difusão de uma mensagem *m* antes da difusão de *m'*, nenhum processo correto entrega *m'* antes de entregar *m*.

Para endereçar um único nodo (ou grupo), basta que se utilize a difusão atômica operando em modo bizantino de forma local. Ao passo que, se se necessita endereçar mais de um grupo, primeiro, é preciso identificar o grupo ancestral comum mais baixo (*Lower Common Ancestor* ou LCA) entre os grupos distintos de destino. O cliente envia a mensagem ao LCA, este ordena entre seus processos (do nodo LCA), que então repassam aos descendentes adequados conforme os demais nodos destinatários da mensagem.

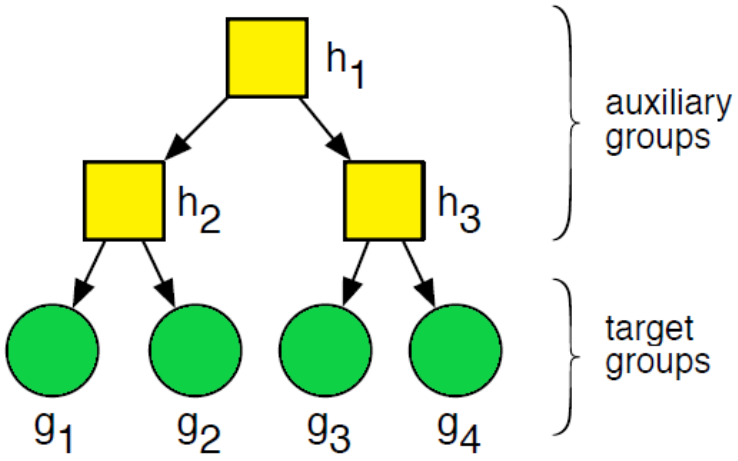


Figura 3.4 – Uma possível topologia do ByzCast, extraído de [6].

A figura 3.4 apresenta uma topologia viável do ByzCast contendo quatro grupos de destino multicast e três grupos auxiliares. A figura 3.5 amplia este conceito e procura demonstrar qual seria o caminho percorrido pelas mensagens para determinados cenários, tendo como referência a topologia da figura 3.4. Por exemplo, dada as mensagens *m1*, *m2* e *m3* transmitidas por multicast atômico para os grupos *{g1, g2}*, *{g2,g3}* e *{g3}*, respectivamente. O leitor atento observará que se tratam de cenários onde as propriedades de ordem, vistas há pouco, aparecem com necessidades diferentes. Assim, para a mensagem *m3*, por se tratar de um único grupo, uma difusão atômica para o grupo *g3* é realizada com a devida a ordem total local na entrega. Por outro lado, na mensagem *m1* endereçada aos grupos *{g1, g2}*, surge a necessidade de um grupo auxiliar *h2* atuando

como LCA, capaz de realizar a difusão atômica para os grupos de destino. A ordem desta mensagem se caracteriza por um modelo de ordem total global na entrega. Por fim, ao se analisar a mensagem *m2* endereçada aos grupos *{g2, g3}*, constatar-se-a que, o *h1* é o LCA dos grupos multicast destino. Depois da raiz, a mensagem percorre os grupos auxiliares *{h2, h3}* até chegaram aos grupos destino *{g2, g3}*, também por difusão atômica e com os mesmos critérios de ordem vistos em *m1*.

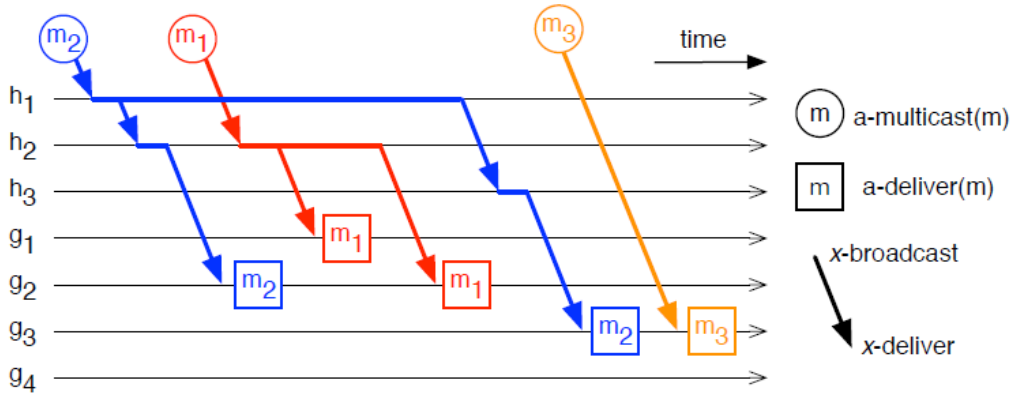


Figura 3.5 – Tráfego multicast contendo três mensagens levado de [6].

A principal invariante do protocolo ByzCast reside na maneira como os dados são dispostos nesta topologia hierárquica, portanto a ordem é mantida na medida que os grupos inferiores da árvore preservam esta ordem, naturalmente induzida pelos grupos superiores. Ou seja, se *m* é ordenado antes de *m'* em um grupo auxiliar *h*, então *m* é ordenado antes de *m'* em qualquer outro grupo que ordene mensagens [6]. Por definição, o fator que torna o protocolo genuíno de forma parcial é o mesmo que permite a ordem total global das mensagens de forma simplificada.

A essência do algoritmo do ByzCast 3.4 pode ser interpretada da seguinte maneira. Para uma mensagem *m* ser multicast para um conjunto de grupos *m.dst*, primeiro um processo realiza uma difusão atômica de *m* para o LCA (a raiz em alguns casos) de *m.dst*. Quando os processos deste grupo auxiliar entregarem a mensagem *m*, a partir de um consenso bizantino, cada processo fará uma nova difusão atômica. Essa disseminação, necessariamente, deverá levar em consideração se o novo grupo de destino pertence ou é caminho da mensagem *m.dst*. Se for o destino, por exemplo, a mensagem *m* será entregue normalmente por multicast atômico com ordem total global. Caso contrário, uma nova disseminação deverá acontecer, levando-se em consideração f+1 mensagens não bizantinas (idênticas) entregues anteriormente.

Do ponto de vista da implementação, o BFT-SMaRt tem protagonismo preponderante no ByzCast. O BFT-SMaRT, de forma conceitual, é uma biblioteca de replicação para BFT que se utiliza de uma variação do Paxos Bizantino [2]. No ByzCast cada grupo distinto corresponde a um máquina de estado replicado tolerante a falhas bizantinas [6]. Deste modo, quando um cliente envia uma mensagem *m*, o líder propõe a mensagem *m*

1: Initialization
2: $\mathcal{T}$ is an overlay tree with groups $\Gamma \cup \Lambda$
3: *A-delivered* $\leftarrow \emptyset$
4: To a-multicast message m:
5: $x_0 \leftarrow lca(m.dst)$ *{lowest common ancestor of m.dst}*
6: x_0-broadcast(m)
7: Each server process p in group x_k executes as follows:
8: **when** x_k-deliver(m)
9: **if** $k = 0$ **or** x_k-delivered m $(f + 1)$ times **then**
10: **for each** $x_{k+1} \in children(x_k)$ such that
$m.dst \cap reach(x_{k+1}) \neq \emptyset$ **do**
11: x_{k+1}-broadcast(m)
12: **if** $x_k \in m.dst$ **and** $m \notin$ *A-delivered* **then**
13: *a-deliver*(m)
14: *A-delivered* $\leftarrow$ *A-delivered* $\cup \{m\}$

Algoritmo 3.4 – ByzCast - retirado de [6]

às réplicas. As réplicas validam esta proposta escrevendo a mesma proposta em outras réplicas. As réplicas então aceitam esta proposta, se e somente se, um quorum bizantino confirmar que a mensagem é a mesma. No final, as replicas aprendem esta proposta e devolvem ao cliente a mensagem m. O algoritmo é similar ao PBFT, contendo mecanismos de eleição de um novo líder em caso de falha, um processo recuperação de uma réplica e uma reconfiguração de grupo caso seja necessário[5].

4. DCC - DAISYCHAINCAST

Neste capítulo se apresenta o núcleo da dissertação, discorre-se primeiramente a proposta do protocolo, algumas estruturas e definições gerais de funcionamento, o algoritmo propriamente descriminado e, a necessária prova de corretude.

4.1 Proposta do Protocolo

A construção de algoritmos de multicast atômico em redes parcialmente síncronas não é uma tarefa trivial e aqui não foi diferente. A ideia inicial do algoritmo DCC nasce a partir dos princípios que fundamentam o algoritmo de D&F. Sendo assim o protocolo DCC assume uma ordem total entre os processos e uma topologia na forma de um GAD para repasse das mensagens enviadas, onde um processo pode repassar mensagens a quaisquer processos mais altos nesta ordem[1]. A comunicação direta entre dois processos é assumida FIFO, através de canais perfeitos. Conforme exposto, com o intuito de evitar o bloqueio de nodos, diferentemente de D&F, DCC não usa uma mensagem de retorno para sincronizar os processos destinatários de uma mensagem. Sem as garantias de ordenação advindas desta sincronização, temos efeitos indesejados a tratar.

Conforme o algoritmo básico D&F, quando um nodo recebe uma mensagem (e não é o último destinatário), ele repassa ao próximo nodo destinatário, na ordem entre os nodos. Como no DCC o nodo não fica bloqueado esperando a entrega final da mensagem, ele pode imediatamente repassar outra mensagem com outro próximo nodo destinatário - veja a Figura 4.1(a). Entretanto conforme velocidade de encaminhamento, um ciclo pode acontecer - veja a Figura 4.1(b). Se não temos mais o bloqueio proposto em D&F, devemos resolver este problema com algum mecanismo que permita a p_4 ordenar m_2 depois de m_1, assim concordando com a ordem atribuída em p_2 - veja a Figura 4.1(c). Neste caso específico, a mensagem m_2, ao passar em p_2 passará a registrar que m_1 a antecede, permitindo que p_4 corrija a ordenação.

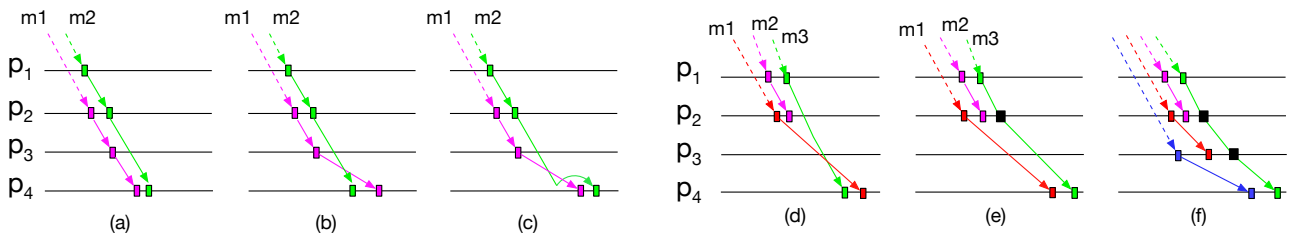


Figura 4.1 – Anomalias na ordenação.

[1]Como os processos tem uma ordem e temos que nos referir relativamente aos mesmos, usaremos expressões: menores, mais baixos e maiores, mais altos, aplicadas aos processos, relativamente a esta ordem.

Um outro caso se apresenta na Figura 4.1(d). Mesmo que, em p_1, m_3 registre ser posterior a m_2, ao chegar em p_4 não há informação para ordenar m_3 depois de m_1. Isto se deve ao desconhecimento de que m_1 antecede m_2 em p_2. Esta dependência transitiva só ocorre devido a p_1 ter enviado uma mensagem a p_2. Assim, a forma proposta para tratar este caso é fazer com que p_1, devido à existência de m_2 (encaminhada por ele), encaminhe m_3 a p_2. O tratamento de m_3 em p_2, representado na Figura 4.1(e) por um evento preto, após m_2 garante que os nodos mais altos, de p_3 em diante, respeitarão a ordem $m_1 \prec m_3$. Complementando, o mecanismo pode ter que ser aplicado recursivamente, como em Figura 4.1(f).

Note que se um nodo repassa mensagens sucessivas aos mesmos destinatários, não há necessidade de incluir nodos intermediários para garantir a ordem pois a dependência de uma outra mensagem só se estabelece por transitividade quando uma mensagem é enviada a algum destinatário diferente. Ou seja, caso exista uma mensagem m_4 com mesmos destinatários de m_3, imediatamente posterior a m_3, na Figura 4.1(e) ou (f), bastaria a p_1 enviar a mesma para o nodo p_4 já que este garantirá a entrega depois de m_3.

4.2 Estruturas e Definições Gerais

Topologia. De forma mais detalhada, a topologia assumida é obtida tomando-se o conjunto D e atribuindo uma ordem total a seus elementos. Chamamos esta ordem $>$ e se para $d_i, d_j \in D$, $(d_i, d_j) \in >$ então d_i tem ordem maior que d_j. Assumimos esta ordem sendo: total (ou seja, dado um elemento d_i, para todo outro elemento d_j temos que $d_i > d_j$ ou $d_j > d_i$), transitiva, irreflexiva e assimétrica. Estabelecemos que o conjunto E de arestas (dirigidas) do GAD como $\{(v, w) \cdot v < w\}$. Ou seja, para todo par de vértices v e w, se $v < w$ então a aresta dirigida $(v, w) \in E$, resultando em $[|D| \times (|D| - 1)/2]$ arestas. Estas definições estão no Algoritmo 4.1, a partir da linha 1. Cada aresta representa um canal direcional, FIFO, perfeito de comunicação. Uma mensagem m endereçada a qualquer subconjunto $m.dst$ de D entra na estrutura de disseminação pelo processo em $m.dst$ de menor ordem e segue pelas arestas que levam aos próximos processos de maior ordem.

