Uma Estratégia de Difusão Agressiva para Aumentar a Vazão do Algoritmo de Consenso HyperPaxos

Djenifer R. Pereira
Departamento de Informática
Universidade Federal do Paraná
Curitiba, Brasil
drpereira@inf.ufpr.br
ORCID ID 0009-0002-8005-4131

Fernando M. Kiotheka
Departamento de Informática
Universidade Federal do Paraná
Curitiba, Brasil
fmkiotheka@inf.ufpr.br
ORCID ID 0009-0009-3788-4253

Elias P. Duarte Jr.
Departamento de Informática
Universidade Federal do Paraná
Curitiba, Brasil
elias@inf.ufpr.br
ORCID ID 0000-0002-8916-3302

***Resumo*—Algoritmos de consenso distribuído são essenciais para sistemas de armazenamento, bancos de dados, controle de acesso e orquestração de aplicações em nuvem. Este trabalho apresenta uma estratégia para melhorar a vazão do algoritmo HyperPaxos em termos de decisões por segundo. O algoritmo HyperPaxos é uma versão hierárquica de um dos principais algoritmos de consenso, o Paxos. O HyperPaxos é baseado na topologia virtual hierárquica vCube, que apresenta diversas propriedades logarítmicas. Os *acceptors* são organizados em *clusters* e os *proposers* executam as duas fases do Paxos escolhendo um *acceptor* dito difusor. O difusor é responsável por retransmitir as mensagens para os demais *acceptors* sobre o vCube. Neste trabalho, propomos que o difusor adote uma estratégia de difusão agressiva para transmitir, de uma só vez, as mensagens para uma maioria de *acceptors* paralelamente. A estratégia proposta foi implementada e comparada à versão original. Resultados obtidos mostram o desempenho superior da estratégia proposta em todos os cenários testados.**

I. Introdução

O acordo distribuído, ou consenso, é possivelmente o problema central da área de sistemas distribuídos. Informalmente, no problema do consenso, os processos propõem valores e, ao final, todos os processos decidem por um mesmo valor entre os propostos. Exemplos de aplicações diversas que utilizam consenso incluem bancos de dados distribuídos como o Google Spanner e o Cassandra [1]–[3], ferramentas de controle de acesso como o Chubby [4] e sistemas para orquestração de aplicações em nuvem como o Kubernetes [5].

Um dos principais algoritmos que resolve o problema do consenso é o Paxos [6]–[8]. No Paxos, os processos assumem papéis, que podem ser: *proposer*, *acceptor* e *learner*. Um *proposer* propõe valores, os *acceptors* decidem por um valor e os *learners* aprendem o valor decidido. O processo de decisão ocorre em duas fases, descritas a seguir de maneira bastante resumida. Na primeira fase, o *proposer* valida um número de proposta com os *acceptors* para propor um valor. Com um número de proposta, na segunda fase, o *proposer* faz uma proposta com valor para os *acceptors*. O consenso é atingido quando uma maioria de *acceptors* aceita um mesmo valor.

Devido à importância do Paxos, e ao fato de ser um algoritmo custoso, diversas variantes têm sido desenvolvidas [9]. Em particular, o Ring Paxos [10] é uma versão que utiliza uma topologia em anel com a proposta de aumentar a vazão em termos do número de decisões por segundo. No Ring Paxos os processos são organizados em anel, de forma que cada processo se comunica com apenas um outro processo, até atingir uma maioria. Porém, a estrutura em anel leva a um potencial aumento na latência do algoritmo. Neste contexto, foi proposto o algoritmo HyperPaxos [11], que se baseia na topologia escalável vCube [12] que apresenta diversas propriedades logarítmicas.

No HyperPaxos, a lógica original do Paxos é mantida, alterando somente a comunicação entre os papéis para criar a topologia virtual do vCube. Os *proposers* fazem as requisições para um *acceptor* denominado difusor e esse se torna responsável em repassar para os demais *acceptors* as requisições feitas pelo *proposer* usando a topologia do vCube. As mensagens são enviadas do maior ao menor *cluster* até atingir uma maioria de *acceptors*. Conforme a árvore de difusão é percorrida, as respostas dos *acceptors* vão sendo concatenadas junto da mensagem original. As respostas são encaminhadas para o *acceptor* responsável pela difusão e caso receba uma maioria de respostas confirmando a requisição, ele reencaminha as respostas para o *proposer*. Caso a maioria não seja atingida, o *acceptor* difusor prossegue para o próximo *cluster*.

Em [11], a libHyperPaxos, uma implementação do HyperPaxos baseada na libPaxos [13], é comparada com a implementação U-Ring Paxos [14] em termos de decisão por segundo. Para valores com tamanho menor que 1024 bytes, a libHyperPaxos tem melhores resultados que o U-Ring Paxos. No entanto, ao investigar cenários com vários números de *acceptors*, o desempenho da libHyperPaxos em termos da vazão, isto é, o número de decisões por segundo, apresenta grande variação. Assim, neste trabalho, propomos uma estratégia agressiva de difusão em que o *acceptor* difusor transmite, de uma só vez, as mensagens para uma maioria de *acceptors*. A estratégia proposta foi implementada e resultados de comparação mostram seu desempenho superior em todos os cenários testados.

O restante deste trabalho está organizado da seguinte forma.

