A Caminho de Uma Alternativa Hierárquica para Implementação do Algoritmo de Consenso Paxos

Acacia Terra[1], Edson Tavares de Camargo[2], Elias P. Duarte Jr.[1]

[1] Universidade Federal do Paraná (UFPR) – Departamento de Informática
Caixa Postal 19018 – 81531-980 – Curitiba – PR – Brasil

[2]Universidade Tecnológica Federal do Paraná - Campus Toledo (UTFPR)
CEP: 85902-490 – Toledo – PR – Brasil

acterra@inf.ufpr.br, edson@utfpr.edu.br, elias@inf.ufpr.br

Resumo. *O Paxos está entre os mais importantes algoritmos de consenso, que permite que um conjunto de processos chegue em um acordo sobre um determinado valor proposto. Este trabalho propõe uma implementação hierárquica de uma instância do Paxos, na qual n processos acceptors estão organizados em uma topologia virtual denominada VCube. Um processo proposer se comunica utilizando um algoritmo de difusão sobre o VCube, para conseguir a maioria de votos para a decisão. As mensagens são propagadas de forma autonômica no VCube, que é escalável por definição, apresentando diversas propriedades logarítmicas, inclusive quando há processos falhos. Assume-se um sistema parcialmente síncrono com falhas de processos por parada, garantindo a segurança (safety), e a progressão (liveness), esta em condições de sincronia fraca. Resultados de simulação incluem uma comparação com o Ring Paxos, uma implementação do Paxos sobre uma rede de sobreposição virtual em anel.*

Abstract. *Paxos is one of the most important consensus algorithms. Consensus allows a set of processes to agree on a proposed value. This paper introduces a hierarchical implementation of a single Paxos instance. In the implementation, n acceptors are organized along a VCube, a virtual topology that is scalable by definition, presenting multiple logarithmic properties, even when there are faulty processes. A proposer communicates using a best-effort broadcast algorithm across the VCube to obtain votes from a majority of acceptors. Messages are autonomously propagated through the VCube. The proposed algorithm assumes an partially synchronous system with crash faults, and ensures safety and liveness (this property under weak synchrony assumptions). Simulation results include a comparison with Ring Paxos, an implementation of Paxos over a virtual ring topology.*

1. Introdução

Atingir o acordo (*agreement)* entre processos é um dos problemas mais fundamentais em computação distribuída. Aplicações distribuídas requerem com frequência que os processos cheguem a um acordo antes de executar ações específicas. Nesse contexto, o problema do consenso consiste em todos os processos, mesmo propondo valores distintos, concordarem com um único valor apesar de falhas [Fischer et al. 1985]. Um dos principais algoritmos de consenso é o Paxos [Lamport 1998, Lamport 2001], descrito a seguir.

O algoritmo Paxos possui características importantes: garante a propriedade de segurança do consenso (*safety*) em um sistema assíncrono e o progresso (*liveness*) apesar de falhas de processos. O algoritmo assume o modelo de falhas de parada com recuperação (*crash-recovery*). Os canais são não-confiáveis, assim mensagens podem ser perdidas. No Paxos, os processos, ou nodos, assumem os seguintes papéis distintos: *proposers*, *acceptors* e *learners*. Um processo pode, ao mesmo tempo, assumir os três papéis. Os *proposers* propõem um valor, os *acceptors* escolhem um valor e os *learners* aprendem o valor decidido. Um único processo pode assumir qualquer uma dessas funções e múltiplas funções simultaneamente. Paxos é ótimo em termos de resiliência [Lamport 2006b]: para tolerar f falhas ele requer $2f + 1$ *acceptors* – isto é, para assegurar o progresso, um quórum de $f + 1$ *acceptors* devem estar sem-falha. Para evitar um cenário em que *proposers* competem indefinidamente, um processo *coordenador* pode ser escolhido. O coordenador é geralmente responsável por enviar e receber mensagens para cada *acceptor*.

Além do algoritmo original, frequentemente chamado Paxos Clássico, diversas otimizações, versões e implementações do algoritmo foram propostas. O Multi-Paxos [Lamport 2001], que consiste em um único coordenador executar diversas instâncias do Paxos. No Fast Paxos [Lamport 2006a], um *proposer* envia sua proposta diretamente aos *acceptors*, ignorando o coordenador. O EPaxos [Moraru et al. 2013], discute a sobrecarga imposta ao coordenador (chamado de líder) e propõe uma solução em que os clientes podem escolher a cada passo para qual réplica enviar um comando. O Ring Paxos [Jalili Marandi et al. 2017] é um protocolo de difusão atômica (*atomic broadcast*) que se baseia no Paxos e assume que $f + 1$ nodos estão organizados em um anel lógico. Ao organizar os *acceptors* em um anel uni-direcional o número de mensagens que o coordenador precisa lidar é reduzido e a comunicação entre os *acceptors* se torna mais balanceada. No Ring Paxos o coordenador precisa aguardar a mensagem percorrer todo o anel para então seguir para a próxima fase.