Relógios Lógicos. Como discutido acima, em nosso algoritmo, propomos o uso de informações de causalidade para auxiliar na ordenação de mensagens. Ou seja, quando um processo l encaminha uma mensagem m para um processo superior h, h deve saber quais mensagens precedem m na perspectiva de l, de modo que, ao entregar mensagens, respeite a ordem de l. A causalidade pode ser rastreada com diferentes mecanismos, como históricos de mensagens ou com relógios lógicos. No DCC evitamos lidar com estruturas crescentes, como históricos por exemplo. Ao invés disso, preferimos uma abordagem mais direta através do uso de relógios vetoriais. No entanto, o uso de relógios vetoriais, como visto em algoritmos típicos de transmissão causal, não é suficiente, justamente por-

que aqui estamos lidando com multicast. Considere, por exemplo, na Figura 4.1(d) m_1 não existe: m_3 é a segunda mensagem de p_1 e, ao chegar em p_4, deveria ser entregue, já que a primeira mensagem de p_1 não é endereçada a p_4. Isso exigiria representar quais processos foram endereçados por quais mensagens no passado de m_3.

Visando uma proposta mais simples, observamos que, devido às restrições topológicas assumidas e aos canais unidirecionais FIFO (de processos inferiores para superiores), quando uma mensagem *m* chega ao seu processo de destino mais baixo *l*, para ser encaminhada a outros destinos, *l* pode atribuir a *m* uma ordem em relação a cada um dos processos superiores que são destino para *m*. Por exemplo, na Figura 4.1(c), p_2 atribui a m_1 as seguintes ordens em relação aos destinos de m_1 $[p_2 - p_3 : 1\ , p_2 - p_4 : 1]$. Da mesma forma, p_1 atribui a m_2 as seguintes ordens:$[p_1 - p_2 : 1\ , p_1 - p_4 : 1]$. Quando m_2 é tratado em p_2, é a primeira mensagem de p_1, não há mensagens em seu passado. Ao encaminhar para p_4, m_2 carregará a informação agregada $[p_1 - p_2 : 1\ , p_1 - p_4 : 1, p_2 - p_3 : 1\ , p_2 - p_4 : 1]$. Em p_4, uma vez que a origem é p_1, $p_1 - p_4 : 1$ refere-se a m_2 em si, e o registro $p_2 - p_4 : 1$ indica que antes de m_2 existe uma mensagem endereçada a p_4. Como podemos ter tantos registros quanto arestas na topologia, chamamos o conjunto desses registros de "relógio vetorial de aresta"ou EVC. Cada nó *n* mantém um EVC registrando a informação agregada recebida de mensagens enviadas de cada nó de entrada para cada outro nó, que é atualizado de acordo com o fluxo de mensagens, visível no algoritmo 4.1, linha 8. Embora esse mecanismo registre a causalidade da mensagem a ser respeitada em nós superiores, não é suficiente para resolver a ordem acíclica, conforme discutido nas Figuras 4.1(d,e,f).

Mensagens e Processos. Uma mensagem tem um conjunto de destinatários *m.dst*. *m.src* neste contexto é o processo de ingresso no GAD overlay. Ainda, a mensagem carrega seus dados e o relógio vetorial associado. Veja Algoritmo 4.1, a partir da linha 9. Um processo *p* tem um relógio vetorial *p.EVC* que marca o acumulado de mensagens enviadas de cada nodo ingresso para cada outro nodo, que vai sendo atualizado conforme o fluxo de mensagens. Ainda, um processo tem uma estrutura para manter mensagens pendentes. Veja Algoritmo 4.1, a partir da linha 15. Assume-se comunicação sem falha.

Mensagens Fast e Slow. Conforme explicado anteriormente, em algumas situações nosso algoritmo necessita encaminhar uma mensagem através de um nodo intermediário, não destinatário da mensagem, mas envolvido em comunicação anterior com algum destinatário da mensagem. Chamamos de *Slow* mensagens que passam por nodos intermediários e de *Fast* mensagens que passam exclusivamente em seus destinatários. A decisão de encaminhamento de uma mensagem, se *Fast* ou *Slow* acontece em cada nodo por onde a mensagem passa avaliando-se as últimas mensagens encaminhadas.

Algoritmo 4.1 – Tipos e Estruturas DCC

1: **Topologia de Comunicação entre Processos Destino**
2: D *{Conjunto de processos destino}*
3: $> : D \times D$ *{Uma ordem total entre os processos}*
4: $E : \{(n, m) \mid n, m \in D, n < m\}$ *{Todo processo tem uma aresta para cada um de seus mais altos}*
5: $GAD = (D, E)$ *{Cada aresta (o,d) representa possibilidade de o enviar via send para d}*
6: $in(d \in D) : \{\forall\ s \in D | (s, d) \in E\}$ *{arestas de entrada de um processo destino d}*
7: **Relógio Vetorial de Arestas**
8: $EVC : E \rightarrow \mathbb{N}$ *{Um relógio vetorial de arestas associa cada aresta a um natural}*
9: **Mensagem**: toda mensagem m tem:
10: $m.EVC : EVC$ *{hora vetorial de m}*
11: $m.src \in D$ *{originador de m}*
12: $m.dst \in \mathcal{P}(D) \mid \forall p \in m.dst, m.src \leq p$ *{destino de m, qualquer subconjunto de processos}*
13: $m.mode \in \{fast, slow, \perp\}$ *{modo fast ou slow}*
14: $m.data$ *{dados da mensagem}*
15: **Processo**: todo processo p tem as variáveis:
16: $p.EVC : EVC$ *{hora vetorial de p}*
17: $p.b$ *{um buffer de mensagens}*
18: **Primitivas de Comunicação**:
19: **send**($d \in D, m : Mensagem$): processo envia m para d em FIFO
20: **receive**($m : Mensagem$): processo $p \in m.dst$ recebe m

4.3 Funcionamento

Envio em aMulticast. Para enviar em aMulticast, deve-se mandar a mensagem para o nodo mais baixo entre os destinatários, conforme Algoritmo 4.2, linha 1.

Recebimento no Processo de Ingresso. Ao receber uma mensagem externa, Algoritmo 4.2, linha 5, o processo de entrada é marcado como origem da mensagem em $m.src$ e a seguir, conforme Algoritmo 4.3, linha 28, o relógio do processo é atualizado. A mensagem é estampada com o relógio atualizado do processo, decide-se o modo de encaminhamento (*slow* ou *fast*), e a mensagem é repassada. Note que esta mensagem é imediatamente entregue, Algoritmo 4.3, linha 29 e o processo pode tratar outra mensagem. A atualização do relógio pelo processo p, por ocasião do ingresso de uma mensagem, está na linha 18 do Algoritmo 4.3, que incrementa o relógio para cada destinatário $d \in m.dst$ da mensagem, na posição representativa do par (p, d). A seguir, envia-se a mensagem ao próximo nodo apropriado, conforme o modo, linha 5 do Algoritmo 4.3. A mensagem *fast* segue ao próximo destinatário dela. A mensagem *slow* segue para o próximo nodo na ordem topológica, conforme a seguir.

Definição de Modo de Encaminhamento. Dada uma mensagem m, a definição do modo de encaminhamento de m em um processo p_i considera se uma mensagem m' enviada anteriormente por p_i tem algum destinatário p_j mais baixo que o próximo destinatário

Algoritmo 4.2 – Algoritmo Principal DCC

```
1:  Para enviar m em aMulticast, cliente executa como segue:
2:    let m ∈ Mensagem with m.mode ← ⊥, m.dst ← dst        {inicializa a mensagem}
3:    send(lowest(m.dst), m)                               {envia aMulticast para primeiro destino de m }

4:  Nodo n executa como segue:
5:    upon receive(em) and em.mode = ⊥                     {recebe mensagem externa}
6:      em.src ← n
7:      DeliverAndForward(em)
8:    upon receive(m) and m.mode ≠ ⊥ and ItsForMe(m)       {mensagem genuína fast ou slow}
9:      if CanDeliverAndForward(m) then                    { sem dependências causais }
10:       DeliverAndForward(m)
11:     else n.b.append(m)                                 {armazena em buffer}
12:   upon receive(m) and m.mode ≠ ⊥ and ¬ItsForMe(m))     {mensagem não genuína slow}
13:     if CanForward(m) then                              { sem dependências causais }
14:       AccumulateAndForward(m)
15:     else n.b.append(m)                                 {armazena em buffer}
16:   upon ∃ m ∈ n.b | ItsForMe(m) ∧ CanDeliverAndForward(m)
17:     DeliverAndForward(m)                               {mensagem no buffer para este nodo pode ser entregue}
18:   upon ∃ m ∈ n.b | ¬ItsForMe(m) ∧ CanForward(m)
19:     AccumulateAndForward(m)                            {mensagem no buffer pode ser repassada}
```

de m. Se este for o caso, pode existir m'' ordenado antes de m' em p_j, ou seja $m'' < m'$, e nesse caso deve-se garantir $m'' < m$. Para tal, faz-se com que m seja encaminhado ao processo p_j, que trata m' e m''. Como p_j encaminha m, este atualiza o relógio vetorial da mensagem para constar a existência de m'' em seu passado. Assim, garante que $m'' < m$ em qualquer destinatário comum entre elas. Como este comportamento tem recursão na estrutura de nodos, a cadeia de dependências pode crescer, evitando ciclos de diversos tamanhos. Este procedimento acaba pois a topologia formada entre processos é um GAD.

Nesta versão do algoritmo implementamos uma versão mais forte (mais restritiva) desta decisão, e por isso mais simples. Dada uma mensagem m no processo de entrada p, m é encaminhada em modo *fast* se ela tem os mesmos destinatários da última mensagem m' encaminhada por p (linha 11 do Algoritmo 4.3). Senão é encaminhada no modo *slow*.

Recebimento de Mensagens internas ao GAD em processos destinatários, e seu repasse. Ao receber uma mensagem externa, conforme Algoritmo 4.2, linha 8, um processo destinatário verifica se ela pode ser entregue. Conforme Algoritmo 4.3, linha 36, isto significa que todas as mensagens anteriores a ela, destinadas a este processo, marcadas no seu relógio vetorial, devem ter sido entregues no processo. Ainda, esta deve ser a próxima mensagem a ser entregue vinda de $m.src$. Caso a mensagem não possa ser entregue, ela entra em um buffer de pendentes. Em caso de entrega habilitada a mensagem

é entregue e repassada a próximos destinatários, Algoritmo 4.3, linha 28. Neste repasse, é feita a atualização do relógio do processo, Algoritmo 4.3, linha 15, que neste caso guarda no relógio do processo o máximo entre o seu valor e o da mensagem entregue.

Recebimento de Mensagens internas ao GAD em processos intermediários, e seu repasse. O caso análogo para processos intermediários é dado em Algoritmo 4.2, linha 12. Neste caso não existe mudança no relógio do processo. Somente o relógio da mensagem é atualizado, Algoritmo 4.3, linha 24. Isto garante que em próximos processos destinatários, as mensagens tratadas neste processo intermediário, até o ponto deste repasse, serão também consideradas anteriores à mensagem sendo repassada.

Tratamento de Pendentes. Conforme as duas últimas guardas do Algoritmo 4.3, a qualquer momento uma mensagem pendente pode se tornar apta para entrega, devido à atualização do relógio do processo. Em tais casos o tratamento é idêntico aos correspondentes anteriores.

4.4 Discussão de Corretude

O algoritmo proposto deve satisfazer as propriedades do multicast atômico. Como assume-se que não há falhas, Validade, Acordo e Integridade derivam da proposição 1 abaixo. A ordem global acíclica deriva das proposições 2 e 3.

Proposição 1. Se um processo faz multicast da mensagem m, m chega a todos processos em $m.dst$ uma única vez.

Argumento 1. m chega a todo destinatário: m ingressa pelo processo mais baixo, l em $m.dst$. A partir de l, m é encaminhada para o próximo processo d em $m.dst$ ou ao próximo processo da topologia px. No segundo caso, px é anterior a d, px repete o tratamento sendo que indutivamente m chega em d. Em d novamente a mensagem pode ser enviada a um d' em $m.dst$ ou a um px' anterior a d'. Como existe uma ordem total entre os processos, toda mensagem chega a todos processos destinatários;

Argumento 2. m chega uma vez: pois assume-se canais confiáveis FIFO.

Proposição 2. Sejam duas mensagens m e m' tal que $|m.dst \cap m'.dst| > 1$, ou seja, há vários destinatários comuns a ambas mensagens. Então todo processo $d \in m.dst \cap m'.dst$ entrega m e m' na mesma ordem.