A Seção II apresenta o algoritmo HyperPaxos, explicando como é feita a organização dos papéis no algoritmo e como é feita a comunicação entre eles. Na Seção III descrevemos as alterações propostas para o HyperPaxos. Em seguida, na Seção IV apresentamos a implementação e os resultados obtidos da comparação com a libHyperPaxos. E por fim, encerramos o trabalho com as conclusões na Seção VI.

II. O Algoritmo HyperPaxos: Uma Visão Geral

O HyperPaxos é um algoritmo de consenso distribuído tolerante a falhas. O algoritmo assume modelo temporal parcialmente síncrono com GST (*Global Stabilization Time*) [15] e o modelo de falhas *crash-recovery* [16]. Além disso, os enlaces de comunicação são perfeitos [17].

O HyperPaxos define para os processos os mesmos papéis do algoritmo Paxos. Apenas a comunicação entre os papéis é alterada, de forma que os *acceptors* formam uma topologia hierárquica vCube de n_a *acceptors*. Como os *proposers* estão fora da topologia, para realizar as fases 1 e 2, eles se comunicam usando um *acceptor* intermediador que realiza a transmissão sobre o vCube para os demais *acceptors*.

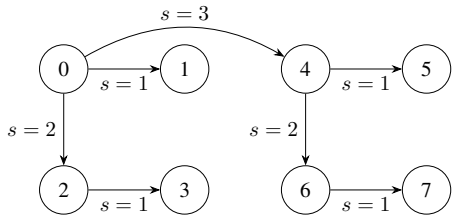
A topologia vCube vem do algoritmo de detecção de falhas vCube, que foi inicialmente proposto em [18], no contexto de diagnóstico distribuído. Nele, os processos são organizados em *clusters* de forma hierárquica, formando um hipercubo quando o número de processos é uma potência de dois e todos os processos estão corretos. O hipercubo apresenta simetria e diâmetro logarítmico. Na falta ou na falha de processos, essa topologia se reorganiza mantendo as propriedades logarítmicas. O vCube já foi utilizado como base para o desenvolvimento de diversos algoritmos distribuídos, como a difusão confiável [19], a difusão causal [20] e construção de rede *overlay* [21].

No HyperPaxos, os *acceptors* formam uma topologia do vCube, e cada *acceptor* se comunica com no máximo $\lceil \log_2 n \rceil$ *clusters* de *acceptors* utilizando um algoritmo de difusão inspirado em [22]. Os *clusters* são definidos pela função $c(i, s)$ que retorna a sequência de *acceptors* do *cluster* s para o *acceptor* i. Uma definição para a função $c(i, s)$ onde $\oplus$ denota a operação de ou exclusivo é dada por

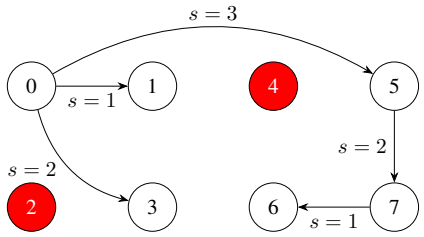
$$c(i,s) = (i \oplus 2^{s-1}, c(i \oplus 2^{s-1}, 1), \\ c(i \oplus 2^{s-1}, 2), \ldots, c(i \oplus 2^{s-1}, s-1)).$$

A Tabela I lista o $c(i, s)$ de todos os *acceptors* em um sistema com número de *acceptors* $n_a = 8$. Cada linha indica o número do *cluster* s e cada coluna indica o *acceptor* i de $c(i, s)$. Cada *acceptor* se comunica com o primeiro *acceptor* correto de cada *cluster*. Assim, num sistema sem falha e quando o número de *acceptors* for uma potência de dois, a topologia forma um hipercubo perfeito.

Em um sistema com 8 *acceptors*, a comunicação partindo do *acceptor* 0 acontece como mostra a Figura 1(a). Primeiramente, o *acceptor* 0 envia a mensagem para o *acceptor* 1 do *cluster* 1, o *acceptor* 2 do *cluster* 2 e o *acceptor* 4 do *cluster* 3. Ao receberem a mensagem, os *acceptors* 1, 2 e 4 continuam a difusão hierarquicamente. O *acceptor* 1 não encaminha mais mensagens, pois não tem mais *clusters* sob sua responsabilidade. Já os *acceptors* 2 e 4, continuam a difusão. O *acceptor* 2 envia a mensagem para o *acceptor* 3 do seu *cluster* 1, e o *acceptor* 4 envia a mensagem para os *acceptors* 5 e 6, dos *clusters* 1 e 2 respectivamente. Por fim, o *acceptor* 6 encaminha a mensagem para o *acceptor* 7 do seu *cluster* 1, completando o caminho mais longo no sistema de tamanho $\log_2 8 = 3$.



(a) Exemplo sem falhas.



(b) Exemplo com os *acceptors* 2 e 4 falhos.

Figura 1: Difusão partindo do *acceptor* 0 em um sistema de 8 *acceptors*.

Em um cenário de falha, o sistema adquire a forma encontrada na Figura 1(b) que mostra os *acceptors* 2 e 4 falhos. Neste caso, o *acceptor* 0 possui como primeiro *acceptor* correto do *cluster* 2 o *acceptor* 3, e o primeiro *acceptor* correto do *cluster* 3 se torna o 5. Note que o 5 que previamente não tinha responsabilidade de envio, agora é responsável por enviar mensagens para os *clusters* 1 e 2. Porém, não há *acceptor* correto em seu *cluster* 1, e o *acceptor* 5 então encaminha a mensagem para o *acceptor* 7 do *cluster* 2. O *acceptor* 7 que também não tinha responsabilidade de enviar uma mensagem agora precisa enviar uma mensagem para o *acceptor* 6 do seu *cluster* 1.