Este trabalho propõe a implementação de uma instância do Paxos sobre uma topologia virtual denominada VCube [Duarte et al. 2014]. O algoritmo implementado é o Paxos Clássico. Processos *acceptors* estão organizados no VCube, agrupados em *clusters* progressivamente maiores, formando um hipercubo completo quando não há processos falhos. Em caso de falhas, os processos corretos se reconectam, mantendo diversas propriedades logarítmicas. No algoritmo proposto, um *proposer* se comunica utilizando a difusão de mensagens hierárquica no VCube. A difusão é autonômica, no sentido que se reorganiza na medida em que nodos falham e se recuperam. O *proposer* tira proveito da topologia para conseguir votos de uma maioria entre os n *acceptors* para a decisão. Resultados de simulação são apresentados, inclusive uma comparação com o Ring Paxos.

Este trabalho segue organizado da seguinte forma. A Seção 2 descreve o Paxos Clássico e algumas das suas variantes. A Seção 3 apresenta o VCube e o algoritmo de difusão de mensagens. A implementação proposta de uma instância do Paxos sobre o VCube é descrita na Seção 4. A implementação através de simulação e os resultados são descritos na Seção 5. A conclusão segue na Seção 6.

2. O Algoritmo Paxos e o Ring Paxos

O Paxos é um algoritmo de consenso tolerante a falhas proposto originalmente no contexto de replicação de máquina de estado [Lamport 1998, Lamport 2001]. O consenso é definido através de quatro propriedades: terminação, validade, integridade e acordo. A terminação é uma propriedade de progressão (*liveness*) e assegura que cada processo correto em algum momento decide por algum valor. Validade, integridade e acordo são propriedades de segurança (*safety*). A validade significa que se um processo decide por um valor *v*, então *v* foi proposto por algum processo. A propriedade de integridade determina que nenhum processo decide duas vezes e a propriedade de acordo afirma que dois processos corretos não decidem valores diferentes [Cachin et al. 2011].

Para garantir que diversas execuções do consenso sejam realizadas, o algoritmo executa sequências separadas de instâncias do Paxos. Cada instância corresponde a uma execução do consenso e está associada a um valor decidido. Uma instância do Paxos executa em duas fases distintas. A Figura 1 apresenta um cenário de execução em duas fases com dois *proposers* (P_0 e P_1), três *acceptors* (A_0, A_1 e A_2) e dois *learners* (L_0 e L_1). A seguir as Fases 1 e 2 são descritas.

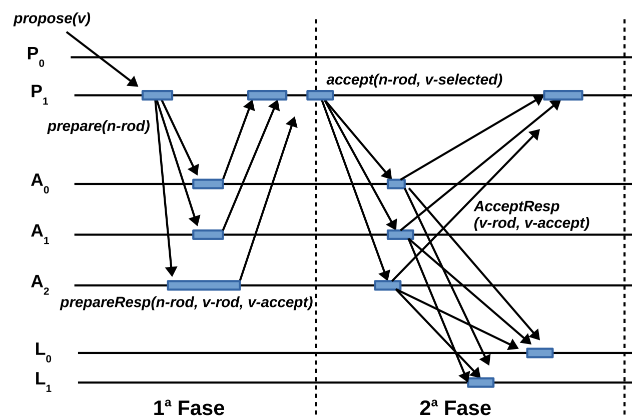


Figura 1. Paxos em duas fases

Fase 1. (a) A execução do consenso inicia quando *proposer* P_1 recebe um valor v. Então, P_1 seleciona uma proposta numerada *n-rod* (número de rodada) e envia uma mensagem *prepare-request* para os *acceptors* A_0, A_1 e A_2 (Figura 1, 1a Fase).

(b) Se um *acceptor* recebe um *prepare-request* com número *n-rod* maior do que qualquer *prepare-request* a que já tenha respondido, então responde ao *proposer* com uma mensagem *prepare-response* com a promessa de não aceitar nenhuma outra proposta com número menor do que *n-rod*. Se um *acceptor* já aceitou um valor (como explicado a seguir) para a instância atual (*v-accept*), irá retornar esse valor a P_1 juntamente com o número de rodada (*v-rod*) recebido quando o valor foi aceito. Caso não haja valor aceito, *v-accept* e *v-rod* são nulos. Quando P_1 recebe respostas de um quórum, ou seja, uma maioria de *acceptors*, o algoritmo segue para a segunda fase.