Argumento: como existe uma ordenação topológica entre todos processos, então assuma que $p \in m.dst \cap m'.dst$ é o processo mais baixo da ordem entre todos processos em $m.dst \cap m'.dst$. Chamamos p de $lcd(m, m')$ (lowest common destination - destinatário comum mais baixo). Pela definição da topologia, $lcd(m, m')$ é único. Este estabelece a ordem entre m e m'. Se p ordena m antes de m', ao tratar m' ele registra em m' a dependência de m. Um

Algoritmo 4.3 – Funções Auxiliares DCC

```
1:  n.Deliver(m : message) : delivers m.data to upper layer
2:  n.NextFastHop(dst : set of nodes) : node  return the next d > n, d ∈ dst
3:  n.NextHop() : node  return the next d > n, d ∈ D
4:  n.ItsForMe(m : message) : boolean  return n ∈ m.dst
5:  n.FastOrSlow(m : message)                          {encaminha mensagem cfe seu modo}
6:    m.EVC ← n.EVC
7:    if m.mode = true then
8:      send(m, NextFastHop(m.dst))
9:    else
10:     send(m, NextHop())
11: n.CheckLastDestination(tmp : EVC) : {fast, slow}   {tem mesmos destinos que última msg?}
12:   result ← ∀ e ∈ n.cacheEVC | n.cacheEVC[e] = tmp[e]
13:   n.cacheEVC ← tmp
14:   if result then return fast else return slow
15: n.UpdateCompareEVC(m : message) : boolean          {retorna se m deve ser fast ou slow (t/f)}
16:   let old = n.EVC
17:   if m.mode = ⊥ then                               {é uma mensagem externa entrando em n}
18:     for each d ∈ m.dst | d ≠ n do                  {para cada aresta (n,d), d um nodo destino }
19:       n.EVC[(n, d)] + +                            {incrementa o relógio na aresta (n,d)}
20:   else                                             {mensagem recebida de outro nodo do GAD}
21:     for each e ∈ n.EVC | n.EVC[e] < m.EVC[e] do    {mantém máx. entre msg e nodo}
22:       n.EVC[e] ← m.EVC[e]                          {se está aqui, canDeliver = true}
23:   return n.ChekLastDestination(n.EVC − old)        {passa EVC com diferença em cada posição}
24: n.AccumulateAndForward(m : message)
25:   for each e ∈ m.EVC do                            {cada posição do relógio de m, é o maior entre}
26:     m.EVC[e] ← MAX(m.EVC[e], n.EVC[e])             { relógio do processo e da mensagem }
27:   send(m, NextHop())                               {manda para o próximo na topologia}
28: n.DeliverAndForward(m : message)
29:   n.Deliver(m)
30:   m.mode ← UpdateCompareEVC(m)                     {atualiza EVC e define modo da msg}
31:   FastOrSlow(m)                                    {encaminha m ...}
32: n.CanForward(m : message) : boolean
33:   if ∃(s, n) ∈ E | ( s ≠ m.src and m.EVC[(s, n)] > n.EVC[(s, n)] ) then   {toda aresta (s,n) ...}
34:     return false                                   {de entrada, que não se origina em m.src, tem que ser satisfeita por n.EVC}
35:   return true                                      {n recebeu através das outras arestas todas mensagens necessárias}
36: n.CanDeliverAndForward(m : message) : boolean
37:   if (m.EVC[(m.src, n)] = n.EVC[(m.src, n)] + 1)
        and CanForward(m)) then                        {é a próxima msg a tratar de m.src e passa por CanForward}
38:     return true
39:   return false
```

destinatário que não é $lcd(m, m')$ segue a informação de dependência registrada em m'. Ou seja, se m' chegar a um processo antes de m, será postergado até que m seja tratada.

Proposição 3. Considere a relação $\prec$ no conjunto de mensagens entregues por todos destinatários definida como como: $m \prec m' \iff$ existe um processo que entrega m antes de m'. A relação $\prec$ é acíclica.

Argumento: sejam quaisquer 3 mensagens tal que $m''.dst \cap m'.dst \neq \emptyset$, $m''.dst \cap m.dst \neq \emptyset$ e $m'.dst \cap m.dst \neq \emptyset$, então uma ordem acíclica deve ser formada entre estas mensagens. Se $lcd(m'', m') = lcd(m', m) = lcd(m'', m) = p$, então pela Proposição 2 mantém-se a ordem entre os destinatários das mensagens. Considere agora que cada par de mensagens tem um lcd diferente, pois tem conjuntos de destinatários diferentes. Então, pela topologia existe uma ordem entre estes lcds. Suponha ordens estabelecidas independentemente, por diferentes lcd's: $m'' < m'$ e $m' < m$ e que $lcd(m'', m)$ é mais alto que os demais lcd's, e não estabeleceu uma ordem entre m'', m. Como $m' < m$, pelo algoritmo necessariamente m passa pelos destinatários de m', depois de m'. Como $m'' < m'$, em algum destes destinatários é registrado em m que m'' aconteceu antes. Assim, quando m chega em um processo que entrega m e m'', necessariamente este processo deve entregar m'' antes de m. Indutivamente pode-se aumentar o conjunto de mensagens envolvidas.

5. TOLERÂNCIA A FALHAS

Neste capítulo se disserta sobre as possíveis variações do protocolo DCC, agora com suporte a tolerância a falhas. Como dito anteriormente, o algoritmo DCC não é tolerante a falhas, assim como o Skeen e D&F também não o são. Isso não impediu a construção de algoritmos tolerantes a falhas do tipo colapso para o Skeen como mencionado nas Seções 3.2.2 e 3.2.1, entre outros da literatura. A mesma situação ocorreu com D&F como mencionado na Seção 3.2.3. Como o algoritmo DCC pode ser visto como uma evolução do algoritmo D&F descrito na Seção 3.2, é possível imaginar uma adaptação capaz de suprir esta necessidade, de forma análoga ao algoritmo 3.3 para falhas do tipo colapso (cft), como também para falhas bizantinas (bft).

Os algoritmos tolerantes a falhas derivados do DCC são apresentados nas seções 5.1 e 5.2. Para melhor acompanhamento, mantemos as mudanças necessárias para adoção de tolerância a falhas no DCC em sublinhado, assim é possível perceber mais facilmente o que foi alterado.

Como não foi realizada a implementação dos algoritmos tolerantes a falhas do DCC, não é possível fazer uma avaliação experimental de desempenho.

5.1 Colapso

Para as falhas do tipo colapso do nodo, descritas em 2.1.1, se apresenta uma possível adaptação do algoritmo DCC com suporte a tolerância a falhas do tipo colapso. A estratégia utilizada para tolerância a falhas no algoritmo DCC é semelhante à aplicada em 3.3.1. Isto é, um nodo do DCC pode ser visto como um grupo de processos replicados e, em cada grupo de processos, uma instância de difusão atômica é executada conforme 2.5.

Sendo assim, cada grupo da topologia utiliza de difusão atômica para garantir a entrega das mensagens em ordem em cada nodo, sendo esse o elemento estruturante do algoritmo DCC tolerante a falhas. Ou seja, dado um conjunto de processos determinísticos a difusão atômica é aquele mecanismo que mantém a consistência entre múltiplas réplicas na percepção de um serviço lógico, de tal modo que, este serviço possa ser visto como uma replicação máquina de estado, conforme visto em 2.3. A máquina de estado, por sua vez, consiste de variáveis que representam um estado e comandos que alteram essas variáveis e produzem uma saída [4]. Assim comandos consistem de programas determinísticos, tal que, a saída da máquina de estado seja determinada pelo estado inicial e a sequência de comandos executados. Portanto, a difusão atômica garante que todas as réplicas entreguem comandos a partir de clientes diferentes na mesma ordem, mantendo

desta maneira o mesmo estado entre elas [4]. A imagem 5.1 em conjunto com o referencial teórico, visto no capítulo 2, ajudam a compreender a complexidade das relações entre os protocolos envolvidos no modelo colapso.

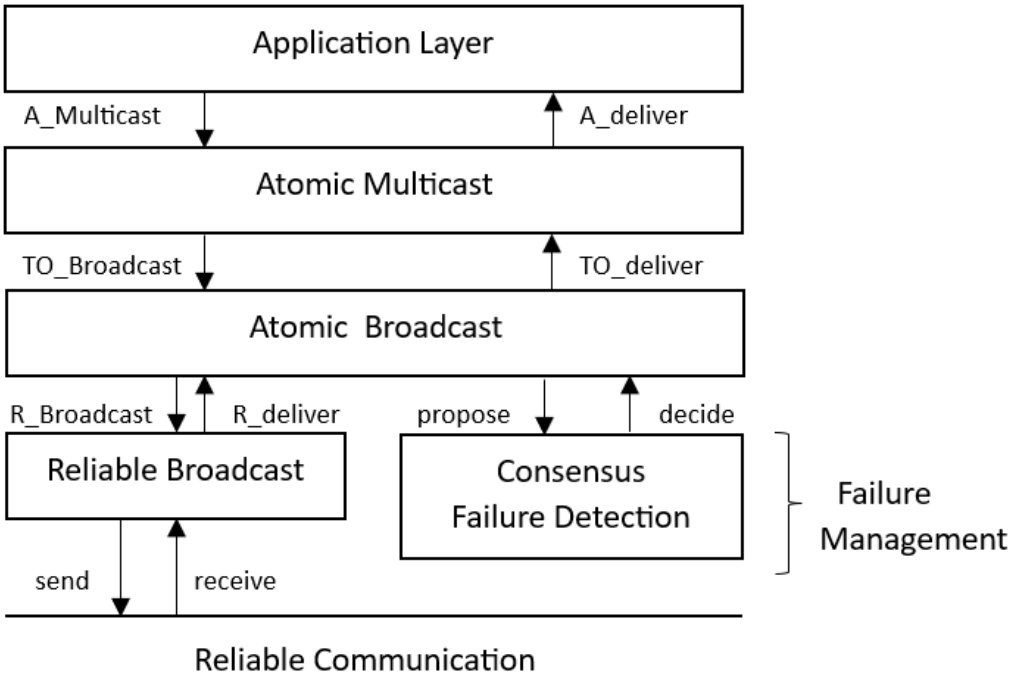


Figura 5.1 – Modelo tolerante a falhas do tipo colapso, adaptado de [15] para o Multicast Atômico.

O processo de estabelecimento de uma liderança (a partir de um grupo de processos), tendo em vista a necessidade de consenso (presente na difusão atômica), está fora do escopo deste trabalho sendo assunto amplamente discutido na literatura [4]. Para algoritmos tolerantes a falhas do tipo colapso assume-se que a maioria dos processos são corretos $f <= [(n-1)/2]$, conforme visto em [4].

Em síntese, um cliente *c* deverá encaminhar uma mensagem *m*, via difusão atômica, visto em 2.5, ao grupo *g* de processos responsável pelo primeiro destino da mensagem na topologia, conforme Algoritmo 5.2, linha 1. Todos os processos corretos do grupo *g* devem receber a mensagem *m*, logo em seguida precisam submetê-la perante um consenso uniforme cujo critério de entrega para a camada acima (DCC) será a ordem do identificador da mensagem (ordem total das mensagens). Uma vez entregue a mensagem *m* no grupo destinatário *g* (ponto de vista do DCC), ela será encaminhada por todos os processos corretos ao próximo grupo multicast também via difusão atômica, conforme Algoritmo 5.3, linha 30. Cada processo receptor do próximo grupo *h*, deverá receber apenas a primeira mensagem *m* e verificar se é um destinatário. Caso seja, o mecanismo se repete até se chegar no próximo grupo multicast. Caso contrário, a primeira mensagem *m* deverá ser tratada por todos os processos corretos do grupo *h* apenas para acumular os registros de causalidade na mensagem *m* postos nos EVCs dos processos, sendo

a mensagem encaminhada por todos os processos corretos ao próximo grupo multicast *i* via difusão atômica, conforme Algoritmo 5.3, linha 26. Por fim, se existir uma situação de transitividade não resolvida, independentemente do grupo ser destinatário ou não, a mensagem *m* deverá ser colocada em uma fila em cada processo correto do grupo, sendo processada na próxima execução. Caso existam mais destinatários, esse paradigma se repete até chegarmos no último destinatário, analogamente, ao algoritmo não tolerante a falhas do DCC. No final apenas a primeira mensagem *m* será entregue pelo cliente *c*. A imagem 5.2 demonstra de forma breve esse estratagema.

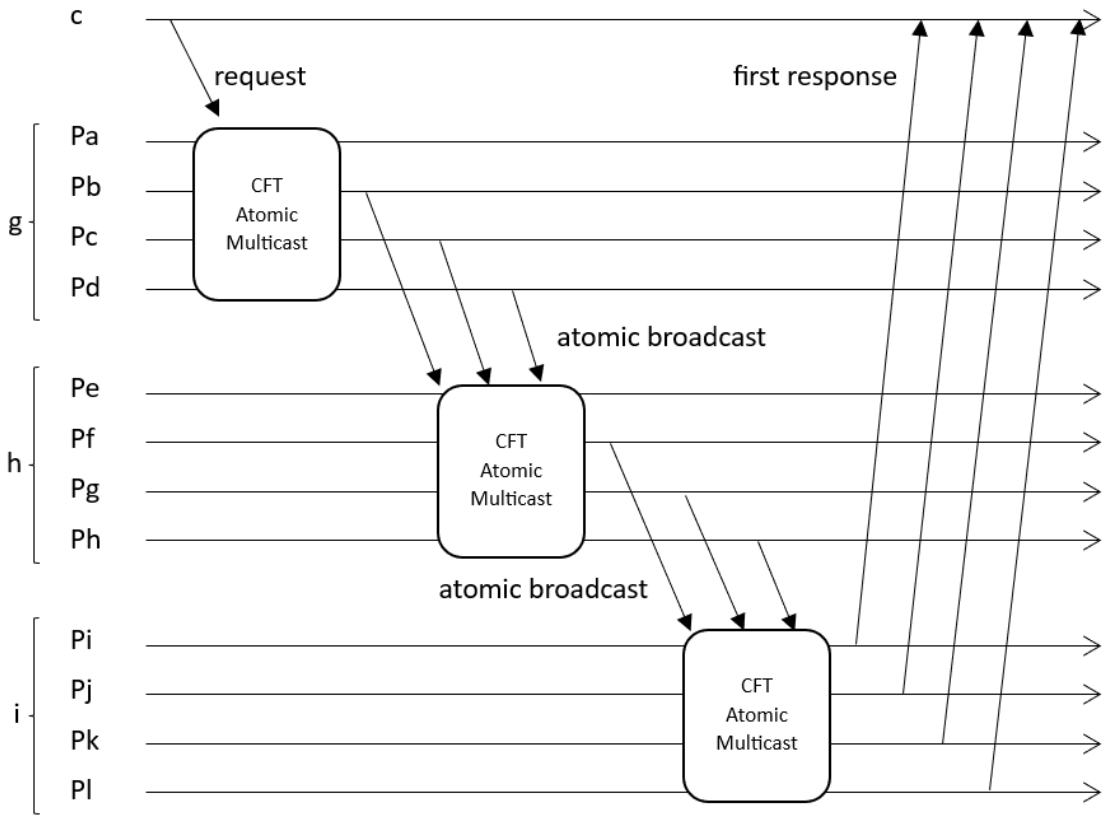


Figura 5.2 – Diagrama de tempo que demonstra o funcionamento do DCC em modo tolerante a falhas do tipo colapso com um processo em falha.

