Daisy Chain Cast - um protocolo para Multicast Atômico*

Carlos Renan Schick Louzada[1], **Fernando Dotti**[1]

[1]Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Caixa Postal 1429 – 90.619-900 – Porto Alegre – RS – Brasil

carlos.schick@edu.pucrs.br, fernando.dotti@pucrs.br

Abstract. *Atomic multicast provides delivery and ordering guarantees to subsets of processes and is a fundamental mechanism to provide scalable services with strong consistency. While many genuine atomic multicast algorithms are derived from Skeen's algorithm, which uses communication from all to all processes, we have on the other side of the spectrum the equally genuine [Delporte-Gallet and Fauconnier 2000] protocol. The second restricts the directionality of communication and is attractive for its simplicity. However, it suffers from the convoy effect. Based on evaluations of both, this article proposes an alternative protocol, with the aim of eliminating the convoy effect of [Delporte-Gallet and Fauconnier 2000].*

Resumo. *O multicast atômico provê garantias de entrega e ordem a subconjuntos 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 and Fauconnier 2000]. Este restringe a direcionalidade da comunicação e é atrativo por sua simplicidade. Sofre porém do efeito comboio. A partir de avaliações de ambos, este artigo propõe um protocolo alternativo, com o objetivo de eliminar o efeito comboio de [Delporte-Gallet and Fauconnier 2000].*

1. Introdução

Serviços computacionais online atuais têm requisitos importantes de alta disponibilidade, desempenho e segurança. A alta disponibilidade implica 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 (RME) [Lamport 1978, Schneider 1990]. Entretanto, a proposta seminal de RME tem desempenho limitado, principalmente devido à redução da concorrência para garantir o determinismo.

Há duas principais abordagens para aumento de vazão em RME: o paralelismo intra-réplica; e o particionamento do serviço. O estado do serviço é particionado (*sharded*) e cada partição é mantida de forma replicada, empregando-se diferentes grupos de réplicas para diferentes partições. Requisições a partições diferentes podem ser processadas em paralelo. Requisições que necessitam de acesso a mais de uma partição (cross-shard) levam à necessidade de ordenação entre partições.

*Apoio: Fundação de Amparo à Pesquisa do Estado Do Rio Grande do Sul - FAPERGS PqG 07/21; Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq Universal 18/21.

A abstração de multicast atômico [Défago et al. 2004] atende a grande parte destes requisitos [Pacheco et al. 2022], 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* [Guerraoui and Schiper 2001]. 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 [Birman and Joseph 1987] 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 [Delporte-Gallet and Fauconnier 2000], 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.

Conforme experimentos realizados com os protocolos mencionados, e relatados neste artigo, onde avaliamos vazão e latência para diversas configurações, identificamos que o protocolo Skeen é penalizado à media que os nodos endereçados nas mensagens cresce, pois o número de mensagens aumenta quadraticamente em relação ao número de endereçados. Por outro lado, 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 [Ahmed-Nacer et al. 2016] 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.

Enquanto o protocolo de Skeen é alvo de desdobramentos na literatura, principalmente no sentido de criar variantes tolerantes a falhas, neste artigo 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.

Propomos neste artigo 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 todos os casos, sendo um protocolo semi-genuino. 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.

O artigo está organizado da seguinte forma: a seção 2 apresenta definições e suposições prévias importantes para o artigo, bem como apresenta Skeen e D&F; a seção 3 apresenta o protocolo DCC; as seções 4 e 5 apresentam experimentos e resultados; em seguida temos considerações finais em 6.

2. Fundamentação

2.1. Modelo de Sistema

Supomos um sistema distribuído usando passagem de mensagem como forma de comunicação, um conjunto ilimitado de processos clientes $C = \{c_1, c_2, ...\}$ e um conjunto limitado de processos destino $D = \{d_1, d_2, ...\}$. Processos são corretos. Eles não falham. Esta suposição é também adotada no protocolo de Skeen [Birman and Joseph 1987] e em [Delporte-Gallet and Fauconnier 2000]. Ela é feita considerando que cada processo tem uma implementação tolerante a falhas. No caso do protocolo de Skeen [Birman and Joseph 1987] pode-se mencionar extensões neste sentido, como [Guerraoui and Schiper 2001, Coelho et al. 2017]. Suposições adicionais referentes a falhas, bem como sincronia mínima para terminação são necessárias para a implementação tolerante a falhas de processos. Entretanto não abordaremos o problema neste nível.