Fase 2. (a) P_1 seleciona um valor de acordo com a seguinte regra: se nenhum *acceptor* no quórum de respostas aceitou um valor, ou seja todos os *v-accept* são nulos, P_1 pode selecionar um novo valor para a instância (no caso, o valor v recebido na requisição inicial ao *proposer*). No entanto, se qualquer um dos *acceptors* retornou um

valor na primeira fase, o *proposer* escolhe entre os valores recebidos aquele com o maior número de rodada, ou seja, maior *v-rod*. O *proposer* então envia uma solicitação *accept* com o número de rodada usado na primeira fase e o valor escolhido (*v-selected*) para os *acceptors*.

(b) Se um *acceptor* recebe um *accept* de uma proposta numerada *n-rod*, ele aceita a proposta, a menos que já tenha respondido a um *prepare* com número ainda maior que *n-rod*. Na Figura 1, ao aceitar a proposta, os *acceptors* A_0, A_1 e A_2 respondem ao *accept* com uma mensagem *acceptResp* informando o número de rodada *n-rod* e o valor *v-accept* que foi aceito. Em seguida, os *acceptors* enviam o valor aceito para os *learners* (L_0 e L_1). Quando P_1 receber mensagens *acceptResp* de um quórum de *acceptors* aceita o valor, alcança-se o consenso para a instância.

Considere que múltiplos *proposers* executam simultaneamente o procedimento anterior para a mesma instância, por exemplo, P_0 e P_1 na Figura 1. Os *proposers* podem competir pelo maior número de rodada indefinidamente, e então nenhum *proposer* pode conseguir executar as duas fases do protocolo e alcançar o consenso. Para evitar esse cenário com *proposers* competindo indefinidamente, um processo *coordenador* pode ser escolhido. Neste caso, *proposers* submetem valores ao coordenador, que então executa a primeira e a segunda fase do protocolo. Se o coordenador falha, outro processo assume a sua função. O Paxos garante consistência apesar de *proposers* concorrentes e terminação na presença de um único coordenador.

2.1. Ring Paxos

O Ring Paxos [Jalili Marandi et al. 2017, Parisa Jalili Marandi et al. 2010] é uma versão do Paxos proposta no contexto de difusão atômica, que garante que mensagens são entregues na mesma ordem por todos os processos destinatários. A difusão atômica pode ser implementada usando uma sequência de execuções de consenso [Chandra and Toueg 1996]. No Ring Paxos os *acceptors* estão organizados em um anel direcional, o que reduz o número de mensagens no coordenador e permite um balanceamento da comunicação entre os *acceptors*. Para ordenar mensagens com eficiência em termos de *throughput*, a Fase 2 do Ring Paxos tira proveito do *pipeline* que se forma no anel. Quando um *acceptor* recebe uma mensagem da Fase 2, ele encaminha essa mensagem juntamente com sua própria resposta para o próximo *acceptor* no anel. Esse processo continua até que o coordenador seja alcançado. O *pipeline* permite o uso extensivo de *batches* de mensagens, o que aumenta o *throughput*. A seguir as duas versões do algoritmo são descritas sucintamente, uma baseada em multicast (M-Ring Paxos) e outra simplesmente usando mensagens unicast, mostradas na Figura 2.

A topologia do anel é configurada nos *acceptors* antes de iniciar a Fase 1. O M-Ring assume uma versão otimizada do Paxos, onde a primeira fase do consenso é executada antecipadamente e para muitas instâncias. A Fase 1 do M-Ring é a mesma do Paxos Clássico, não envolve o anel. É na segunda fase que as mensagens são propagadas no anel lógico unidirecional. O coordenador também é um *acceptor* e está no final do anel. A Fase 2 inicia com o coordenador enviando as mensagens denominadas Fase 2a usando *multicast*. Ao receber uma mensagem da Fase 2a, um *acceptor* verifica se pode votar para o valor proposto. Se sim, então armazena em uma variável chamada *v-vid* o valor proposto. O primeiro *acceptor* no anel envia uma mensagem da Fase 2b para seu