5.2 Bizantino

Para as falhas arbitrárias se demonstra uma possível variação do algoritmo DCC agora com suporte a tolerância a falhas do tipo bizantino. Essa tolerância a falhas arbitrárias no algoritmo DCC pode ser representada da seguinte maneira: Assim como no modo tolerante a falhas do tipo colapso, cada grupo da topologia utiliza da difusão atômica para garantir a entrega das mensagens em ordem em cada nodo. Diferentemente de antes, dado um cenário bizantino uma difusão atômica tolerante a falhas arbitrárias faz-se ne-

Algoritmo 5.1 – Tipos e Estruturas DCC-cft

1: **Topologia de Comunicação entre Processos Destino**
2: G *{Conjunto de grupos de processos destino}*
3: $> : G \times G$ *{Uma ordem total entre os grupos de processos}*
4: $E : \{(n, m) \mid n, m \in G, n < m\}$ *{Todo grupo de processos tem uma aresta para cada um de seus mais altos}*
5: $GAD = (G, E)$ *{Cada aresta (o,g) representa possibilidade de o enviar via broadcast para g}*
6: $in(g \in G) : \{\forall\ s \in G | (s, g) \in E\}$ *{arestas de entrada de um grupo de processos destino g}*
7: **Relógio Vetorial de Arestas**
8: $EVC : E \rightarrow \mathbb{N}$ *{Um relógio vetorial de arestas associa cada aresta a um natural}*
9: **Mensagem**: toda mensagem m tem:
10: $m.EVC : EVC$ *{hora vetorial de m}*
11: $m.src \in G$ *{originador de m}*
12: $m.dst \in \mathcal{P}(G) \mid \forall g \in m.dst, m.src \leq g$ *{destino de m, qualquer subconjunto de grupos de processos}*
13: $m.mode \in \{fast, slow, \perp\}$ *{modo fast ou slow}*
14: $m.data$ *{dados da mensagem}*
15: **Processos**: cada processo p pertencente a um grupo g tem as variáveis:
16: $p.EVC : EVC$ *{hora vetorial do grupo g}*
17: $p.b$ *{um buffer de mensagens pendentes}*
18: **Primitivas de Comunicação**:
19: **broadcast**(m : *Mensagem*, *set of nodes* $\in g$): envia m via difusão atômica para
20: cada processo do grupo g
21: **deliver**(m : *Mensagem*): cada processo do grupo $g \in m.dst$ recebe m por broadcast

cessário agora. A difusão atômica bizantina segue a mesma ordem total vista em 3.3.1. Novamente, a imagem 5.3 mais o referencial teórico, visto no capítulo 2, ajudam na compreensão da complexidade posta no modelo bizantino.

O processo de estabelecimento de uma liderança (a partir de um grupo de processos) tendo em vista a necessidade de consenso bizantino, assim como a todo o processo criptográfico envolvido, estão fora do escopo deste trabalho sendo assunto amplamente discutido na literatura [4]. Para algoritmos tolerantes a falhas do tipo bizantino assume-se que a grande maioria dos processos são corretos (não bizantino) $f <= [(n-1)/3]$, conforme visto em [4].

Para resumir, um cliente c deverá encaminhar uma mensagem m via difusão atômica bizantina, visto em 2.2, ao grupo g de processos responsável pelo primeiro destino da mensagem na topologia, conforme Algoritmo 5.5, linha 1. Todos os processos do grupo g devem receber a mensagem m, logo em seguida precisam submetê-la perante um consenso bizantino cujo critério de entrega para a camada acima (DCC) será a ordem do identificador da mensagem (ordem total das mensagens). Uma vez entregue a mensagem m (no DCC) ela será disseminada ao próximo grupo multicast h por todos os processos através de uma difusão atômica bizantina, conforme Algoritmo 5.6, linha 30. Obtendo $f + 1$ mensagens idênticas ou corretas em cada processo receptor, o próximo passo é verificar se é um destinatário. Se o grupo h for destino, o mecanismo se repete em 5.5, linha 8. Se

Algoritmo 5.2 – Algoritmo Principal DCC-cft

1: **Para enviar** *m* **em aMulticast, cliente executa como segue:**
2: let $m \in Mensagem$ with $m.mode \leftarrow \perp$, $m.dst \leftarrow dst$ *{inicializa a mensagem}*
3: $broadcast(m, processOfGroup(lowest(m.dst))$ *{envia para o primeiro grupo destino de m via difusão atômica}*

4: **Cada processo** *n* **de um grupo** *g* **executa como segue:**
5: **upon deliver**(*em*) **and** $em.mode = \perp$ *{recebe mensagem externa}*
6: $em.src \leftarrow n$
7: $DeliverAndForward(em)$
8: **upon deliver** first(m) **and** $m.mode \neq \perp$ **and** $ItsForMe(m)$ *{primeira msg genuína via difusao atômica}*
9: **if** $CanDeliverAndForward(m)$ **then** *{ sem dependencias causais}*
10: $DeliverAndForward(m)$
11: **else**
12: $n.b.append(m)$ *{armazena em buffer}*
13: **upon deliver** first(m) **and** $m.mode \neq \perp$ **and** $\neg ItsForMe(m))$ *{primeira msg não genuína via difusao atômica}*
14: **if** $CanForward(m)$ **then** *{ sem dependencias causais }*
15: $AccumulateAndForward(m)$
16: **else**
17: $n.b.append(m)$ *{armazena em buffer}*
18: **upon** $\exists\ m \in n.b \mid ItsForMe(m) \wedge CanDeliverAndForward(m)$
19: $DeliverAndForward(m)$ *{mensagem no buffer para este grupo pode ser entregue}*
20: **upon** $\exists\ m \in n.b \mid \neg ItsForMe(m) \wedge CanForward(m)$
21: $AccumulateAndForward(m)$ *{mensagem no buffer pode ser repassada}*

não, a mensagem *m* deverá ser tratada por todos os processos do grupo *h* apenas para acumular os registros de causalidade na mensagem *m*, sendo a mensagem encaminhada por difusão atômica bizantina ao próximo grupo multicast *i*, conforme Algoritmo 5.6, linha 26. A mesma situação de enfileiramento ocorre em caso da mensagem apresentar alguma dependência causal não resolvida. No final a mensagem *m* será devolvida ao cliente *c* novamente com $f + 1$ mensagens idênticas ou corretas. A imagem 5.4 demonstra de forma sucinta esse funcionamento.

5.3 Discussão de Corretude

Como em todas as técnicas de replicação máquina de estado, existe dois requisitos para as réplicas satisfazerem: elas devem operar de forma determinística (dada a mesma entrada, todas as réplicas devem se comportar igualmente) e, todas as réplicas devem começar no mesmo estado inicial [20]. Dado esses dois requisitos e, mesmo diante

Algoritmo 5.3 – Funções Auxiliares DCC-cft

```
1:  n.Deliver(m : message) : delivers m.data to upper layer
2:  n.NextFastHop(dst : set of groups) : group  return the next g > currentGroup(), g ∈ dst
3:  n.NextHop() : group  return the next g > currentGroup(), g ∈ G
4:  n.ItsForMe(m : message) : boolean  return currentGroup() ∈ m.dst
5:  n.processOfGroup(dst : group) : set of nodes  return p ∈ dst
6:  n.currentGroup() : group  return g ⊃ n
7:  n.FastOrSlow(m : message)                                   {encaminha mensagem cfe seu modo}
8:    m.EVC ← n.EVC
9:    if m.mode = true then
10:     broadcast(m, processOfGroup(NextFastHop(m.dst)))      {cada nodo envia a msg para o próximo
          grupo de destinatários}
11:   else
12:     broadcast(m, processOfGroup(NextHop()))               {cada nodo envia a msg para o próximo grupo da
          topologia}
13: n.CheckLastDestination(tmp : EVC) : boolean               {tem mesmos destinos que última msg?}
14:   result ← ∀ e ∈ n.cacheEVC | n.cacheEVC[e] = tmp[e]
15:   n.cacheEVC ← tmp
16:   return result
17: n.UpdateCompareEVC(m : message) : boolean                 {retorna se m deve ser fast ou slow (t/f)}
18:   let old = n.EVC
19:   if m.mode = ⊥ then                                      {é uma mensagem externa entrando em n}
20:     for each d ∈ m.dst | d ≠ n do                         {para cada aresta (n,d), d um nodo destino }
21:       n.EVC[(n, d)] + +                                   {incrementa o relógio na aresta (n,d)}
22:   else                                                    {mensagem recebida de outro nodo do GAD}
23:     for each e ∈ n.EVC | n.EVC[e] < m.EVC[e] do           {mantém máx. entre msg e nodo}
24:       n.EVC[e] ← m.EVC[e]                                 {se está aqui, canDeliver = true}
25:   return n.ChekLastDestination(n.EVC − old)               {passa EVC com diferença em cada posição}
26: n.AccumulateAndForward(m : message)
27:   for each e ∈ m.EVC do                                   {cada posição do relógio de m, é o maior entre}
28:     m.EVC[e] ← MAX(m.EVC[e], n.EVC[e])                    { relógio do processo e da mensagem }
29:   broadcast(m, processOfGroup(NextHop()))                 {cada nodo envia a msg para o próximo grupo da topologia}
30: n.DeliverAndForward(m : message)
31:   m.mode ← UpdateCompareEVC(m)                            {atualiza EVC e define modo da msg}
32:   n.Deliver(m)
33:   FastOrSlow(m)                                           {encaminha m ...}
34: n.CanForward(m : message) : boolean
35:   if ∃(s, n) ∈ E | ( s ≠ m.src and m.EVC[(s, n)] > n.EVC[(s, n)] ) then   {toda aresta (s,n) ...}
36:     return false                                          {de entrada, que não se origina em m.src, tem que ser satisfeita por n.EVC}
37:   return true                                             {n recebeu através das outras arestas todas mensagens necessárias}
38: n.CanDeliverAndForward(m : message) : boolean
39:   if (m.EVC[(m.src, n)] = n.EVC[(m.src, n)] + 1)
        and CanForward(m)) then                               {é a próxima msg a tratar de m.src e passa por CanForward}
40:     return true
41:   return false
```

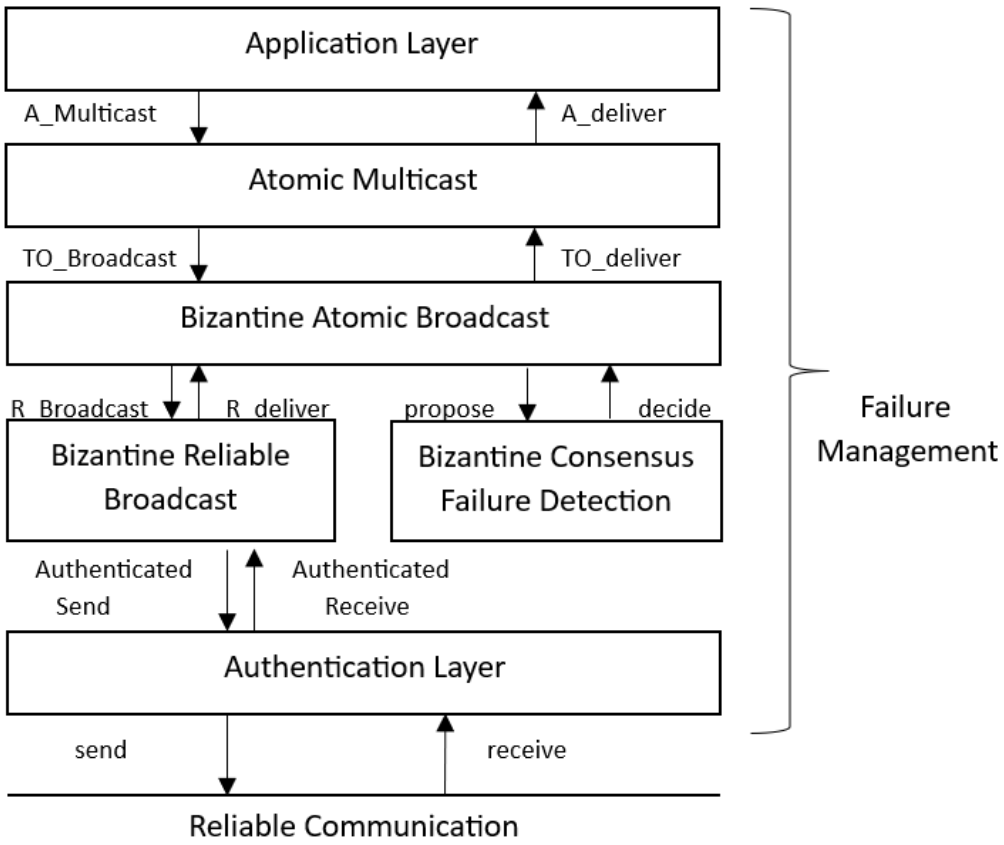


Figura 5.3 – Modelo tolerante a falhas bizantinas, adaptado de [15] para o Multicast Atômico.

de nodos defeituosos, os algoritmos presentes nos seções 5.1 e 5.2 procuram garantir as propriedades de progresso e vivacidade, na medida que, todas as réplicas não defeituosas concordem com uma ordem total (em virtude da difusão atômica) para a execução das solicitações. Para tal, os algoritmos propostos precisam satisfazer as propriedades do multicast atômico vistas em 2.6.