Na fase 1, o *proposer* envia um pedido de preparação para um *acceptor* dito difusor, que será responsável por difundir a mensagem no vCube. O difusor faz a difusão para os seus *clusters*, do maior para o menor, um por vez. Isto é, o *acceptor* difusor i envia uma mensagem para o primeiro *acceptor* correto j de $c(i, s)$, começando com $s = \lceil \log_2 n_a \rceil$, aguardando as respostas desse *cluster*. A resposta do próprio *acceptor* difusor vai junto do pedido de preparação que ele difunde para seus *clusters*.

Cada *acceptor* i, ao receber um pedido de preparação de um *acceptor*, encaminha o pedido para os todos os primeiros *acceptors* corretos dos seus *clusters* de $c(i, 1)$ até $c(i, s-1)$, junto da sua própria resposta ao pedido de preparação. Caso o *acceptor* não tenha a quem transmitir a mensagem, isto é, é uma folha na árvore de difusão, então esse *acceptor* encaminha

s	$c(0,s)$	$c(1,s)$	$c(2,s)$	$c(3,s)$	$c(4,s)$	$c(5,s)$	$c(6,s)$	$c(7,s)$
1	(1)	(0)	(3)	(2)	(5)	(4)	(7)	(6)
2	$(2,3)$	$(3,2)$	$(0,1)$	$(1,0)$	$(6,7)$	$(7,6)$	$(4,5)$	$(5,4)$
3	$(4,5,6,7)$	$(5,4,7,6)$	$(6,7,4,5)$	$(7,6,5,4)$	$(0,1,2,3)$	$(1,0,3,2)$	$(2,3,0,1)$	$(3,2,1,0)$

Tabela I: Valores de $c(i,s)$ para 8 *acceptors*.

uma mensagem com todas as respostas de volta ao *acceptor* difusor.

Quando o *acceptor* difusor recebe as respostas de todo um *cluster* para o pedido de preparação, ele avalia se existe uma maioria de promessas que validam o número de proposta do *proposer*. Em caso afirmativo, o *acceptor* encaminha as respostas recebidas de volta para o *proposer*. Se não, o *acceptor* deve continuar a difundir a mensagem para um *cluster* de tamanho s menor. Caso o *acceptor* esgote todos os *clusters*, sem validação do número de proposta, o *proposer* é instruído a reiniciar a fase 1 com um número de proposta maior.

No melhor caso, a difusão para o maior *cluster* é suficiente, particularmente quando o número de *acceptors* é uma potência de 2 e todos estão corretos, pois neste caso, o maior *cluster* tem tamanho $n_a/2$. Como a difusão é inicializada com a resposta do difusor para o pedido de preparação, são $n_a/2+1$ respostas. Assim, se todas as respostas forem promessas que validam o número de proposta, a maioria necessária para a validação é atingida.

A Figura 2 mostra um sistema distribuído com 2 *proposers* e 8 *acceptors*. Neste exemplo, o *proposer* escolhe o *acceptor* 0 como difusor e envia seu pedido de preparação PREP com número de proposta 1. Ao receber o pedido de preparação, o *acceptor* 0 encaminha o pedido de preparação com a sua resposta PROM para o *acceptor* 4 do *cluster* 3. O *acceptor* 4 faz o mesmo, encaminhando o pedido de preparação com a resposta anterior e sua resposta para os *acceptors* 5 e 6, dos *clusters* 1 e 2. Por fim, o *acceptor* 6 envia o pedido de preparação com as respostas para o *acceptor* 7.

Os *acceptors* 5 e 7 são folhas na árvore de difusão, e portanto encaminham as respostas para o *acceptor* difusor 0. Ao receber todas as respostas, o *acceptor* 0 verifica se existe uma maioria de promessas que validam o número de proposta para poder enviar para o *proposer*. Como existe uma maioria nesse caso, o *acceptor* 0 pode enviar essas respostas para o *proposer*, validando o número de proposta 1 e permite que ele prossiga para a próxima fase.

A execução da fase 2 é semelhante à fase 1. O *proposer* envia a proposta com valor para outro *acceptor*, que será responsável por difundir a mensagem. A difusão ocorre da mesma maneira que na fase 1, de *cluster* em *cluster*. Ao atingir a maioria, o difusor encaminha a decisão para todos os *proposers* e todos os *learners*. Caso a maioria não seja atingida, o *proposer* é instruído a iniciar uma nova fase 1 com um novo número de proposta.

A Figura 3 mostra um cenário com 8 *acceptors* onde o *proposer* 0 executa a fase 2 e o *acceptor* 7 falhou. O *proposer* escolhe o *acceptor* 1 para a difusão e envia sua proposta PROP com número 1 e valor x. O *acceptor* 1, ao receber a proposta, a aceita e a encaminha com sua resposta ACC para o *acceptor* 5 do seu maior *cluster*, o *cluster* 3. O *acceptor* 5 faz o mesmo, aceitando a proposta e a encaminhando com as respostas para o *acceptor* 4 do *cluster* 1 e o *acceptor* 6 do *cluster* 2, já que o *acceptor* 7 está falho. Como os *acceptors* 4 e 6 são folhas na árvore de difusão, eles retornam as respostas para o *acceptor* difusor 1. Ao receber todas as respostas do *cluster*, o *acceptor* 1 verifica se tem uma maioria de aceites para a proposta.