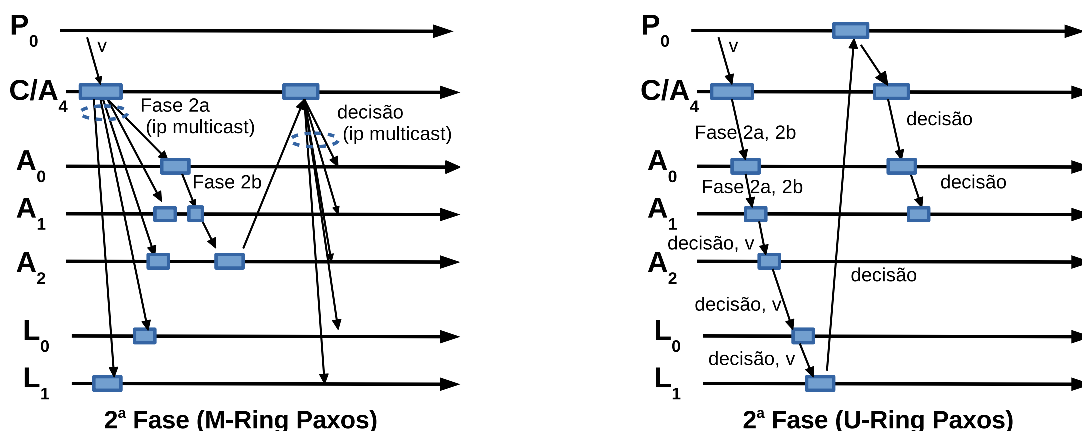


Figura 2. M-Ring Paxos e U-Ring Paxos

sucessor no anel. O próximo *acceptor* recebe a mensagem 2b e verifica se recebeu a mensagem previamente através do *multicast* da Fase 2a e o identificador *v-vid* para ver se se trata da mesma mensagem. Quando a mensagem atinge o último *acceptor*, ou seja, o coordenador, este envia por *multicast* a decisão através de uma nova mensagem.

No U-Ring Paxos, ao contrário do M-Ring, há mensagens tanto da Fase 1 como da Fase 2 que são propagadas no anel utilizando *unicast*. Tanto a Fase 1 como a Fase 2 são organizadas em etapas: a & b. Quando um *proposer* propõe um valor, ele é encaminhado ao longo do anel até chegar ao coordenador, que prosseguirá com a Fase 1, que é idêntica à do Paxos Classico. Quando o coordenador recebe mensagens da Fase 1b de um quorum, o coordenador verifica qual valor pode ser proposto e atribui um identificador exclusivo ao valor a ser proposto, como no M-Ring Paxos. O coordenador envia as mensagens da Fase 2a e 2b para o seu sucessor no anel. Da mesma forma que Paxos e M-Ring Paxos, o coordenador em U-Ring Paxos pode executar a Fase 1 antecipadamente, antes que um valor seja proposto, reduzindo a latência do protocolo.

Ao receber uma mensagem da Fase 2a/2b, um *acceptor* verifica se pode votar no valor proposto. Nesse caso, ele atualiza suas variáveis locais, como no Paxos Clássico. Se o *acceptor* não preceder o último *acceptor* no anel, ele envia a mensagem da Fase 2a/2b ao seu sucessor. Diferente do M-Ring, onde o coordenador é quem verifica se uma decisão pode ser tomada na instância, no U-Ring Paxos, é ao último *acceptor* que cabe a decisão. Após a decisão, que envia ao seu sucessor no anel. A decisão com o valor escolhido percorre então o anel completo, até chegar ao predecessor do *acceptor* que emitiu a decisão.

O Ring Paxos tolera tanto perda de mensagens quanto falhas de processos. De acordo com os autores, o modo mais simples de abordar os casos de falhas é trocar o modo de execução para o Paxos Clássico. Um coordenador que suspeita da falha de um ou mais *acceptors* simplesmente tenta contactar todos os *acceptors* para reunir um quórum. A solução reduz o *throughput*. Mensagens perdidas são resolvidas com retransmissão, se o coordenador não recebe repostas nas Fases 1a/2a, simplesmente reenvia as mensagens com um número de rodada maior. Eventualmente, o coordenador ou receberá uma resposta ou suspeitará de falha de um processo, que pode ser incorreta. O coordenador então estabelece um novo anel, excluindo o processo suspeito, e reexecuta a Fase 1.

Outras soluções para o caso de falhas também são propostas [Jalili Marandi et al. 2017].

3. Difusão de Mensagens sobre o VCube

Esta seção inicia descrevendo o VCube [Duarte et al. 2014] para em seguida apresentar a difusão de melhor-esforço no VCube [Rodrigues et al. 2014, Jeanneau et al. 2017].