5.3.1 DCC-cft

Na tolerância a falhas do tipo colapso, a validade e o acordo derivam da proposição 1 abaixo. A integridade deriva da proposição 2 e, a ordem global acíclica da proposição 3.

Proposição 1. (Validade) Se um processo correto p faz multicast de uma mensagem m, então todos os processo corretos $q \in g$, onde $g \in m.dst$, entregam m. (Acordo) Se um processo correto p no grupo k entrega uma mensagem m, então todos os processos corretos $q \in g$, onde $g \in m.dst$, entregam m.

Um cliente DCC faz multicast da mensagem mandando-a para o grupo destinatário de menor ordem por difusão atômica. Pelas propriedades da difusão atômica, todo o

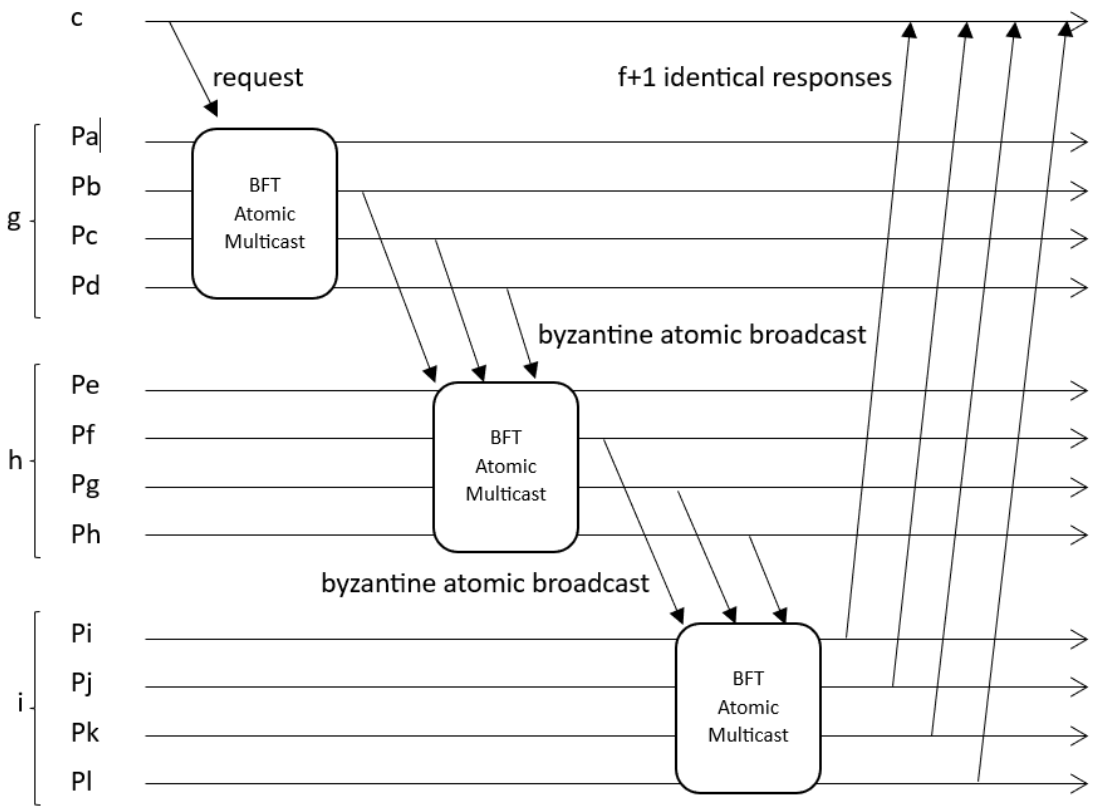


Figura 5.4 – Diagrama de tempo que demonstra o funcionamento do DCC em modo tolerante a falhas bizantinas com um processo não responsivo.

processo correto p do primeiro grupo destinatário g recebem a mensagem m. Os processos do primeiro grupo encaminham, em difusão atômica, para os processos do próximo grupo g', conforme o algoritmo 5.2. Dada a suposição de no máximo f processos possam falhar, ao menos $n - f$ processos encaminham a mensagem. Assim, todos os membros do grupo g' recebem a mensagem ao menos $n - f$ vezes. Como se trata de tolerância a falhas do tipo colapso, na primeira recepção a mensagem pode ser tratada pois todas serão idênticas. Assim, a despeito de falhas, as mensagens são da mesma forma sucessivamente encaminhadas a todos grupos de processos necessários conforme o DCC. Aqui não argumentamos sobre o aspecto do DCC fazer chegar a mensagem a todo destino devido. Isto já está argumentado junto ao algoritmo não tolerante a falhas em 4.4. Aqui argumentamos tão somente que o DCC funcionará da mesma forma a despeito de falhas conforme a suposição, tendo-se nodos tolerantes a falhas por replicação como descrito.

Proposição 2. (Integridade) Para qualquer processo correto p e qualquer mensagem m, p entrega apenas uma vez, somente se, $p \in g$, $g \in m.dst$ e m foi previamente enviada por um cliente c.

Uma mensagem m enviada ao DCC por um cliente externo é endereçada ao primeiro grupo g da topologia entre seus destinatários. Este grupo g entrega a mensagem e encaminha aos próximos nodos do grupo g' conforme o DCC. Um nodo que recebe a

Algoritmo 5.4 – Tipos e Estruturas DCC-bft

Topologia de Comunicação entre Processos Destino
  G {Conjunto de grupos de processos destino}
  $> : G \times G$ {Uma ordem total entre os grupos de processos}
  $E : \{(n, m) \mid n, m \in G, n < m\}$ {Todo grupo de processos tem uma aresta para cada um de seus mais altos}
  $GAD = (G, E)$ {Cada aresta (o,g) representa possibilidade de o enviar via broadcast para g}
  $in(g \in G) : \{\forall\ s \in G | (s, g) \in E\}$ {arestas de entrada de um grupo de processos destino g}
Relógio Vetorial de Arestas
  $EVC : E \rightarrow \mathbb{N}$ {Um relógio vetorial de arestas associa cada aresta a um natural}
Mensagem: toda mensagem m tem:
  $m.EVC : EVC$ {hora vetorial de m}
  $m.src \in G$ {originador de m}
  $m.dst \in \mathcal{P}(G) \mid \forall g \in m.dst, m.src \leq g$ {destino de m, qualquer subconjunto de grupos de processos}
  $m.mode \in \{fast, slow, \perp\}$ {modo fast ou slow}
  $m.data$ {dados da mensagem}
Processos: cada processo p pertencente a um grupo g tem as variáveis:
  $p.EVC : EVC$ {hora vetorial do grupo g}
  $p.b$ {um buffer de mensagens pendentes}
Primitivas de Comunicação:
  broadcast(m : *Mensagem*, *set of nodes* $\in g$): envia m via difusão atômica bizantina para cada processo do grupo g
  deliver(m : *Mensagem*): cada processo do grupo $g \in m.dst$ recebe m por broadcast

mensagem repassada pode ser intermediário ou outro destino. Em qualquer caso, uma mensagem recebida em um nodo, teve um originador externo, não tendo sido criada em nenhum nodo da topologia. Ainda, dado o uso da difusão atômica para o repasse entre grupos, e pelas propriedades da mesma, a mensagem será recebida uma única vez em cada processo do grupo. Pelo algoritmo, um processo reage a uma mensagem somente na primeira recepção da mesma. Assim, cada processo entrega a mensagem m somente uma vez e, somente se, a mensagem teve um cliente externo à topologia.

Proposição 3. (Ordem Global Acíclica) Relação $<$ é acíclica.

Dada as propriedades da difusão atômica vistas em 2.5 e, como o DCC mantém a ordem total global argumentada em 4.4, o algoritmo tolerante a falhas bizantinas analogamente respeita a ordem da mesma forma.

5.3.2 DCC-bft

Na tolerância a falhas do tipo bizantino, a validade e o acordo derivam da proposição 1 abaixo. A integridade deriva da proposição 2 e, a ordem global acíclica da proposição 3.

Algoritmo 5.5 – Algoritmo Principal DCC-bft

1: **Para enviar** m **em aMulticast, cliente executa como segue:**
2: let $m \in Mensagem$ with $m.mode \leftarrow \perp$, $m.dst \leftarrow dst$ *{inicializa a mensagem}*
3: $broadcast(m, processOfGroup(lowest(m.dst))$ *{envia para o primeiro grupo destino de m via difusão atômica bizantina}*

4: **Cada processo** n **de um grupo** g **executa como segue:**
5: **upon deliver**(em) **and** $em.mode = \perp$ *{recebe mensagem externa}*
6: $em.src \leftarrow n$
7: $DeliverAndForward(em)$
8: **upon deliver** m (f+1) times **and** $m.mode \neq \perp$ **and** $ItsForMe(m)$ *{1 msg genuína via difusao atômica bizantina de um nodo honesto}*
9: **if** $CanDeliverAndForward(m)$ **then** *{ sem dependencias causais}*
10: $DeliverAndForward(m)$
11: **else**
12: $n.b.append(m)$ *{armazena em buffer}*
13: **upon deliver** m (f+1) times **and** $m.mode \neq \perp$ **and** $\neg ItsForMe(m)$) *{1 msg não genuína via difusao atômica bizantina de um nodo honesto}*
14: **if** $CanForward(m)$ **then** *{ sem dependencias causais }*
15: $AccumulateAndForward(m)$
16: **else**
17: $n.b.append(m)$ *{armazena em buffer}*
18: **upon** $\exists\ m \in n.b \mid ItsForMe(m) \wedge CanDeliverAndForward(m)$
19: $DeliverAndForward(m)$ *{mensagem no buffer para este grupo pode ser entregue}*
20: **upon** $\exists\ m \in n.b \mid \neg ItsForMe(m) \wedge CanForward(m)$
21: $AccumulateAndForward(m)$ *{mensagem no buffer pode ser repassada}*

Proposição 1. (Validade) Se um processo correto p faz multicast de uma mensagem m, então todos os processo corretos $q \in g$, onde $g \in m.dst$, entregam m. (Acordo) Se um processo correto p no grupo k entrega uma mensagem m, então todos os processos corretos $q \in g$, onde $g \in m.dst$, entregam m.

Um cliente DCC faz multicast da mensagem mandando-a para o grupo destinatário de menor ordem por difusão atômica bizantina. Pelas propriedades da difusão atômica bizantina, todo o processo correto p do primeiro grupo destinatário g recebem a mensagem m. Os processos do primeiro grupo encaminham, em difusão atômica bizantina, para os processos do próximo grupo g', conforme o algoritmo 5.5. Dada a suposição de no máximo f processos sejam bizantinos, ao menos $n - f$ processos encaminham a mensagem. Assim, todos os membros do grupo g' recebem a mensagem ao menos $n - f$ vezes. Como se trata de tolerância a falhas do tipo bizantino, ao menos uma mensagem idêntica m vinda de um nodo honesto será suficiente. Assim, a despeito de falhas, as mensagens são da mesma forma sucessivamente encaminhadas a todos grupos de processos necessários conforme o DCC.