Como neste caso não há maioria, o *acceptor* difusor 1 prossegue para o seu *cluster* 2, enviando a proposta para o *acceptor* 3 como mostra a Figura 4. O *acceptor* 3 aceita a proposta e repassa para o *acceptor* 2 do seu *cluster* 1. O *acceptor* 2 é uma folha na árvore, então retorna a resposta para o *acceptor* difusor 1. Agora o *acceptor* 1 conseguiu a maioria de aceites atingindo o consenso e pode enviar as respostas para todos os *proposers* e todos os *learners*.

III. Uma Estratégia de Difusão Agressiva para o HyperPaxos

Na versão original do HyperPaxos, a difusão para o primeiro *cluster* é suficiente apenas no caso do número de *acceptors* ser uma potência de 2, e o sistema não apresentar falhas. Caso contrário, a primeira difusão pode não ser suficiente para atingir a maioria necessária para a fase 1 e 2. Isso é exemplificado nas Figuras 3 e 4, em um sistema com 8 *acceptors* e o *acceptor* 7 falho, a primeira difusão feita pelo *acceptor* difusor 1 é para o *cluster* 3, que não forma uma maioria. O mesmo ocorre se a difusão fosse iniciada nos *acceptors* 0, 2 e 3. Nesse caso, a difusão de *cluster* em *cluster* aguarda o retorno do *cluster* 3 antes de transmitir para o *cluster* 2, aumentando a latência.

Assim, propomos uma estratégia agressiva para o difusor enviar, de uma só vez, a mensagem para diversos *clusters* de *acceptors* ao mesmo tempo, paralelamente. Isso é possível pois o HyperPaxos depende de um detector de falhas, que informa quais processos estão falhos. Considerando quantos *acceptors* estão falhos em cada *cluster*, a estratégia seleciona os *clusters* necessários para formar uma maioria. Duas variações são propostas. A primeira variação busca os maiores *clusters* que formam a maioria, utilizando a menor quantidade possível de *clusters*. Desta forma, nas Figuras 3 e 4, o difusor 1 faz a difusão em paralelo para os *clusters* 2 e 3. Como o *cluster* 3 possui 3 *acceptors* (4, 5 e 6), e o *cluster* 2 possui dois *acceptors* (2 e 3) e a difusão começa pelo *acceptor* 1, então 6 *acceptors* são atingidos: 1, 2, 3, 4, 5 e 6, formando a maioria necessária.

A segunda variação proposta utiliza a força bruta para selecionar o melhor conjunto de *clusters* que juntos formam uma maioria. Assim, o número de *acceptors* fica mais próximo do mínimo necessário. Esta variação demanda examinar todas

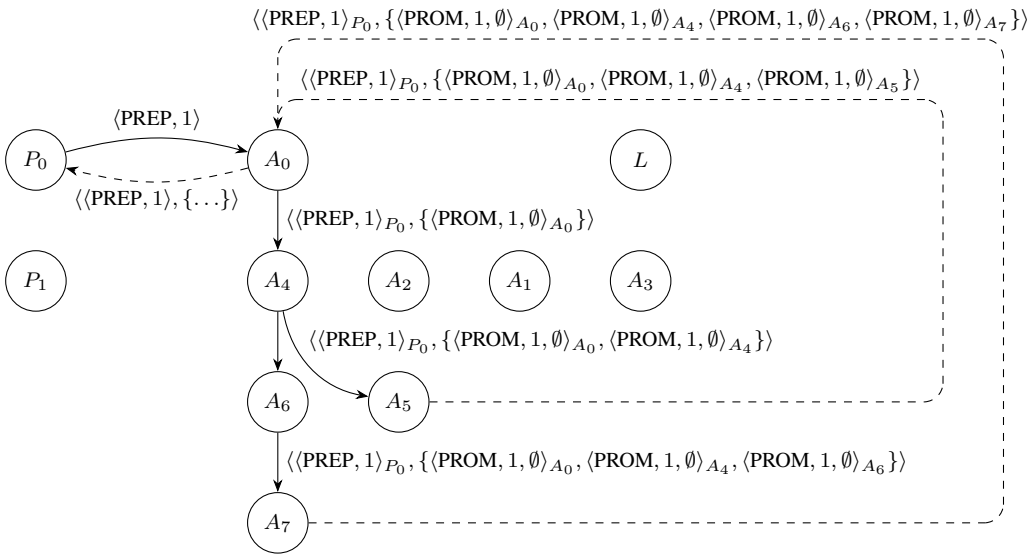


Figura 2: Difusão na fase 1 do maior *cluster* do *acceptor* 0 em um sistema com $n_a = 8$.