3.1. VCube

O VCube, ou *Virtual Hypercube*, é uma topologia virtual que organiza os processos de um sistema distribuídos em um hipercubo quando todos os processos estão sem-falha. A topologia lógica é mantida pelo sistema de monitoramento subjacente ao VCube, que funciona como um detector de falhas. Os processos executam testes entre si para determinar seus estados. Um processo pode estar sem-falha (correto) ou falho, assume-se falhas *crash*. Neste trabalho assume-se um sistema assíncrono, portanto processos poder ser incorretamente suspeitos de terem falhado. No VCube, os processos são agrupados em *clusters* para a realização de testes. Os *clusters* são conjuntos de processos com tamanhos que são sempre potências de 2. A Figura 3 mostra um sistema com $n = 8$ processos e a organização dos *clusters* para esse sistema, neste exemplo é considerado que todos os processos estão sem-falha.

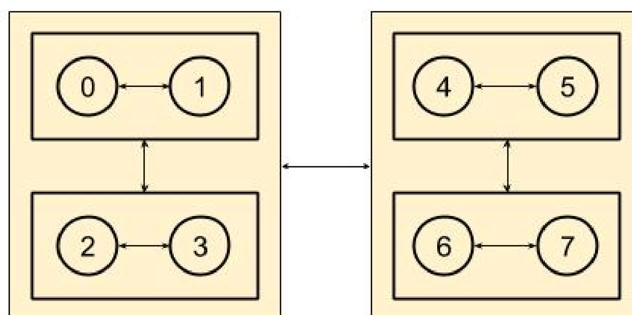


Figura 3. Organização dos *clusters* para um VCube com 8 processos.

Para cada processo o *cluster* de tamanho 1 corresponde ao processo cujo identificador tem todos os bits idênticos exceto o *bit* menos significativo, por exemplo os processos 4 (100) e 5 (101) são cada um *cluster* de tamanho 1 do outro. De cada processo sai um arco (que corresponde a um enlace virtual) para o processo do outro cluster com o qual comunica. Estes dois processos também formam, juntos, um *cluster* de tamanho 2 que tem arestas com outro *cluster* de igual tamanho. Por exemplo, o processo 4 (100) tem uma aresta com o processo 6 (110) e o processo 5 (101) tem uma aresta com o processo 7 (111). Esses 4 processos juntos formam um outro *cluster* de tamanho 4 que tem arestas com outro *cluster*, onde o processo 4 (100) tem aresta com o processo 0 (000), o processo 5 (101) com o processo 1 (001), o processo 6 (110) com o processo 2 (010) e o processo 7 (111) com o processo 3 (011). Cada processo pertence a $\log n$ *clusters* em um sistema com n processos.

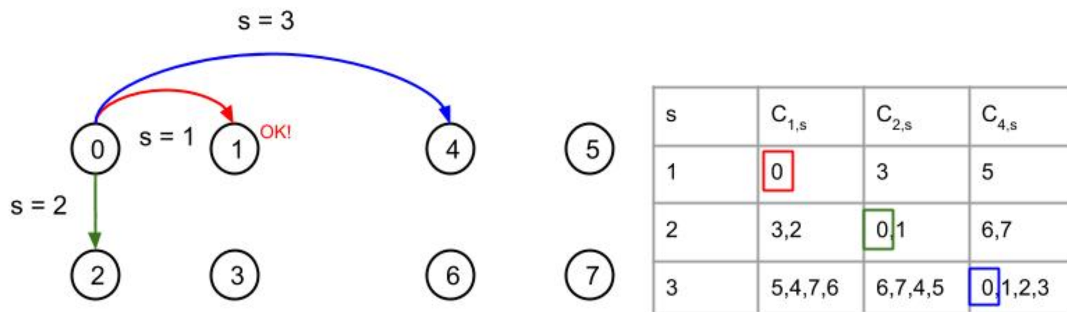
Em intervalos de tempo periódicos os processos executam testes em todos os seus $\log n$ *clusters*. A função `Cluster(i,s)`, definida a seguir, retorna uma lista ordenada de processos testados pelo processo i no *cluster* de índice s. O símbolo $\oplus$ indica uma operação binária de OU exclusivo (XOR), que é utilizado para inverter um dos *bits* do identificador de um determinado processo.

$$Clusteri, s = i \oplus 2^{s-1} \;||\; Clusteri \oplus 2^{s-1}, k \mid k = 1, ..., s-1$$

Um processo mantém informações sobre o estado de todos os outros processos do sistema. As informações de estado dos processos são armazenadas na forma de contadores armazenados em um vetor local $state[0...n-1]$. Quando um processo i executa um teste em um processo $j \in$ `Cluster(i,s)`, caso o processo j esteja sem-falha, então o processo i obtém novas informações de monitoramento a partir do processo j sobre os demais processos do sistema. Ao obter uma nova informação, o contador correspondente é incrementado. Quando os testes são realizados por todos os processos do sistema, uma rodada de testes é finalizada. No algoritmo do VCube é garantido que serão executados no máximo $n \log n$ testes por rodada de testes, sendo n a quantidade de processos do sistema.