2.2. Multicast Atômico

O multicast atômico é uma abstração fundamental para construir sistemas distribuídos confiáveis pois permite que clientes enviem mensagens a subconjuntos de processos com garantias de entrega e de ordenação [Hadzilacos and Toueg 1994]. Um processo ($m.src$) invoca a primitiva $multicast(m)$ para enviar a mensagem m a um subconjunto de destinatários dados em $m.dst$. Cada destinatário entrega m, ou $deliver(m)$, conforme as propriedades a seguir:

- **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 enviado por multicast pelo seu originador.
- **Ordem de prefixo**: para quaisquer duas mensagens m e m' e quaisquer dois processos p e q destinatários destas mensagens, se p entrega m e q entrega m' então p entrega m' antes de m ou q entrega m antes de m'.
- **Ordem acíclica**: seja a relação $\prec$ no conjunto de mensagens entregues pelos destinatários como: $m \prec m' \iff$ existe um processo que entrega m antes de m', a relação $\prec$ é acíclica.

Segundo [Guerraoui and Schiper 2001], para um algoritmo de multicast atômico *genuíno A*, cada execução *R* de *A* satisfaz ainda a seguinte propriedade:

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

Algoritmos que não satisfazem esta propriedade são ditos não genuínos. Se satisfazem parcialmente, são ditos semi-genuínos.

2.3. Protocolo de 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 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.

2.4. Protocolo de Delporte e Fauconnier (D&F)

Este algoritmo, apresentado em [Delporte-Gallet and Fauconnier 2000] 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.

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 1 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.

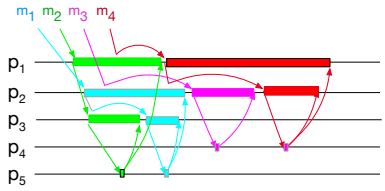


Figura 1. Troca de mensagens usando D&F. P_1 tem a menor ordem e P_5 a maior.

3. DCC - DaisyChainCast

Discutimos as idéias gerais do algoritmo, apresentamos o mesmo em detalhes, e discutimos aspectos de corretude do mesmo.

3.1. Idéia geral

Assim como D&F, o protocolo DCC assume uma ordem total entre os processos e topologia na forma de um GAD para repasse das mensagens enviadas, onde um processo pode repassar mensagens a quaisquer processos maiores. 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 2(a). Entretanto conforme velocidade de encaminhamento, um ciclo pode acontecer - veja a Figura 2(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 2(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.

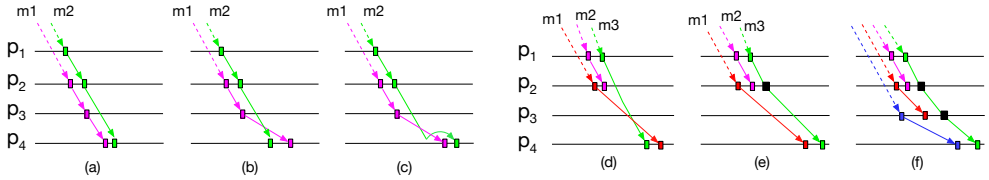


Figura 2. Anomalias na ordenação.

Um outro caso se apresenta na Figura 2(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 2(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 2(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 2(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.