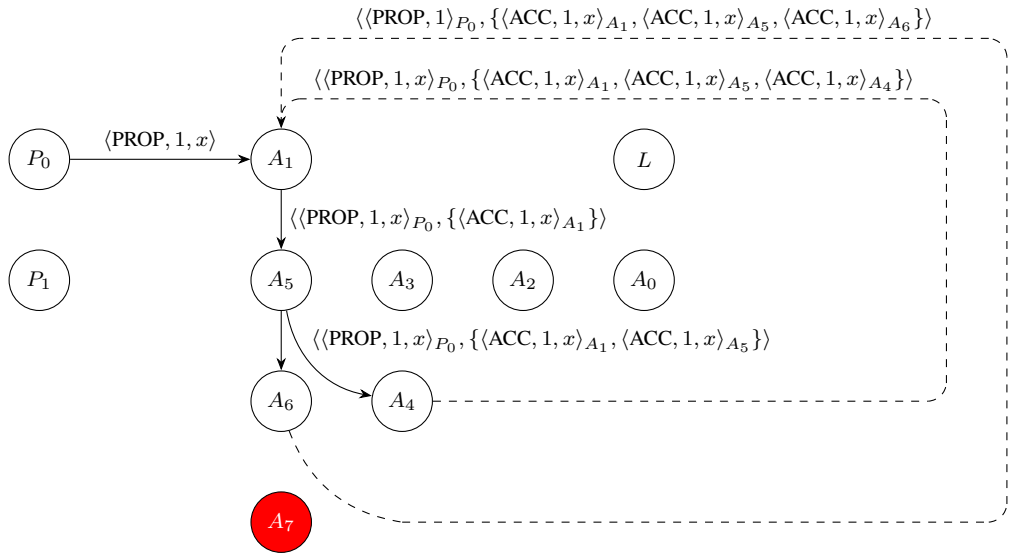


Figura 3: Difusão na fase 2 do maior *cluster* do *acceptor* 1 em um sistema com $n_a = 8$ e *acceptor* 7 falho.

as alternativas de combinação de *clusters*, verificando quantos *acceptors* corretos há em cada um, para escolher o conjunto de *clusters* para os quais fazer a difusão em paralelo. A complexidade da segunda variação é, portanto, exponencial no número de *clusters*, ou seja, $\mathcal{O}(2^{\log_2 n_a})$, onde n_a é o número de *acceptors*. Como o número de *clusters* é logarítmico, a complexidade se torna linear no número de *acceptors*.

No mesmo cenário das Figuras 3 e 4, a segunda variação faz a primeira difusão em paralelo para os *clusters* 1 e 3. Desta maneira, a quantidade de *acceptors* atingidos é 5 (0, 1, 4, 5 e 6), o mínimo necessário para maioria. Considere agora outro cenário com 14 *acceptors* sem falha, como mostra a Figura 5. Considerando que o difusor é o *acceptor* 0, na primeira variação o difusor comunica em paralelo com os dois maiores *clusters*, i.e. *clusters* 4 e 3), ou seja, com 11 *acceptors*. Na segunda variação, são selecionados os *clusters* 4 e 1, portanto a comunicação ocorre com apenas 8 *acceptors*.

Note que é possível que haja mais de um subconjunto de *clusters* que formam a maioria na estratégia na segunda variação. Neste caso, é escolhido o subconjunto com a menor quantidade de *clusters*. Vale observar que ainda outras estratégias podem ser adotadas.

É importante destacar que qualquer variação da estratégia proposta utiliza apenas um único *cluster* nos casos em que o sistema forma uma hipercubo perfeito, i.e. o número de *acceptors* é uma potência de 2 e todos os *acceptors* estão corretos. Neste caso, é mantido o desempenho original. Nos casos em que o sistema não forma um hipercubo perfeito, vários *clusters* recebem as mensagens de forma simultânea, diminuindo a latência do algoritmo.

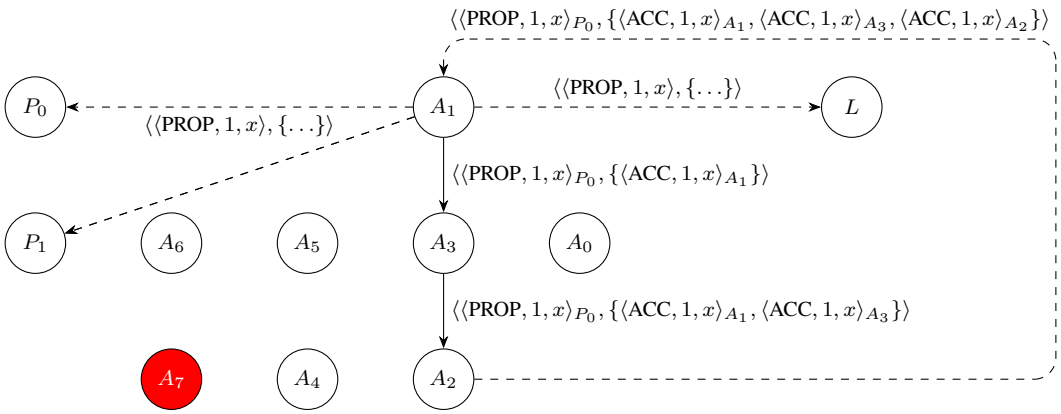


Figura 4: Difusão na fase 2 do segundo maior *cluster* do *acceptor* 1 em um sistema com $n_a = 8$ e *acceptor* 7 falho.

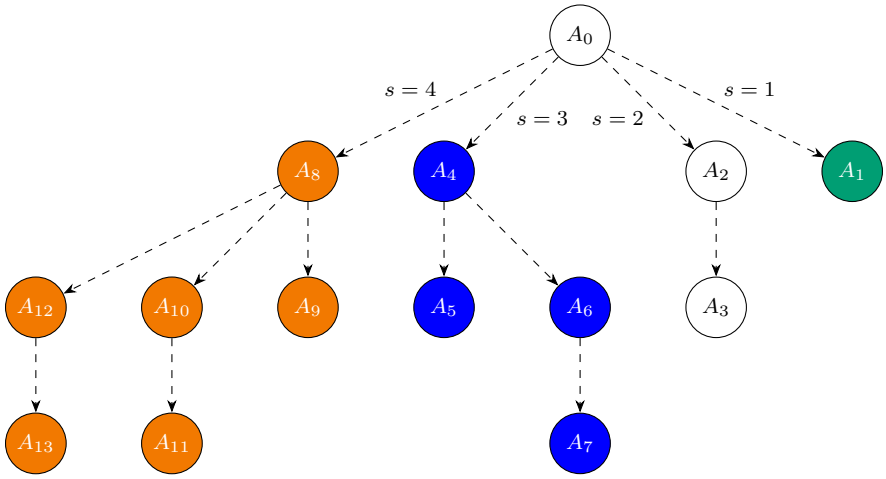


Figura 5: Difusão partindo do *acceptor* 0 em um sistema com $n_a = 14$.