3.2. Difusão de Mensagens no VCube

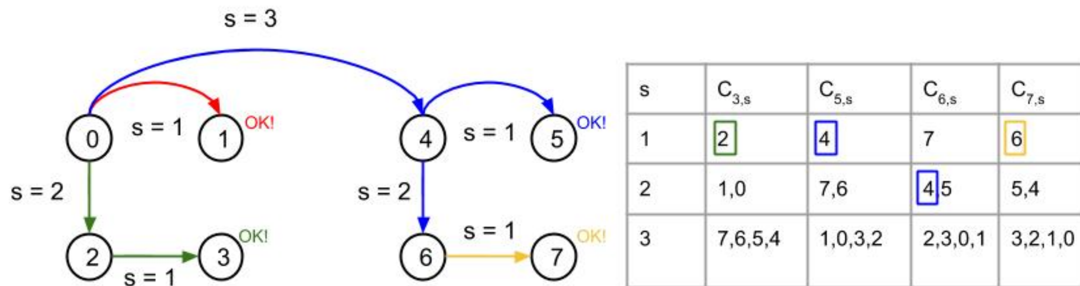
O algoritmo de difusão de melhor esforço para o VCube [Rodrigues et al. 2014] é considerado autonômico porque reconstrói a árvore dinamicamente à medida que processos falhos são detectados. Em [Rodrigues et al. 2014] o algoritmo proposto assume o modelo síncrono e os processos falham por parada (*crash*). Em versão mais recente desse trabalho, em [Jeanneau et al. 2017], um algoritmo de difusão atômica confiável é proposto para sistemas assíncronos sujeito a falhas por parada. As falhas são eventualmente detectadas por todos os processos corretos. O protocolo ainda tolera falsas suspeitas e mesmo assim consegue manter as propriedades logarítmicas do VCube.



s	$C_{1,s}$	$C_{2,s}$	$C_{4,s}$
1	0	3	5
2	3,2	0,1	6,7
3	5,4,7,6	6,7,4,5	0,1,2,3

Figura 4. Mensagens enviadas pelo processo 0.

O algoritmo [Rodrigues et al. 2014] inicia com o processo fonte gerando uma mensagem *m*, que contém dois parâmetros, o identificador da origem (processo que gerou a mensagem) e o *timestamp*, que é um contador local de mensagens transmitidas pelo processo. Juntos, os dois parâmetros permitem a identificação única de cada mensagem gerada. O processo fonte invoca o *broadcast*, que faz primeiramente uma entrega local da mensagem *m* e, em seguida, envia a mensagem para um conjunto de processos do VCube. Concretamente envia para os primeiros nodos sem-falha de cada um dos seus *clusters*. Ao receber uma mensagem, o processo verifica se é uma mensagem nova, comparando os *timestamps* da mensagem recebida e da última mensagem anteriormente recebida do mesmo processo. Se *m* for uma nova mensagem, então ela é entregue para a aplicação. O processo retransmite a mensagem para outros processos, de forma descrita posteriormente.



s	$C_{3,s}$	$C_{5,s}$	$C_{6,s}$	$C_{7,s}$
1	2	4	7	6
2	1,0	7,6	4,5	5,4
3	7,6,5,4	1,0,3,2	2,3,0,1	3,2,1,0

Figura 5. Mensagens enviadas pelos demais processos.

As Figuras 4 e 5 mostram o caminho percorrido por uma mensagem *m* gerada e enviada pelo processo 0 em um sistema com 8 processos onde todos estão sem-falha. O processo 0 verifica de acordo com sua função `Cluster(i,s)` para quais processos deve enviar a mensagem. Envia para o primeiro vizinho sem-falha do primeiro `Cluster(0,1)`, o processo 1, depois para o primeiro vizinho sem-falha do `Cluster(0,2)`, depois `Cluster(0,3)`, e assim por diante, até `Cluster(0,log n)`. O processo que recebe a mensagem repassa para os seus vizinhos nos seus *clusters*. O número de mensagens depende do *cluster* em que o processo está, como descrito a seguir. Quando todos os processos corretos entregam a mensagem, o *broadcast* termina.