3.2. 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 $ord(d), d \in D$ esta ordem. Estabelecemos que o conjunto E de arestas (dirigidas) do GAD como $\{(v,w) \cdot ord(v) < ord(w)\}$. Ou seja, para todo par de vértices v e w, se $ord(v) < ord(w)$ então a aresta dirigida $(v,w) \in E$, resultando em $[|D| \times (|D|-1)/2]$ arestas. Estas definições estão no Algoritmo 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. Em nosso algoritmo, propomos o uso de informação de causalidade para ajudar na ordenação de mensagens. Ou seja, quando um processo l (l para *low*, mais baixo) repassa uma mensagem m, os demais processos, mais altos, devem saber quais mensagens antecedem m do ponto de vista de l, de tal forma que ao entregar mensagens concorrentes possam respeitar a ordem de l. Como cada mensagem pode ter um subconjunto diferente de destinatários, um relógio de mensagens tratadas por processo não é suficiente. Propomos manter a informação de, para cada processo de ingresso, quantas mensagens repassou a cada destinatário h mais alto. Ou seja, para cada relação (l,h) faz-se necessário um registro. Como usamos o GAD, cada relação destas corresponde a exatamente uma aresta do GAD. Assim, chamamos de *relógio vetorial de arestas* o conjunto destes registros, sigla EVC para *edge vector clock*. Então, para uma dada topologia, um EVC é uma tupla com tantos relógios lógicos quantas arestas a topologia tem, e sabe-se relacionar a aresta com sua posição no relógio. Algoritmo 1, linha 8.

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 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 1, a partir da linha 14. 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.

Algorithm 1 Tipos e Estruturas

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: $EVC : E \to \mathbb{N}$ {*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: $m.dst \in \mathcal{P}(D)$ {*destino de* m*, qualquer subconjunto de processos*}
13: $m.data$ {*dados da mensagem*}
14: **Processo**: todo processo p tem as variáveis:
15: $p.EVC : EVC$ {*hora vetorial de* p}
16: $p.lastMsg : Mensagem$ {*última mensagem tratada pelo processo*}
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

3.3. Funcionamento

Envio em aMulticast. Para enviar em AMULTICAST, deve-se mandar a mensagem para o nodo mais baixo entre os destinatários, conforme Algoritmo 2, linha 1.

Recebimento no Processo de Ingresso. Ao receber uma mensagem externa, Algoritmo 2, linha 5, o processo de entrada é marcado como origem da mensagem em $m.src$ e a seguir, conforme Algoritmo 3, linha 25, 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 3, linha 26 e o processo pode tratar outra mensagem. A atualização do relógio pelo processo p, por ocasião do ingresso de uma mensagem, está na linha 15 do Algoritmo 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 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 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,

Algorithm 2 Algoritmo Principal

```
1: Para enviar m em aMulticast, cliente executa como segue:
2:     m.mode ← ⊥
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 ItsForMe(m)                 {be m fast or slow}
9:         if CanDeliverAndForward(m) then             { no dependencies }
10:            DeliverAndForward(m)
11:        else n.b.append(m)                          {armazena em buffer}

12:    upon receive(m) and ¬ItsForMe(m))               {slow only}
13:        if CanForward(m) then                       { no dependencies }
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}
```

m é encaminhada em modo $fast$ se ela tem os mesmos destinatários da última mensagem m' encaminhada por p (linha 8 do Algoritmo 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 2, linha 8, um processo destinatário verifica se ela pode ser entregue. Conforme Algoritmo 3, linha 33, 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 3, linha 25. Neste repasse, é feita a atualização do relógio do processo, Algoritmo 3, linha 12, 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 2, linha 12. Neste caso não existe mudança no relógio do processo. Somente o relógio da mensagem é atualizado, Algoritmo 3, linha 21. 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 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.

3.4. 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.