Algoritmo 5.6 – Funções Auxiliares DCC-bft

```
1:  n.Deliver(m : message) : delivers m.data to upper layer
2:  n.NextFastHop(dst : set of groups) : group  return the next g > currentGroup(), g ∈ dst
3:  n.NextHop() : group  return the next g > currentGroup(), g ∈ G
4:  n.ItsForMe(m : message) : boolean  return currentGroup() ∈ m.dst
5:  n.processOfGroup(dst : group) : set of nodes  return p ∈ dst
6:  n.currentGroup() : group  return g ⊃ n
7:  n.FastOrSlow(m : message)                          {encaminha mensagem cfe seu modo}
8:    m.EVC ← n.EVC
9:    if m.mode = true then
10:     broadcast(m, processOfGroup(NextFastHop(m.dst)))   {cada nodo envia a msg para o próximo
          grupo de destinatários}
11:   else
12:     broadcast(m, processOfGroup(NextHop()))   {cada nodo envia a msg para o próximo grupo da
          topologia}
13: n.CheckLastDestination(tmp : EVC) : boolean        {tem mesmos destinos que última msg?}
14:   result ← ∀ e ∈ n.cacheEVC | n.cacheEVC[e] = tmp[e]
15:   n.cacheEVC ← tmp
16:   return result
17: n.UpdateCompareEVC(m : message) : boolean          {retorna se m deve ser fast ou slow (t/f)}
18:   let old = n.EVC
19:   if m.mode = ⊥ then                               {é uma mensagem externa entrando em n}
20:     for each d ∈ m.dst | d ≠ n do                  {para cada aresta (n,d), d um nodo destino }
21:       n.EVC[(n, d)] + +                            {incrementa o relógio na aresta (n,d)}
22:   else                                             {mensagem recebida de outro nodo do GAD}
23:     for each e ∈ n.EVC | n.EVC[e] < m.EVC[e] do    {mantém máx. entre msg e nodo}
24:       n.EVC[e] ← m.EVC[e]                          {se está aqui, canDeliver = true}
25:   return n.ChekLastDestination(n.EVC − old)        {passa EVC com diferença em cada posição}
26: n.AccumulateAndForward(m : message)
27:   for each e ∈ m.EVC do                            {cada posição do relógio de m, é o maior entre}
28:     m.EVC[e] ← MAX(m.EVC[e], n.EVC[e])             { relógio do processo e da mensagem }
29:   broadcast(m, processOfGroup(NextHop()))          {cada nodo envia a msg para o próximo grupo da topologia}
30: n.DeliverAndForward(m : message)
31:   m.mode ← UpdateCompareEVC(m)                     {atualiza EVC e define modo da msg}
32:   n.Deliver(m)
33:   FastOrSlow(m)                                    {encaminha m ...}
34: n.CanForward(m : message) : boolean
35:   if ∃(s, n) ∈ E | ( s ≠ m.src and m.EVC[(s, n)] > n.EVC[(s, n)] ) then   {toda aresta (s,n) ...}
36:     return false                                   {de entrada, que não se origina em m.src, tem que ser satisfeita por n.EVC}
37:   return true                                      {n recebeu através das outras arestas todas mensagens necessárias}
38: n.CanDeliverAndForward(m : message) : boolean
39:   if (m.EVC[(m.src, n)] = n.EVC[(m.src, n)] + 1)
        and CanForward(m)) then                        {é a próxima msg a tratar de m.src e passa por CanForward}
40:     return true
41:   return false
```

Proposição 2. (Integridade) Para qualquer processo correto p e qualquer mensagem m, p entrega apenas uma vez, somente se, $p \in g$, $g \in m.dst$ e m foi previamente enviada por um cliente c.

Uma mensagem m enviada ao DCC por um cliente externo é endereçada ao primeiro grupo g da topologia entre seus destinatários. Este grupo g entrega a mensagem e encaminha aos próximos nodos do grupo g' conforme o DCC. Um nodo que recebe a mensagem repassada pode ser intermediário ou outro destino. Em qualquer caso, uma mensagem recebida em um nodo, teve um originador externo, não tendo sido criada em nenhum nodo da topologia. Ainda, dado o uso da difusão atômica bizantina para o repasse entre grupos, e pelas propriedades da mesma, será recebida ao menos uma mensagem idêntica m vinda de um nodo honesto. Pelo algoritmo, um processo reage ao menos uma mensagem idêntica m. Assim, cada processo entrega a mensagem m somente uma vez e, somente se, a mensagem teve um cliente externo à topologia.

Proposição 3. (Ordem Global Acíclica) Relação $<$ é acíclica.

Dada as propriedades da difusão atômica bizantina vistas anteriormente e, como o DCC mantém a ordem total global argumentada em 4.4, o algoritmo tolerante a falhas bizantinas analogamente respeita a ordem da mesma forma.

6. EXPERIMENTOS

Neste quinto capítulo definimos o escopo dos experimentos para um cenário não tolerante a falhas. Deste modo, o escopo aborda questões de funcionamento dos protótipos em 6.1.1, o ambiente onde os protótipos rodaram em 6.1.2, os cenários propostos de execução em 6.1.3. No final a metodologia utilizada nos experimentos e quais valores foram coletados em 6.1.4.

6.1 Escopo

Como descrito no capítulo 1 a proposta lançada pelo DCC busca ultrapassar as limitações impostas pelo algoritmo D&F eliminado a necessidade de sincronização com o uso das mensagens de END. Tendo claro esse objetivo e, para fins de comparação daquilo é considerado o estado da arte em protocolos de multicast atômico, conforme visto nas seções 3.2.1 e 3.2.2, o Skeen é peça fundamental nesse contexto. Portanto os experimentos estão circunscritos nos três algoritmos básicos: Skeen, D&F e DCC.

6.1.1 Protótipos

Os três algoritmos, Skeen, D&F, e o aqui proposto DCC, respectivamente descritos nas seções 3.1.1, 3.1.2 e capítulo 4, foram implementados na mesma linguagem, com a mesma biblioteca de distribuição e avaliados no mesmo ambiente. Os módulos cliente e nodo constituem o núcleo da implementação em cada caso: **Clientes** submetem mensagens em loop fechado, ou seja, mandam uma mensagem e esperam retorno para mandar a próxima. A carga de trabalho é incrementada com o aumento do número de clientes. Cada **Nodo** no GAD é um processo distribuído. Conforme o protocolo (Skeen, D&F, DCC), o último processo a receber a mensagem, ou todos, responde(em) ao cliente para fechar o loop.

6.1.2 Ambiente de Avaliação

O ambiente utilizado na avaliação é composto por 16 servidores de Rack do tipo PowerEdge R740xd da Dell. Cada servidor possui dois processadores escaláveis Intel® Xeon® da segunda geração, com 28 núcleos (SMT) Intel(R) Xeon(R) Gold 5120 CPU @2.20GHz 64bits cada. A capacidade da memória é 512GB do tipo DDR4 Synchronous LR-DIMM 2666 MHz (0.4 ns). Além disso possui quatro interfaces ethernet do tipo NetXtreme

II BCM57800 1/10 Gigabit Ethernet (modo de operação LACP 10Gb) com MTU de 1500 bytes e 24 discos magnéticos de 2365GB cada do tipo TOSHIBA com 10000rpm. Os nodos de processamento estão conectados diretamente em um switch de 48 portas do tipo Dell S4048T-ON na velocidade de 10Gb com suporte a LACP. O sistema operacional utilizado na avaliação foi o RedHat 8.4 64 bits (cerne 4.18.0-305.25.1) em conjunto com a runtime java 11.0.13 LTS provida pelo OpenJDK. A biblioteca (JAR) utilizada na comunicação via sockets foi netty-all-4.1.30.Final.jar.

Nenhum software intrusivo ou de inventário estão concorrendo com os recursos, tão pouco existe qualquer política de economia de energia definida que possa atrapalhar o processamento. Do ponto de vista do sistema operacional, não foi atribuída de forma administrativa qualquer tipo de localidade entre o processo da máquina virtual java e o nó NUMA, ficando totalmente a critério do escalonador a escolha da melhor CPU disponível.

6.1.3 Cenários de Avaliação

Avaliamos os protocolos para um total de 16 nodos conforme descrito em 6.1.2. Para avaliar o desempenho de protocolos multicast é importante observar o comportamento dos mesmos na presença de distintos padrões de carga para sub-conjuntos de nodos. Sendo assim, propomos três classes de cenários de avaliação, a saber:

- Sem localidade, seguindo uma carga aleatória. As mensagens são arbitrariamente enviadas a sub-conjuntos diferentes.
- Com localidade, seguindo uma carga do tipo TPC-C [21]. O TPC-C é um *benchmark* utilizado para comparar o desempenho de sistemas de processamento de transações online (OLTP). Sendo um sistema particionado (*warehouses*), operações podem ser enviadas a uma ou mais partições conforme definido em sua especificação.
- Com localidade, seguindo uma carga para sistemas particionados. Considera uma carga contendo combinações específicas de partições, aqui mapeadas para nodos do GAD. De forma recorrente a carga se aplica as combinações contendo as mesmas partições.

6.1.4 Metodologia

Para cada algoritmo, em cada cenário acima, consideramos carga incremental, com 1, 2, 4, 8, 16, 32 e 64 clientes, buscando a saturação. Cada experimento contempla a execução do protótipo por no mínimo três vezes. O tempo destinado ao processo de disseminação está limitado a 30 segundos para cada experimento, sendo que, os valores

extremos para cima e para baixo (~5%) são desconsiderados. Para avaliar os protocolos, à medida que aumentamos a carga de trabalho, coletamos as seguintes informações:

- Dados de latência (média em m/s por mensagem) e vazão (média de mensagens entregues por segundo).
- Volume de mensagens trocadas para entrega de uma mensagem.
- Tamanho adicional no cabeçalho da mensagem.
- Número de passos necessários para entrega de uma mensagem.
- Taxa de genuinidade, ou seja, o percentual de mensagens que envolvem somente os destinatários da mensagem.
- Taxa de Fast, ou seja, o percentual de mensagens que foram entregues de forma totalmente genuína.

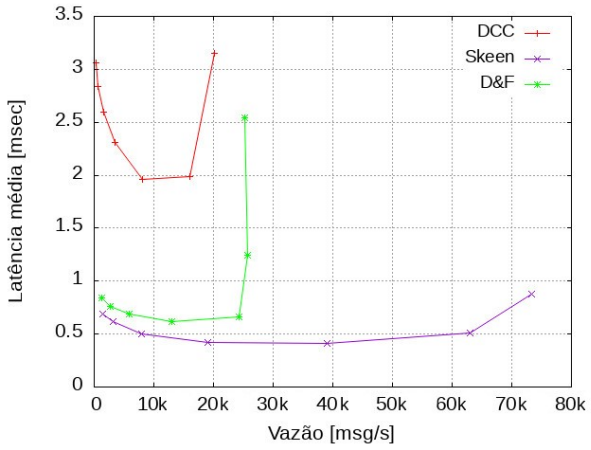
7. RESULTADOS

Neste de capítulo seis, começamos avaliando os resultados dos experimentos a partir da metodologia estabelecida no capítulo 5. Para que possamos avaliar a performance da proposta, o ponto central é sabermos se de fato o DCC pode ser considerado uma evolução do protocolo D&F em matéria de latência e vazão, tendo como critério de comparação mais realista os resultados do protocolo Skeen.

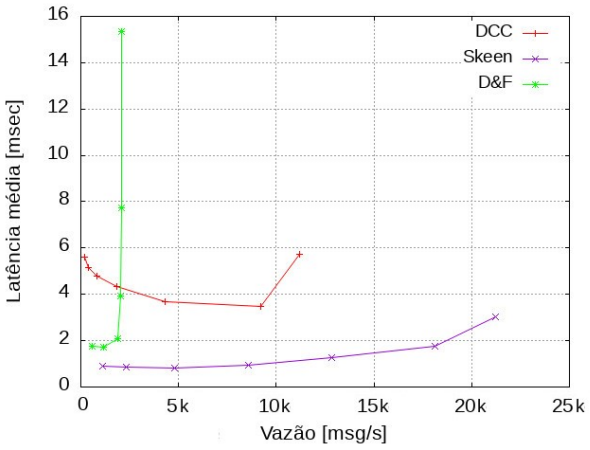
Inicialmente os resultados são postos para uma carga de trabalho sem localidade em 7.1. Logo após uma avaliação para duas cargas de trabalho com localidade em 7.2. Em 7.3 procuramos avaliar o impacto da redução do EVC no protocolo. Mais a frente são apresentadas as métricas específicas do protocolo DCC em 7.4 e, por fim, as considerações a cerca dos resultados 7.5.

7.1 Análise sem localidade

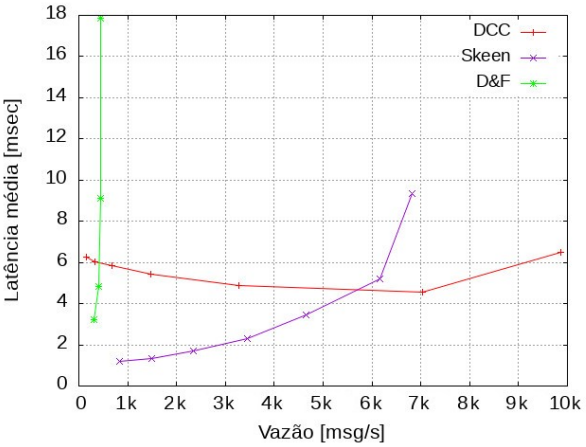
Na figura 7.1 estão representadas as curvas de latência e vazão para os três algoritmos. Para entender o efeito do tamanho do conjunto de destinatários, consideramos mensagens enviadas para 2, 4, e 8 processos, respectivamente a, b e c da figura 7.1. Adicionalmente avaliamos também com escolha aleatória do número de destinatários, até 16 por mensagem, gerando a figura 7.1d. O algoritmo D&F apresenta clara queda de desempenho à medida que mensagens são enviadas para subconjuntos maiores de destinatários. Como referido na seção 3.1.2, esta queda pode ser atribuída ao efeito comboio. Ou seja, quanto maior a quantidade de destinatários na mensagem multicast, maior será quantidade de nodos bloqueados a espera da mensagem de sinalização de desbloqueio, o elevando a latência e impedindo o aumento de vazão. Com relação ao algoritmo Skeen, também observamos perda de vazão quando o número de destinatários de cada mensagem aumenta. Atribuímos este comportamento ao crescimento quadrático do número de mensagens conforme o número de destinatários. Comparativamente aos demais, o DCC tem melhor vazão quanto maior o número de destinatários. Isto se deve ao já discutido, sendo e que para o DCC à medida que o número de destinatários cresce, diminui o tráfego não genuíno. À medida que o número de destinatários é menor, ainda que exista o modo *fast*, aumenta a possibilidade da mensagem ter que atravessar vários nodos de forma não genuína. Em relação à latência, o DCC apresenta regularmente patamar acima dos demais. Isto se deve tanto ao envolvimento de nodos adicionais como devido ao processamento imposto pelo mecanismo de EVC.