IV. Resultados Experimentais

Modificamos a libHyperPaxos [23] de acordo com a Seção III para testar a estratégia proposta. Foram realizados experimentos comparando a quantidade de decisões por segundo na libHyperPaxos original, e em versões modificadas. Nos testes utilizamos um cliente e várias réplicas. O cliente é o responsável por submeter valores a um *proposer* e medir a quantidade de valores decididos por segundo.

Uma réplica é um processo que implementa os três papéis do Paxos: *proposer*, *acceptor* e *learner*. Os processos foram executados em núcleos distintos de uma mesma máquina física. A máquina possui processador AMD EPYC 7401 que possui 32 núcleos, e executa sistema operacional Linux Mint LMDE 5. Cada processo foi assinalado a um núcleo de processamento diferente.

Os sistemas testados variam de 3 a 32 réplicas, mais o cliente. As réplicas são inicializadas com o cliente. Foi medido o tempo de relógio necessário para o que o sistema decida 600 000 valores de 64 bytes, usando uma janela de pré-execução de 128 instâncias e 1024 valores simultâneos. Cada experimento foi realizado 20 vezes, sendo utilizados os maiores valores de decisões por segundo dos testes realizados. A Figura 6 apresenta os resultados de todos os experimentos executados.

Assim como em [11], a quantidade de decisões por segundo tende a cair conforme o número de réplicas aumenta. Isso não é diferente na estratégia proposta, já que o número de mensagens necessário para obter maioria cresce linearmente. Porém, a libHyperPaxos apresenta picos quando o número de réplicas é uma potência de dois, 2^k e $2^k - 1$. Isso acontece devido ao fato que o maior *cluster* possui a maioria necessária para fazer uma decisão. Já a estratégia proposta nesse trabalho apresenta curvas mais uniformes, com algumas ondulações resultantes do tamanhos dos *clusters* utilizados do vCube.

Obtemos essas curvas pois enviamos mensagens para uma maioria de imediato de forma paralela, sem precisar aguardar as respostas de cada um dos *clusters* como no algoritmo original. O algoritmo de difusão agressiva visa espelhar o caso do algoritmo original onde é necessário mandar mensagens apenas para o primeiro *cluster*, onde se encontram os seus picos de desempenho, mandando mensagens para todos os *clusters* necessários de uma vez.

É possível verificar que as duas curvas para as duas variações da estratégia proposta possuem comportamento

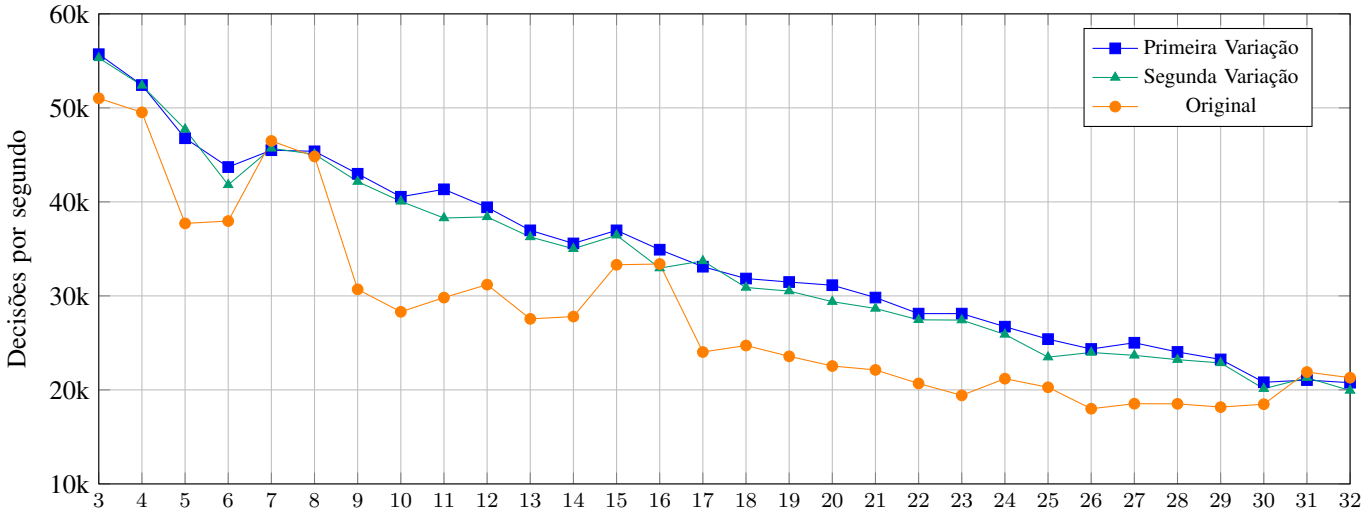


Figura 6: Valores decididos por segundo em relação ao número de réplicas.

muito parecido, pois as duas variações são idênticas em várias ocasiões. Porém, a segunda variação perde em alguns cenários. A primeira variação é mais agressiva, fazendo a difusão de uma vez para os maiores *clusters*. A segunda busca a combinação de *clusters* com o menor número de *acceptors* necessário. Desta forma, um tempo de resposta mais longo de um dos *acceptors* pode impactar o resultado final.