Algorithm 3 Funções Auxiliares

1: **n.Deliver**($m : message$) : delivers $m.data$ to upper layer
2: **n.NextFastHop**($dst : set\ of\ nodes$) : $node$ **return** the next $d > n$, $d \in dst$
3: **n.NextHop**() : $node$ **return** the next $d > n$, $d \in D$
4: **n.ItsForMe**($m : message$) : $boolean$ **return** $n \in m.dst$
5: **n.FastOrSlow**($m : message$) {*encaminha mensagem cfe seu modo*}
6: $m.EVC \leftarrow n.EVC$
7: **if** $m.mode$ = **true** **then** $send(m, NextFastHop(m.dst))$ **else** $send(m, NextHop())$

8: **n.CheckLastDestination**($tmp : EVC$) : $boolean$ {*tem mesmos destinos que última msg?*}
9: $result \leftarrow \forall\ e \in n.cacheEVC \mid n.cacheEVC[e] = tmp[e]$
10: $n.cacheEVC \leftarrow tmp$
11: **return** $result$

12: **n.UpdateCompareEVC**($m : message$) : $boolean$ {*retorna se m deve ser fast ou slow (t/f)*}
13: **let** $old = n.EVC$
14: **if** $m.mode = \perp$ **then** {*é uma mensagem externa entrando em n*}
15: **for all** $d \in m.dst \mid d \neq n$ **do** {*para cada aresta (n,d), d um nodo destino* }
16: $n.EVC[(n,d)]++$ {*incrementa o relógio na aresta (n,d)*}
17: **else** {*mensagem recebida de outro nodo do GAD*}
18: **for all** $e \in n.EVC \mid n.EVC[e] < m.EVC[e]$ **do** {*mantém máx. entre msg e nodo*}
19: $n.EVC[e] \leftarrow m.EVC[e]$ {*se está aqui, canDeliver = true*}
20: **return** $n.ChekLastDestination(n.EVC - old)$ {*passa EVC com diferença em cada posição*}

21: **n.AccumulateAndForward**($m : message$)
22: **for all** $e \in m.EVC$ **do** {*cada posição do relógio de m, é o maior entre*}
23: $m.EVC[e] \leftarrow MAX(m.EVC[e], n.EVC[e])$ { *relógio do processo e da mensagem* }
24: $send(m, NextHop())$ {*manda para o próximo na topologia*}

25: **n.DeliverAndForward**($m : message$)
26: $n.Deliver(m)$
27: $m.mode \leftarrow UpdateCompareEVC(m)$ {*atualiza EVC e define modo da msg*}
28: $FastOrSlow(m)$ {*encaminha m ...*}

29: **n.CanForward**($m : message$) : boolean
30: **if** $\exists (s,n) \in E \mid (\ s \neq m.src$ **and** $m.EVC[(s,n)] > n.EVC[(s,n)]\ )$ **then** {*toda aresta (s,n) ...*}
31: **return false** {*de entrada, que não se origina em m.src, tem que ser satisfeita por n.EVC*}
32: **return true** {*n recebeu através das outras arestas todas mensagens necessárias*}

33: **n.CanDeliverAndForward**($m : message$) : boolean
34: **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*}
35: **return true**
36: **return false**

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 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 $m''.dst \cap m'.dst \neq \varnothing$, $m''.dst \cap m.dst \neq \varnothing$ e $m'.dst \cap m.dst \neq \varnothing$, 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.

4. Experimentos

Protótipos. Os três algoritmos, Skeen, D&F, e o aqui proposto DCC, respectivamente descritos nas seções 2.3, 2.4 e 3, foram implementados na mesma linguagem, com 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.

Ambiente de Avaliação. O ambiente utilizado na avaliação é composto por de 16 servidores do tipo Intel(R) Xeon(R) Gold 5120 CPU @2.20GHz 64bits com 512GB de memória e 56 núcleos (SMT), duas interfaces ethernet de 10Gb (modo de operação LACP) e 24 discos magnéticos de 1.5TB cada. Os nodos de processamento estão conectados diretamente em um switch de 48 portas a 10Gb também 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.

Cenários de Avaliação. Avaliamos os protocolos para um total de 16 nodos. Para investigar o desempenho de protocolos multicast é importante avaliar o comportamento dos mesmos na presença de distintos padrões de carga para sub-conjuntos de nodos. Propomos três classes de cenários de avaliação:

- Sem localidade, seguindo uma carga aleatória. As mensagens são arbitrariamente enviadas a sub-conjuntos diferentes.