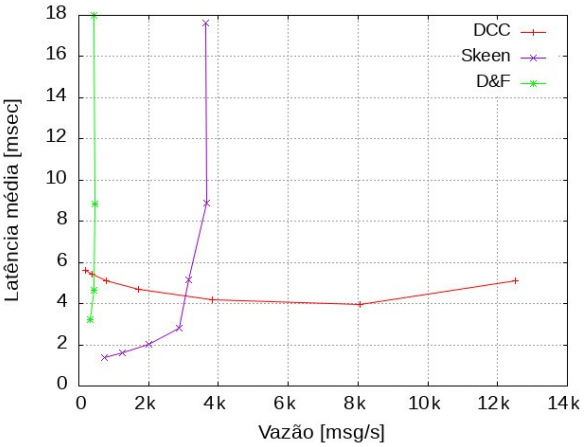
(a)



(b)

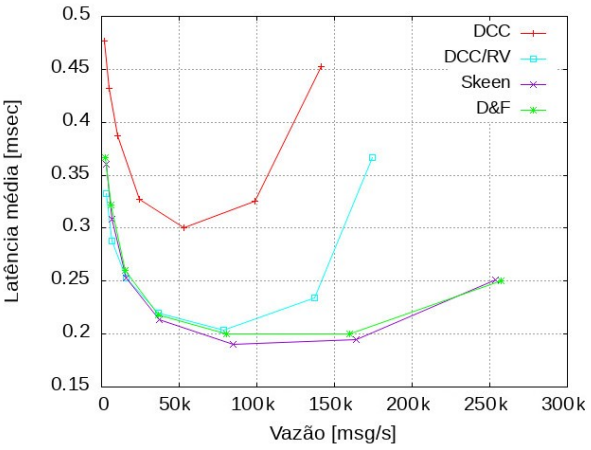


(c)

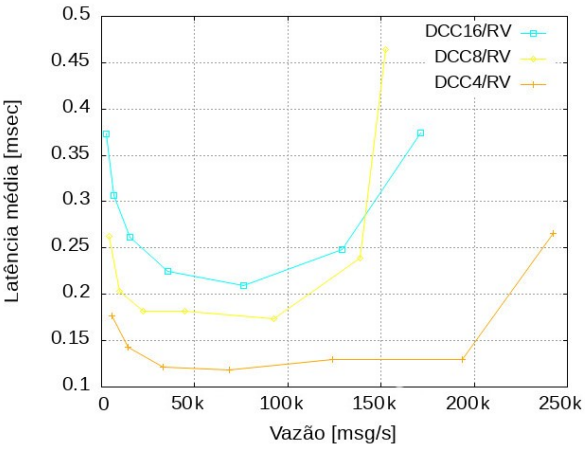


(d)

Figura 7.1 – Carga Aleatória (sem localidade): 2 (a), 4 (b), 8 (c), N (d) Nodos



(a)



(b)

Figura 7.2 – Carga TPC-C (localidade) e redução do EVC com GADs menores

7.2 Análise com localidade

Diferente da carga anterior, vamos explorar mensagens que tenham a tendência de ser encaminhadas aos mesmos nodos de processamento.

7.2.1 Carga TPC-C

A carga do TPC-C possui algumas particularidades, como por exemplo, a geração de altas taxas de mensagens com apenas um destinatário, algumas mensagens com dois destinatários e raras mensagens com três ou mais destinatários. Notamos (figura 7.2a) novamente que com baixo número de destinatários o DCC tem menor desempenho comparativo, mesmo com a redução do EVC aplicada.

7.2.2 Carga para sistemas particionados

Avaliamos aqui o envio de mensagens para grupos fixos de nodos, variando o tamanho dos mesmos em: oito grupos formados por dois nodos (2x8), quatro grupos compostos por quatro nodos (4x4) e dois grupos formados por oito nodos cada (8x2), respectivamente nas figuras 7.3(a,b), (c,d), e (e,f). Com isso, aproximamos o nicho de aplicações que utilizam dados particionados. Observamos nas figuras 7.3(a), (c), (e), que o DCC apresenta bom desempenho em geral quando o tráfego se mantém somente para os grupos definidos. Nota-se que quando se aumenta o tamanho do grupo, como o DCC propicia o pipeline através dos nodos envolvidos (não tem o bloqueio), o mesmo consegue sustentar taxas superiores aos demais. Dado o projeto do DCC, é importante considerar situações em que tráfego aleatório, para diferentes combinações de nodos, é também injetado. Neste caso, o protocolo DCC mudará em diversas situações do modo *fast* para *slow*, impactando seu desempenho. Nas figuras 7.3(b), (d), (f) mostramos para o mesmo caso base, as situações onde injetamos 1, 2 e 5% de mensagens para conjuntos de nodos aleatoriamente escolhidos, quantificando o impacto sobre o DCC.

7.3 Sobrecarga devido ao relógio vetorial

Para avaliar a sobrecarga imposta ao DCC pelo uso de relógio vetorial, experimentamos um mecanismo de redução dos mesmos, fazendo com que somente atualizações sejam transmitidas (só o que mudou) ao invés da estrutura completa. Essa otimiza-

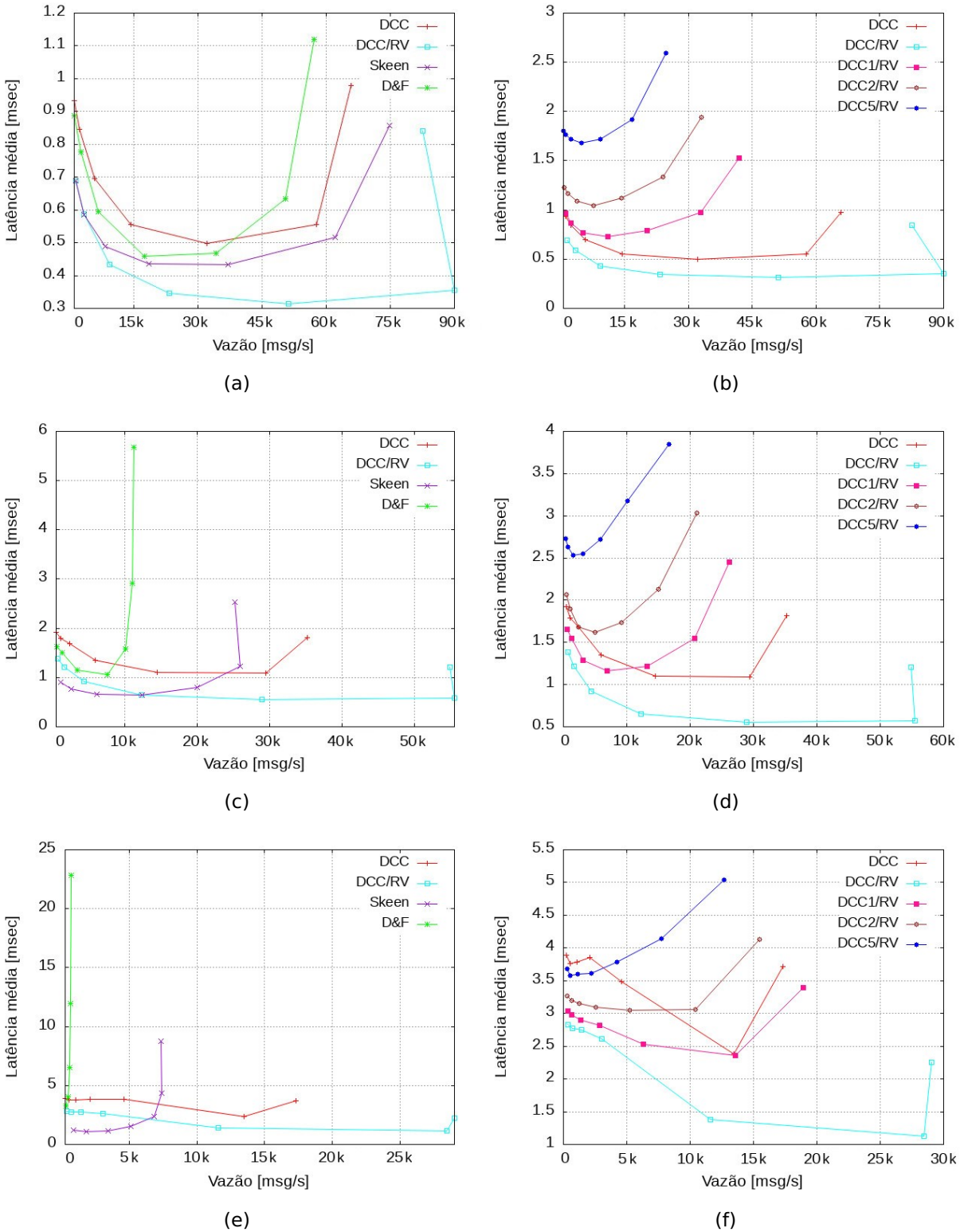


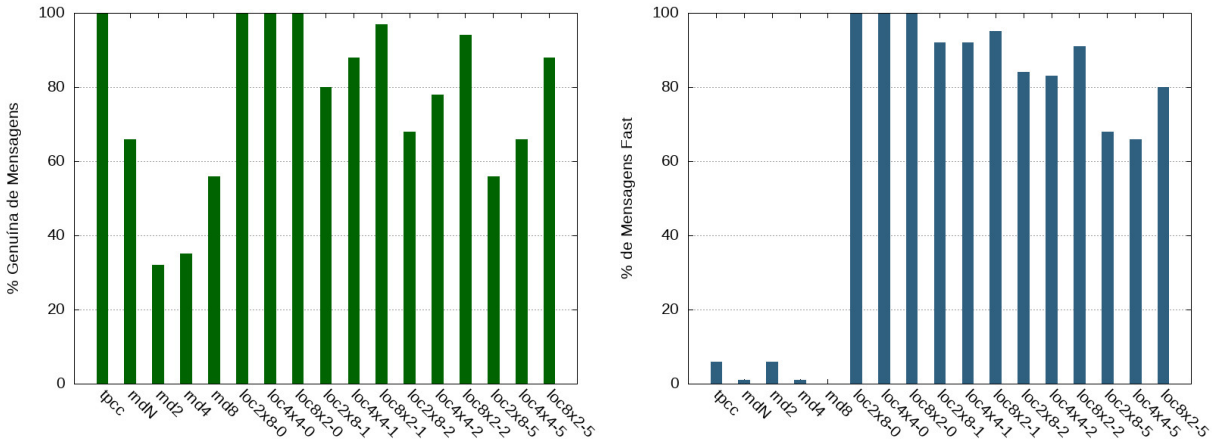
Figura 7.3 – Carga para sistemas particionados (localidade): 2x8 (a,b), 4x4 (c,d), 8x2 (e,f) Nodos/Grupos e redução do EVC com a taxa de mensagens aleatórias

ção traz um ganho significativo de latência e impacta positivamente também a vazão, Na figura 7.2a pela curva ’DCC/RV’ comparativamente a ’DCC’ fica mais clara essa diferença. Complementarmente, como a estrutura do relógio aumenta com o número de arestas,

diminuímos o GAD de 16 para 8 e 4 nodos, avaliando a latência neste cenário, que se mostra em 7.2b. De forma análoga fizemos o mesmo procedimento de otimização nas figuras 7.3(b), (d), (f), pela curva ’DCC/RV’ comparativamente a ’DCC’.

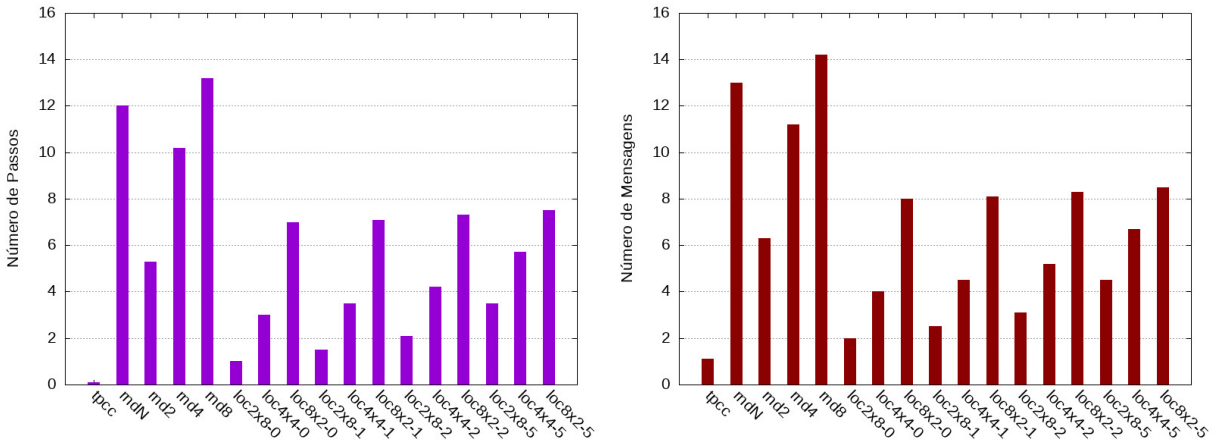
7.4 Métricas

Uma vez observado as curvas de latência e vazão, apresentamos os percentuais de mensagens genuínas e do tipo fast (figuras 7.4a e 7.4b), assim como o número de passos e volume de mensagens trafegadas (figuras 7.5a e 7.5b). Essas são métricas igualmente importantes nesse contexto de avaliação e compreensão dos resultados.