V. Trabalhos Relacionados

Regis e Mendizabal [9] apresentam diversas versões do algoritmo Paxos que fazem uso da paralelização para a decisão dos valores. No caso do Generalized Paxos, há um relaxamento do requisito de ordenação total entre valores decididos [24]. Quanto ao Egalitarian Paxos, existe a possibilidade de várias instâncias ficarem pendentes se não existirem conflitos entre os valores propostos [25]. E por fim, no Multi-Ring Paxos, é possível operar com diferentes grupos de processos que formam anéis distintos, permitindo uma maior vazão [26].

E no sentido da alteração da comunicação para permitir maior vazão, o PigPaxos é um protocolo que faz difusão distribuída através de processos retransmissores escolhidos aleatoriamente [27]. Esta proposta tem similaridade com o HyperPaxos, porém é limitada em escalabilidade pois aplica a distribuição em apenas um nível, enquanto que a topologia do vCube é multi-nível, mantendo propriedades logarítmicas [23]. Além disso, o PigPaxos não utiliza um detector de falhas, limitando a sua ação rápido em caso de potenciais falhas.

Além de algoritmos distribuídos de diagnóstico de redes representáveis por um grafo completo, como o vCube, outra família importante é dos algoritmos para redes de topologia arbitrária [28], [29]. Estes algoritmos se diferenciam principalmente na forma como difundem as informações de diagnóstico entre os processos. Por exemplo, em Duarte Jr e Mattos [30] é apresentado um algoritmo baseado em inundação, que paraleliza ao máximo a difusão, enquanto que em Duarte Jr e Cestari [31] é apresentado um algoritmo estritamente sequencial, com uma única mensagem de diagnóstico em transmissão a cada instante na rede.

Um outro modelo de diagnóstico é o baseado em comparações, que permite a detecção de falhas a partir da comparação de valores produzidos pela execução de uma determinada tarefa [32]. Aplicações incluem, por exemplo, a computação distribuída de integridade de dados replicados na rede [33]. E em diagnóstico, as estratégias baseadas em inteligência artificial também tem se mostrado promissoras, como em algoritmos de computação evolutiva [34] e de programação genética [35].

VI. Conclusão

Neste trabalho apresentamos uma estratégia para melhorar a vazão em termos de decisões por segundo para o algoritmo HyperPaxos, uma versão hierárquica do algoritmo Paxos sobre a topologia vCube. No algoritmo original, a difusão acontece de *cluster* em *cluster*, aguardando as respostas antes de prosseguir para o próximo *cluster*. A nossa proposta é que o difusor faça a transmissão em paralelo para o conjunto de *clusters* necessário para formar uma maioria. Duas variações foram implementadas e comparadas com a implementação original do HyperPaxos.

A primeira variação utiliza os primeiros maiores *clusters* com uma maioria de *acceptors* corretos, enquanto a segunda procura o melhor conjunto de *clusters* que juntos possuem uma maioria de *acceptors*. As duas variações apresentam desempenho mais uniforme que a estratégia originalmente proposta.

Trabalhos futuros incluem investigar mais estratégias para melhorar ainda mais o desempenho do HyperPaxos. Esta investigação inclui fazer modificações na função $c(i, s)$ para cenários de hipercubos imperfeitos na topologia vCube, e o uso de outras estratégias para seleção de *clusters*.

REFERÊNCIAS

[1] E. Brewer, "Spanner, TrueTime and the CAP Theorem," Google, Tech. Rep., 2017.

[2] Google, "Replication | Cloud Spanner | Google Cloud," https://cloud.google.com/spanner/docs/replication, 2023.

[3] J. Ellis, "Lightweight transactions in Cassandra 2.0," https://www.datastax.com/blog/lightweight-transactions-cassandra-20, 2013.

[4] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 335–350.

[5] Red Hat, "What is etcd?" https://www.redhat.com/en/topics/containers/what-is-etcd, 2019.

[6] L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998. [Online]. Available: https://www.microsoft.com/en-us/research/publication/part-time-parliament/

[7] ——, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, Dec. 2001.

[8] R. v. Renesse and D. Altinbuken, "Paxos Made Moderately Complex," *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, pp. 1–36, Feb. 2015. [Online]. Available: https://doi.org/10.1145/2673577

[9] S. Regis and O. M. Mendizabal, "Análise comparativa do algoritmo Paxos e suas variações," in *Anais do XXIII Workshop de Testes e Tolerância a Falhas*, May 2022, pp. 71–84. [Online]. Available: https://sol.sbc.org.br/index.php/wtf/article/view/21506

[10] P. Jalili Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: High-throughput atomic broadcast," *The Computer Journal*, vol. 60, no. 6, pp. 866–882, 2017.

[11] F. M. Kiotheka, D. R. Pereira, E. T. Camargo, and E. P. Duarte Jr, "HyperPaxos: Uma Versão Hierárquica do Algoritmo de Consenso Paxos," *41o Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)*, pp. 1–14, 2023.