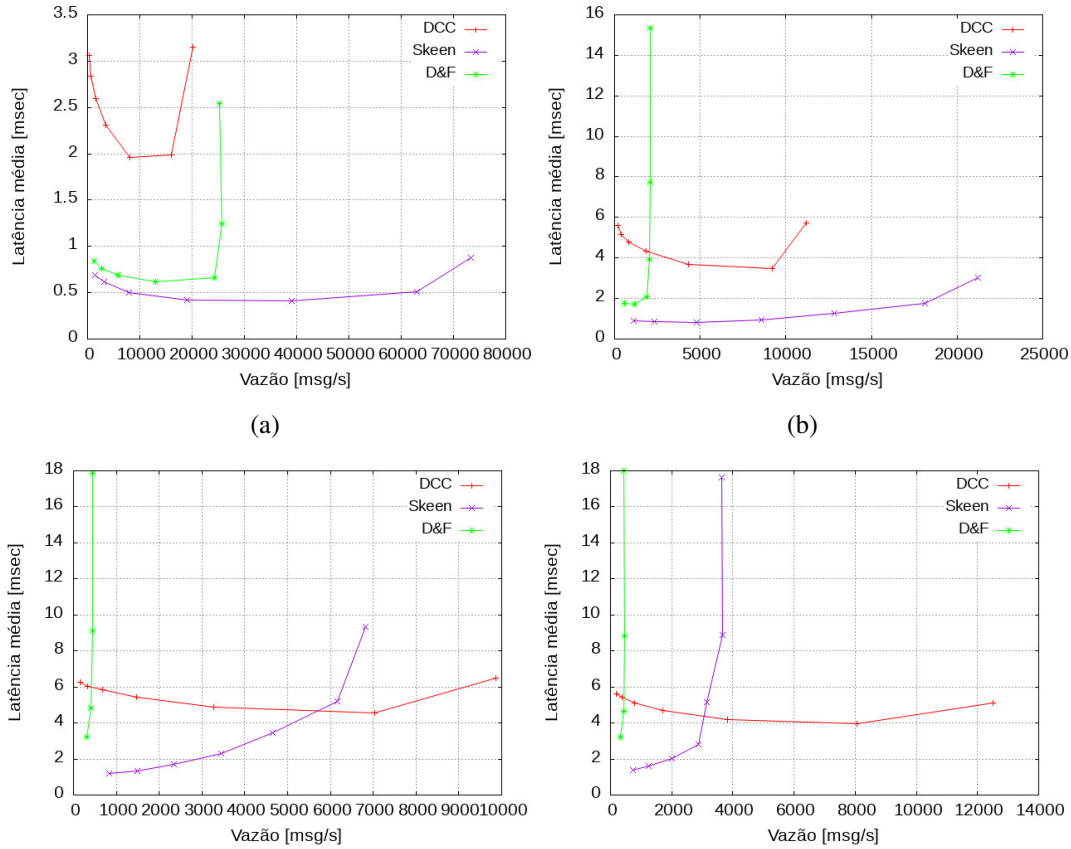


Figura 3. Carga Aleatória (sem localidade): 2 (a), 4 (b), 8 (c), N (d) Nodos

- Com localidade, seguindo uma carga do tipo TPC-C [Serlin et al. 1992]. 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.

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, geramos dados de latência (média em m/s por mensagem) e vazão (média de mensagens entregues por segundo).