Cada processo, ao receber uma mensagem, verifica a qual *cluster* ele pertence para o processo de quem recebeu a mensagem. O valor *cluster*-1 indica para quantos outros processos a mensagem deve ser enviada por aquele processo. Por exemplo, se ele pertence ao *cluster* 3 do processo que enviou a mensagem, então precisa enviar a mensagem para outros dois *clusters*, os primeiros processos sem-falha dos seus *clusters* 1 e 2, respectivamente. Se ele pertence ao *cluster* 1 do processo que enviou a mensagem, então é um nodo folha da árvore e não é preciso repassar a mensagem para nenhum outro processo.

4. Uma Instância do Paxos sobre o VCube

A seguir é apresentado o algoritmo que implementa uma instância do Paxos tirando proveito da topologia virtual hierárquica do VCube. O modelo de sistema é parcialmente síncrono com GST (*Global Stabilization Time*), isto é, após um período de instabilidade o sistema passa a respeitar limites de tempo de transmissão de mensagens e processamento. Os canais de comunicação são confiáveis.

O algoritmo tem as duas fases do Paxos. A Fase 1 é descrita abaixo. Nessa fase, o *proposer* dissemina o *prepare-request* no VCube com o objetivo de obter uma maioria de *prepare-responses*. Após receber respostas de uma maioria de *acceptors*, o *proposer* utiliza o algoritmo de difusão de melhor esforço apresentado na Seção 3 (com algumas modificações) para enviar para os *acceptors* o *accept-request* com o maior número de rodada recebido na Fase 1 e o valor correspondente. Se entre a Fase 1 e a Fase 2 algum *acceptor* da maioria recebida falhar, uma nova Fase 1 deve ser executada. Ao final, os *learners* aprendem a decisão.

A Fase 1 do algoritmo proposto tira proveito da organização dos processos em *clusters* para acelerar sua execução. O *proposer* é também *acceptor*, desta maneira são necessários $n/2$ *prepare-responses* para obter uma maioria. O *proposer* envia o *prepare-request* apenas para o maior *cluster*, de índice $\log n$ que consiste de $n/2$ nodos. Se nenhum nodo deste *cluster* estiver falho ou incorretamente suspeito, o *proposer* já terá a maioria. Caso contrário, segue para o *cluster* de índice $\log n - 1$, e assim por diante, até conseguir $n/2$ *prepare-responses*. Neste ponto a Fase 1 está completa.

O algoritmo `VCubeProposerFase1` executado pelo *proposer* é mostrado como Algoritmo 1. O algoritmo `VCubeAcceptorsFase1` executado pelos *acceptors* é mostrado como Algoritmo 2.

No algoritmo `VCubeProposerFase1` o *proposer* envia o *prepare-request* para o maior *cluster* de índice $logN$. O número de respostas inicialmente é 1, apenas a do próprio *proposer*. No algoritmo `VCubeAcceptorFase1` os *acceptors* vão concatenando seus *prepare-responses*. Ao receber o *prepare-request*, o *acceptor* verifica se o número de rodada recebido é maior que aquele que mantém. Neste caso adota o número de rodada recebido, e concatena seu *prepare-response* na mensagem recebida, re-encaminhando para seus *clusters*, alterando os índices de *clusters* adequadamente. O algoritmo usado para determinar quais processos se comunicam é exatamente a difusão descrita na Seção 3, com uma diferença: um nodo folha envia a mensagem que concatena todos os *prepare-responses* para o *proposer*. O *proposer* aguarda respostas de todos os nodos folha esperados. Como o VCube funciona como um detector de falhas o *proposer* sabe quantas mensagens deve receber. Caso haja uma falsa suspeita, um processo correto pode ser indevidamente considerado falho. Neste caso, a única consequência é que o *proposer* não vai considerar as respostas deste processo. O *proposer* então processa as mensagens concatenadas, determinando o maior número de rodada e se há um valor já aceito. Se o número de *prepare-responses* recebido for maior que $n/2$ passa para a Fase 2.