[12] E. P. Duarte Jr, L. C. E. Bona, and V. K. Ruoso, "VCube: A Provably Scalable Distributed Diagnosis Algorithm," in *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, Nov. 2014, pp. 17–22. [Online]. Available: http://ieeexplore.ieee.org/document/7016729/

[13] M. Primi and D. Sciascia, "LibPaxos: Open-source Paxos," http://libpaxos.sourceforge.net/, 2013.

[14] S. Benz, "sambenz/URingPaxos: URingPaxos - A high throughput atomic multicast protocol," https://github.com/sambenz/URingPaxos, 2017.

[15] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[16] M. Hurfin, A. Mostefaoui, and M. Raynal, "Consensus in asynchronous systems where processes can crash and recover," in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, 1998, pp. 280–286.

[17] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*, 2nd ed. Springer Science & Business Media, 2011.

[18] E. P. Duarte Jr and T. Nanya, "A hierarchical adaptive distributed system-level diagnosis algorithm," *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 34–45, Jan. 1998. [Online]. Available: https://ieeexplore.ieee.org/document/656078/

[19] D. Jeanneau, L. A. Rodrigues, L. Arantes, and E. P. Duarte Jr., "An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detection," *Journal of the Brazilian Computer Society*, vol. 23, no. 1, p. 15, Dec. 2017. [Online]. Available: https://doi.org/10.1186/s13173-017-0064-9

[20] J. P. de Araujo, L. Arantes, E. P. Duarte Jr, L. A. Rodrigues, and P. Sens, "VCube-PS: A causal broadcast topic-based publish/subscribe system," *Journal of Parallel and Distributed Computing*, vol. 125, pp. 18–30, 2019.

[21] L. C. E. Bona, E. P. Duarte Jr, S. L. V. Mello, and K. V. O. Fonseca, "HyperBone: Uma Rede Overlay Baseada em Hipercubo Virtual sobre a Internet," in *XXIV Simpósio Brasileiro de Redes de Computadores*, 2006.

[22] L. A. Rodrigues, E. P. Duarte Jr, and L. Arantes, "A distributed k-mutual exclusion algorithm based on autonomic spanning trees," *Journal of Parallel and Distributed Computing*, vol. 115, pp. 41–55, 2018.

[23] F. M. Kiotheka and D. R. Pereira, "HyperPaxos / LibHyperPaxos - GitLab," https://gitlab.c3sl.ufpr.br/hyperpaxos/libhyperpaxos, 2022.

[24] L. Lamport, "Generalized Consensus and Paxos," Mar. 2005. [Online]. Available: https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/

[25] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in Egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 358–372. [Online]. Available: https://dl.acm.org/doi/10.1145/2517349.2517350

[26] P. J. Marandi, M. Primi, and F. Pedone, "Multi-Ring Paxos," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, Jun. 2012, pp. 1–12, iSSN: 2158-3927.

[27] A. Charapko, A. Ailijiang, and M. Demirbas, "PigPaxos: Devouring the Communication Bottlenecks in Distributed Consensus," in *Proceedings of the 2021 International Conference on Management of Data*, Jun. 2021, pp. 235–247. [Online]. Available: https://dl.acm.org/doi/10.1145/3448016.3452834

[28] E. P. Duarte Jr, A. Weber, and K. V. O. Fonseca, "Distributed Diagnosis of Dynamic Events in Partitionable Arbitrary Topology Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 8, pp. 1415–1426, Aug. 2012.

[29] E. P. Duarte Jr and A. Weber, "A distributed network connectivity algorithm," in *The 6th International Symposium on Autonomous Decentralized Systems, 2003. ISADS 2003.*, Apr. 2003, pp. 285–292.

[30] E. P. Duarte Jr and G. d. O. Mattos, "Diagnóstico em Redes de Topologia Arbitrária: Um Algoritmo Baseado em Inundação de Mensagens," in *Anais do Workshop de Testes e Tolerância a Falhas (WTF)*. SBC, Jul. 2000, pp. 82–87, iSSN: 2595-2684. [Online]. Available: https://sol.sbc.org.br/index.php/wtf/article/view/23479

[31] E. P. Duarte Jr and J. M. A. P. Cestari, "O Agente Chinês para Diagnóstico de Redes de Topologia Arbitrária," in *Anais do Workshop de Testes e Tolerância a Falhas (WTF)*. SBC, Jul. 2000, pp. 88–93, iSSN: 2595-2684. [Online]. Available: https://sol.sbc.org.br/index.php/wtf/article/view/23480

[32] R. P. Ziwich and E. P. Duarte Jr, "A Nearly Optimal Comparison-Based Diagnosis Algorithm for Systems of Arbitrary Topology," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3131–3143, Nov. 2016.

[33] R. Ziwich, E. Duarte, and L. Albini, "Distributed integrity checking for systems with replicated data," in *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, vol. 1, Jul. 2005, pp. 363–369 Vol. 1, iSSN: 1521-9097.

[34] B. T. Nassu, E. P. Duarte Jr, and A. T. Ramirez Pozo, "A comparison of evolutionary algorithms for system-level diagnosis," in *Proceedings of the 7th annual conference on genetic and evolutionary computation*, Jun. 2005, pp. 2053–2060. [Online]. Available: https://doi.org/10.1145/1068009.1068350

[35] E. P. Duarte, A. T. R. Pozo, and B. T. Nassu, "Fault diagnosis of multiprocessor systems based on genetic and estimation of distribution algorithms: a performance evaluation," *International Journal on Artificial Intelligence Tools*, vol. 19, no. 01, pp. 1–18, Feb. 2010. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S0218213010000017