5. Resultados

5.1. Análise sem localidade

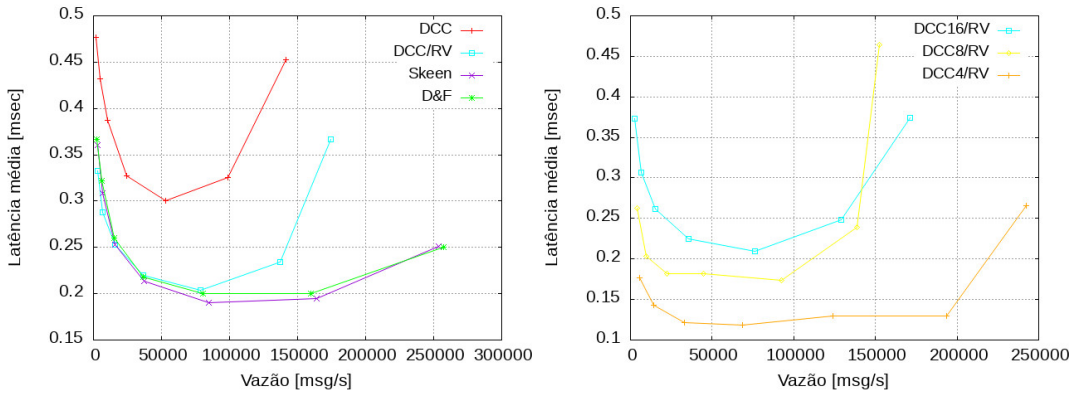
Na figura 3 estão representadas as curvas de latência e vazão para os três algoritmos. Para entender o efeito do tamanho do conjunto de destinatários, consideramos

mensagens enviadas para 2, 4, e 8 processos, respectivamente a, b e c da figura 3. Adicionalmente avaliamos também com escolha aleatória do número de destinatários, até 16 por mensagem, gerando a figura 3d. O algoritmo D&F apresenta clara queda de desempenho à medida que mensagens são enviadas para subconjuntos maiores de destinatários. Como referido na seção 2.4, 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.

5.2. Análise com localidade

Carga TPC-C. Diferente do cenário anterior, o TPC-C gera alta taxa de mensagens com apenas um destinatário, algumas mensagens com dois destinatários e raras mensagens com três ou mais destinatários. Notamos (figura 4a) novamente que com baixo número de destinatários o DCC tem menor desempenho comparativo.

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 otimização traz um ganho significativo de latência e impacta positivamente também a vazão, como se mostra na figura 4a pela curva 'DCC/RV' comparativamente a 'DCC'. Complementarmente, como a estrutura do relógio aumenta com o número de arestas, diminuímos o DAG de 16 para 8 e 4 nodos, avaliando a latência neste cenário, que se mostra em 4b.



(a) (b)