(a) (b)

Figura 7.4 – Pecentual de mensagens Genuínas e Fast



(a) (b)

Figura 7.5 – Número de Passos e Volume

Tendo o referencial teórico dos protocolos Skeen e D&F vistos nas seções 3.1.1 e 3.1.2 respectivamente e, se valendo das métricas obtidas das figuras 7.4 e 7.5 foi possível gerar uma análise comparativa. A tabela 7.1 sumariza os valores para a situação de uma mensagem multicast ser enviada para k destinatários em um sistema de n nodos levando-se em consideração o melhor e pior caso possível.

Algoritmo	**Volume**	**Tamanho**	**Passos**	**Genuinidade**	**Fast**
DCC	$\mathcal{O}n \mid \Omega_k$	$\mathcal{O}[n \times (n-1)/2] \mid \Omega_k$	$\mathcal{O}n \mid \Omega_k$	$\Theta_{3/4}n$	$\Theta_{n/2}$
Skeen	k^2	Δ	2	Δ	*na*
D&f	$k+1$	Δ	$k+1$	Δ	*na*

Tabela 7.1 – Análise comparativa.

Na tabela 7.1, diferentemente dos demais algoritmos onde a carga de trabalho é menos relevante, no DCC esse fator é absolutamente determinante e os seus valores podem ser interpretados da seguinte maneira: O volume de mensagens assim como o número de passos estão diretamente relacionados ao número de nodos não genuínos, portanto no melhor caso temos k mensagens disseminadas a nodos genuínos. No pior caso, temos até n mensagens endereçadas a todos os nodos da topologia. Do ponto de vista do tamanho adicional da mensagem, isto é, a quantidade arestas transcritas na forma dos EVCs, no pior caso esse valor é totalmente dependente do tamanho da topologia. No melhor caso está limitado ao número k de arestas diretamente envolvidas no processo de disseminação (mensagens Fast). Tanto o índice de genuinidade como o Fast se mostram atrelados a carga aplicada, sendo que, a genuinidade depende também do tamanho da topologia. Ou seja, quanto maior a topologia maior será a quantidade de mensagens não genuínas. Como dito anteriormente, a taxa de mensagens fast é proporcional a localidade, portanto quanto maior a localidade da carga maior será este indicador.

7.5 Considerações Finais

Como visto, o algoritmo de Skeen pode apresentar limitações de desempenho à medida que o número de processos cresce devido ao número de mensagens trocadas. Por outro lado, D&F sofre de efeito comboio devido ao bloqueio excessivo que ocorre, também à medida que o número de processos envolvidos cresce. Neste contexto, propusemos o protocolo DCC, que elimina o bloqueio de D&F, para isso abre-se mão da genuinidade do protocolo, sobretudo em cargas de trabalho sem localidade. Além disso emprega um mecanismo para registrar causalidade entre mensagens (EVC) a partir de um GAD.

Em síntese o DCC mostra boa vazão quando o número de processos destinatários de mensagens multicast cresce, sendo tanto melhor quanto estes grupos forem disjuntos.

Esta percepção vai ao encontro daquilo que foi explanado em 2.8. Ou seja, diferentemente do Skeen, que é um protocolo essencialmente de baixa latência, haja vista o número constante de passos, o DCC pode ser classificado como um protocolo de alta latência, em virtude do número maior de passos envolvidos conforme tabela 7.1. Por outro lado o DCC apresenta uma vantagem não presente no Skeen, a alta vazão representada nas figuras 7.3[c,d,e,f]. Ou seja, se aumentamos o número de destinatários da mensagem, se aumenta o paralelismo (pipeline mais profundo) da entrega. Sendo assim o protocolo DCC consegue sustentar taxas superiores aos demais em determinadas cargas de trabalho, mesmo que ele apresente uma latência eventualmente maior. O ponto central para que o protocolo possa chegar nesse patamar é taxa de localidade das mensagens. Aplicações modernas como bigdata e computação na nuvem apresentam muitas vezes a necessidade de particionamento e replicação dos dados. Assim é natural que exista uma recorrência no fluxo destas mensagens entre os mesmos grupos multicast.

8. CONCLUSÕES E TRABALHOS FUTUROS

Neste capítulo final são apresentadas as conclusões do presente trabalho e as possibilidades para a continuidade da pesquisa nesta área de estudo envolvendo algoritmos de multicast atômico.

8.1 Conclusões

A presente dissertação apresentou um novo algoritmo de multicast atômico chamado DaisyChainCast. Durante a construção do DCC alguns aspectos norteadores do algoritmo de D&F foram levados em consideração, como por exemplo, um processo interno de disseminação unicast atrelado a uma topologia disposta em GAD. O controle da consistência é o ponto central do protocolo. No algoritmo D&F a consistência está mantida na medida que os nodos esperam, com bloqueio, a mensagem de finalização. Ao passo que no DCC é proposto uma técnica não convencional através do uso dos relógios vetoriais de aresta (EVC) como mecanismo protetor do ciclo. E, para que as relações de causalidade estejam todas postas, faz-se necessário o encaminhamento das mensagens por nodos intermediários não genuínos, eliminando então a necessidade de bloqueio.

Dada a quantidade reduzida de algoritmos de multicast atômico disponíveis, as contribuições deste trabalho não se dão apenas na proposição de mais um algoritmo a cena, mas na própria evolução do algoritmo D&F. De acordo com os experimentos realizados e os seus respectivos resultados, foi possível constatar uma melhoria significativa entre os dois protocolos. Por se tratar de um algoritmo de alta vazão, a diferença para melhor se manifestou em praticamente todos cenários da avaliação, sobretudo quando temos presente o quesito localidade durante o processo de disseminação, naturalmente, em conjunto com a técnica de redução do EVC. De forma surpreendente, o DCC obteve melhores indicadores que o protocolo Skeen em determinados tipos de cargas de trabalho, principalmente aquelas com uma maior quantidade de destinatários.

8.2 Trabalhos Futuros

Durante a pesquisa envolvendo o algoritmo DCC, algumas possibilidades de melhoria foram levantadas, sobretudo no sentido das otimizações. Descrever-se-á algumas sugestões que possam dar continuidade ao protocolo.

8.2.1 Otimizações e Melhorias

Em primeiro lugar um quesito absolutamente faltante nesta pesquisa foi a não realização dos experimentos dentro de uma WAN. Isso implica em dizer que não sabemos a priori qual seria o comportamento do protocolo sujeito a latências (RTTs) maiores assim como taxas de retransmissão distintas. Também não é sabido o comportamento do protocolo acoplado a um mecanismo de tolerância a falhas (CFT e BFT), conforme fora colocado no capítulo 5.

Outro quesito importante mas que não foi possível investigar em maiores detalhes foi o impacto dos relógios vetoriais de aresta na latência do algoritmo. Na medida que o tamanho do GAD de nodos aumenta, a quantidade arestas aumenta proporcionalmente. Isto é, quanto maior o tamanho da topologia maior será o processamento, uma vez que, não apenas será necessário serializar o EVC como também processar a informação de causalidade presente. Do ponto de vista das otimizações, ou seja, além da compactação ou redução explícita dos relógios vetoriais (EVC) levando-se em consideração apenas as mudanças ocorridas durante o processo de disseminação dentro da topologia, a busca por um protocolo multicast atômico genuíno passa necessariamente pela eliminação ou no mínimo a redução de mensagens não genuínas. Aqui mensagens não genuínas são aquelas que denotam nenhuma dependência causal geradora de ciclo. Com base nisso, algumas tentativas de otimização foram experimentadas durante a pesquisa, mas todas esbarraram em impossibilidades ou em ineficiência. Por exemplo uma adaptação no algoritmo D&F removendo as mensagens de END e, consequentemente, a espera dos demais nodos (efeito comboio desapareceria). Tal situação foi pensada, exclusivamente, para momentos específicos do algoritmo quando houvesse a localidade nas mensagens. Isto é, o protocolo poderia se comportar no modo D&F quando não houvesse localidade, do contrário as mensagens seriam disseminadas em lock-free. Outra estratégia buscava a redução de passos através de inferências quando do chaveamento do modo FAST para o SLOW, entretanto o quesito latência (redes parcialmente síncronas) durante o processo de disseminação das mensagens cria cenários verdadeiramente desafiadores de serem tratados ou controlados.

Uma estratégia que talvez melhorasse o quesito da minimalidade é adoção de um grafo de dependência com base no histórico de mensagens entregues por nodo. Com tal estrutura de dados, tendo o conhecimento acumulado pelos nodos durante o processo de disseminação e, se valendo do algoritmo de caminho mais curto para um GAD, seria em tese possível se eliminar a participação de nodos intermediários sem nenhuma relação de transitividade com a mensagem. Vale lembrar que, por um lado estaríamos reduzindo o número de mensagens não genuínas mas, por outro lado, estaríamos aumentando o custo

computacional, uma vez que para encontrarmos o próximo salto teríamos a complexidade de tempo adicional de $\mathcal{O}(Arestas + Vertices)$ em virtude do caminho mais curto [7].

Por fim, outro fator a ser considerado para fins de latência é o processo de serialização deste grafo de dependência dentro de cada mensagem. Notadamente o DCC utiliza como principal elemento verificador de consistência (ciclo) os relógios vetoriais de cada aresta e, para tal, é necessário um processo de serialização para cada mensagem trafegada na topologia. Do mesmo modo, para nos valermos desta estratégia, precisaríamos encapsular uma parte deste grafo de dependência, levando-se em consideração estritamente as relações verdadeiramente importantes do ponto de vista da relação causal.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Ahmed-Nacer, T.; Sutra, P.; Conan, D. “The convoy effect in atomic mcast”. In: 2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops, 2016, pp. 67–72.

[2] Bessani, A.; Sousa, J.; Alchieri, E. “State machine replication for the masses with BFT-SMaRt”. In: DSN, 2014.

[3] Birman, K.; Joseph, T. “Reliable communication in the presence of failures”, *Trans. on Computer Systems*, vol. 5–1, fev. 1987, pp. 47–76.

[4] Cachin, C.; Guerraoui, R.; Rodrigues, L. E. T. “Introduction to Reliable and Secure Distributed Programming (2. ed.).” Springer, 2011.

[5] Castro, M.; Liskov, B. “Practical Byzantine Fault Tolerance”. In: Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI’99), 1999.

[6] Coelho, P.; Junior, T.; Bessani, A.; Dotti, F.; Pedone, F. “Byzantine fault-tolerant atomic multicast”, 2018, pp. 39–50.

[7] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. “Introduction to Algorithms, Second Edition”. The MIT Press and McGraw-Hill Book Company, 2001.

[8] Défago, X.; Schiper, A.; Urbán, P. “Total order broadcast and multicast algorithms: Taxonomy and survey”, *ACM Computing Surveys*, vol. 36–4, 2004, pp. 372–421.

[9] Défago, X.; Schiper, A.; Urbán, P. “Total order broadcast and multicast algorithms: Taxonomy and survey”, *ACM Comput. Surv.*, vol. 36–4, 2004.

[10] Delporte-Gallet, C.; Fauconnier, H. “Fault-tolerant genuine atomic multicast to multiple groups”. In: OPODIS, 2000.

[11] Fischer, M.; Lynch, N.; Patterson, M. “Impossibility of distributed consensus with one faulty process”. In: Proceedings of the 2nd Symposium on Principles of Database Systems (PODS’83), 1983.

[12] Gotsman, A.; Lefort, A.; Chockler, G. “White-box atomic multicast (extended version)”. 1904.07171, 2019.

[13] Guerraoui, R.; Levy, R. R.; Pochon, B.; Quéma, V. “Throughput optimal total order broadcast for cluster environments”, *ACM Trans. Comput. Syst.*, vol. 28–2, jul 2010.

[14] Guerraoui, R.; Schiper, A. “Genuine atomic multicast in asynchronous distributed systems”, *Theoretical Computer Science*, vol. 254–1-2, 2001, pp. 297–316.

[15] Hommel, G. “Communication-based systems. Proceedings of the 3rd international workshop, TU Berlin, Germany, March 31 – April 1, 2000”. 2000.

[16] Lamport, L. “Time, clocks, and the ordering of events in a distributed system”, *CACM*, vol. 21–7, jul. 1978, pp. 558–565.

[17] Le, L. H.; Eslahi-Kelorazi, M.; Coelho, P.; Pedone, F. “Ramcast: Rdma-based atomic multicast”. In: Proceedings of the 22nd International Middleware Conference, 2021, pp. 172–184.

[18] Pacheco, L.; Dotti, F.; Pedone, F. “Strengthening atomic multicast for partitioned state machine replication”. In: Proceedings of the Latin-American Symposium on Dependable Computing (LADC), 2022.

[19] Pacheco, L.; Dotti, F.; Pedone, F. “Strengthening atomic multicast for partitioned state machine replication”. In: Proceedings of the 11th Latin-American Symposium on Dependable Computing, 2023, pp. 51–60.

[20] Schneider, F. “Implementing fault-tolerant services using the state machine approach: A tutorial”, *ACM Computing Surveys*, vol. 22–4, dez. 1990, pp. 299–319.

[21] Serlin, O.; Sawyer, T.; Gray, J. “Tpc-c is an on-line transaction processing benchmark - https://www.tpc.org/tpcc/”, 1992.

[22] Skeen, D. “Crash recovery in a distributed database system”, Tese de Doutorado, University of California at Berkeley, Department of EECS, 1982.

[23] Urbán, P.; Défago, X.; Schiper, A. “Contention-aware metrics: Analysis of distributed algorithms”, 01 2000.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br