Figura 4. Carga TPC-C (localidade) e redução do EVC com DAGs menores

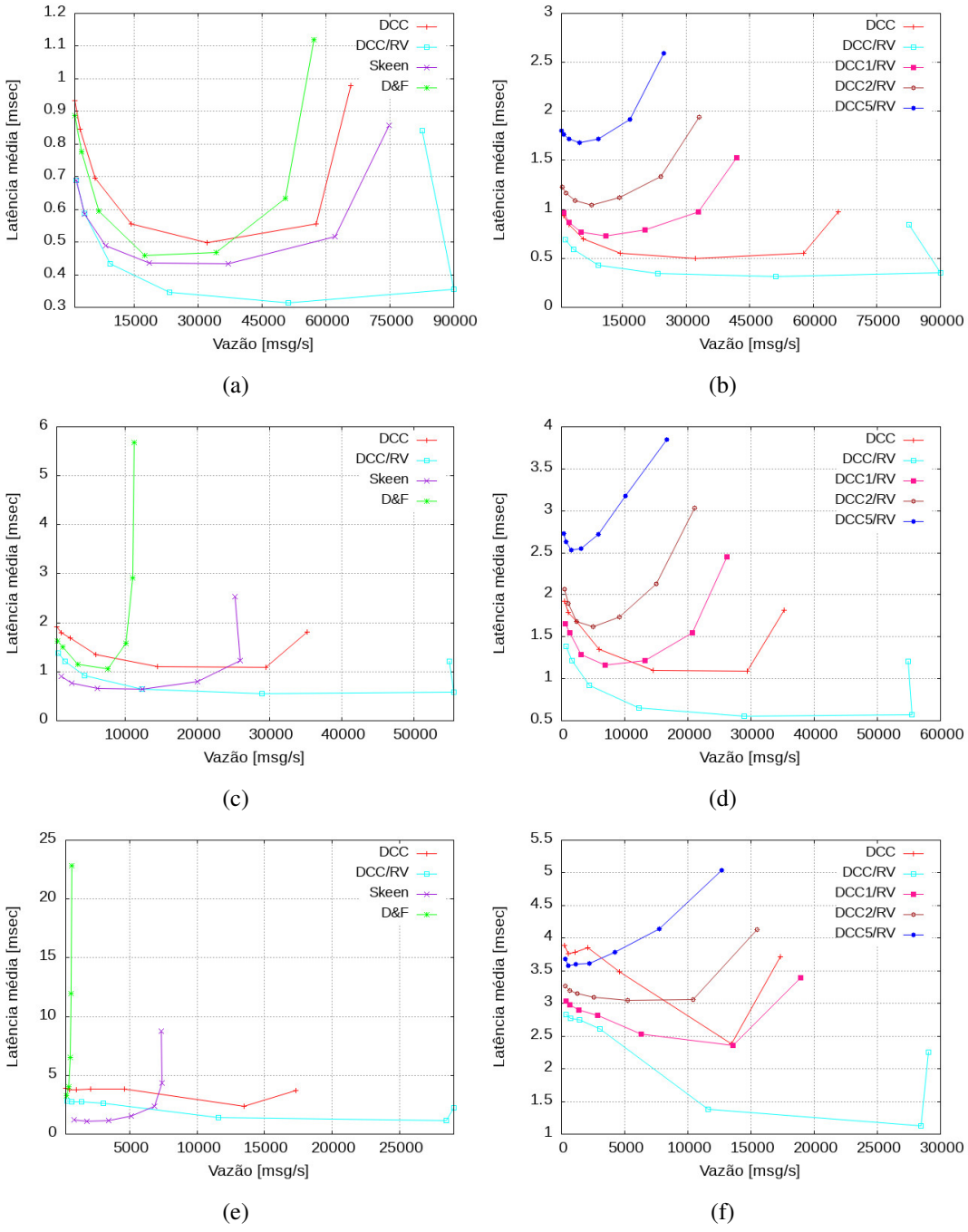


Figura 5. Carga para sistemas particionados (localidade): 2x8 (a,b), 4x4 (c,d), 8x2 (e,f) Nodos/Grupos e redução do EVC com a taxa de mensagens aleatórias

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 5(a,b), (c,d), e (e,f). Com isso, aproximamos o nicho de aplicações que utilizam dados particionados. Observamos nas figuras 5(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 5(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.

6. 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 abrindo mão da genuinidade do protocolo e empregando um mecanismo para registrar causalidade entre mensagens. DCC mostra boa vazão quando o número de processos destinatários de mensagens multicast cresce, sendo tanto melhor quanto estes grupos forem disjuntos. Em próximos passos pretende-se estender o uso do modo fast, ou seja, transmitir genuinamente, usando condições menos restritivas.

Referências

Ahmed-Nacer, T., Sutra, P., and Conan, D. (2016). The convoy effect in atomic mcast. In *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops*, pages 67–72.

Birman, K. P. and Joseph, T. A. (1987). Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76.

Coelho, P., Schiper, N., and Pedone, F. (2017). Fast atomic multicast. In *DSN*.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4).

Delporte-Gallet, C. and Fauconnier, H. (2000). Fault-tolerant genuine atomic multicast to multiple groups. In *OPODIS*.

Guerraoui, R. and Schiper, A. (2001). Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254(1-2):297–316.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565.

Pacheco, L., Dotti, F., and Pedone, F. (2022). Strengthening atomic multicast for partitioned state machine replication. In *Proceedings of the Latin-American Symposium on Dependable Computing (LADC)*.

Schneider, F. (1990). Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

Serlin, O., Sawyer, T., and Gray, J. (1992). Tpc-c is an on-line transaction processing benchmark - https://www.tpc.org/tpcc/.