Algoritmo 1: VCUBEPROPOSERFASE1

1. *Cluster* $\leftarrow \log n$ `// Maior cluster`
2. *NumPrepResp* $\leftarrow 1$ `//` *Proposer* `já tem seu prepare-response`
3. **repita**
4. VCUBEACCEPTORSFASE1(*Proposer, NumeroRodada, Valor, Cluster, NULL*)
5. Aguarde mensagens dos processos folhas do *Cluster* com as prepare-responses concatenadas
6. Selecione o maior *NumeroRodada* e o *Valor* correspondente, se houver
7. Atualize *NumPrepResp* com o número de prepare-responses recebidos
8. *Cluster* $\leftarrow$ *Cluster* $- 1$
9. **até** ($NumPrepResp > n/2$) **ou** ($Cluster = 0$)

Um *acceptor* executando algoritmo `VCubeAcceptorsFase1` recebe cinco parâmetros: o nodo que está fazendo o broadcast, o número da rodada, o valor, o seu próprio índice do *cluster* e as *prepare responses* concatenadas. Uma chamada ao `VCubeAcceptorsFase1` retorna o número de confirmações obtidas naquele *cluster*. Este algoritmo é apresentado a seguir.

Algoritmo 2: VCUBEACCEPTORSFASE1

Entrada: *Proposer*, *NumeroRodada*, *Valor*, *Cluster*, *mensagens-concatenadas*

1 **se** *numRodadaLocal* $<$ *NumeroRodada* **então**
2 | *numRodadaLocal* = *NumeroRodada*
3 **fim**
4 Concatena prepare response a *mensagens-concatenadas*
5 **enquanto** *Cluster* > 1 **faça**
6 | VCUBEACCEPTORSFASE1(*Proposer, NumeroRodada, Valor, Cluster* $- 1$)
7 **fim**
8 **se** *Cluster* $= 1$ **então**
9 | envia *mensagens-concatenadas* para o *Proposer*
10 **fim**

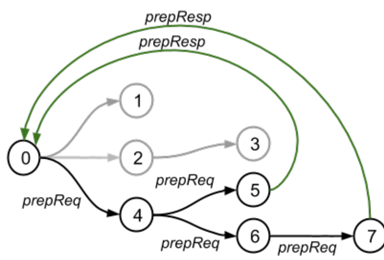
Antes do GST, é possível que nodos sem-falha sejam incorretamente suspeitos de terem falhado e por isso não recebem o *prepare-request*. Neste algoritmo se o número de falsas suspeitas for menor que $n/2$, a Fase 1 não é afetada. Por outro lado, com $n/2$ ou mais falsas suspeitas o *proposer* não consegue a maioria, e a progressão (*liveness*) do algoritmo fica comprometida. Por lado, a segurança (*safety*) não é comprometida: a decisão ocorre uma única vez devido à maioria de *acceptors* necessária.

A Figura 6 apresenta o caminho que as mensagens percorrem durante a Fase 1 do Paxos utilizando os algoritmos apresentados nesta seção. Nas figuras, a representação é a seguinte. Processos sem-falha que recebem mensagem *prepare-request* são representados por círculos pretos, enquanto processos sem-falha que não recebem mensagem *prepare-request* são representados por círculos cinzas, os processos considerados falhos são representados por círculos vermelhos. Mensagens: *prepare-request* é representada por setas pretas, *prepare-response* por setas verdes.

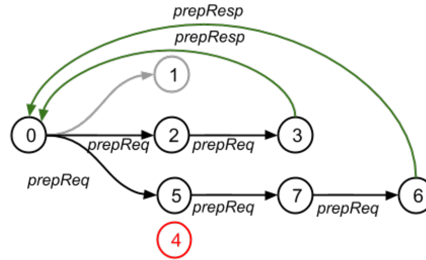
São apresentados três exemplos, todos com $n = 8$ e consideram o processo 0 como *proposer* coordenador. Na Figura 6a todos os processos estão sem-falha e o *proposer* obtém a maioria enviando a *prepare-request* apenas para seu maior *cluster* ($\log n$). Os *acceptors* folhas 5 e 7 enviam para o *proposer* a mensagem *prepare-response*. Na Figura 6b o processo 4 está falho, então não basta enviar a *prepare-request* apenas para o maior *cluster*, assim o processo 0 envia também para o processo 2, que está no *cluster* $\log n - 1$. Neste ponto, a quantidade de respostas já é suficiente para a maioria, mas como o *acceptor* 2 não é folha, ainda encaminha a mensagem para o *acceptor* 3, que também responde a mensagem e, como é folha, envia o *prepare response* para o *proposer*. Na Figura 6c há $n/2 - 1$ processos falhos, o número máximo de falhas suportadas pelo Paxos. Nela, os processos 2, 4 e 7 estão falhos, então o *proposer* envia a *prepare request* para todos os *clusters* do VCube até atingir a maioria e recebe três *prepare responses*, sendo neste caso uma de cada *cluster*.

5. Implementação e Resultados

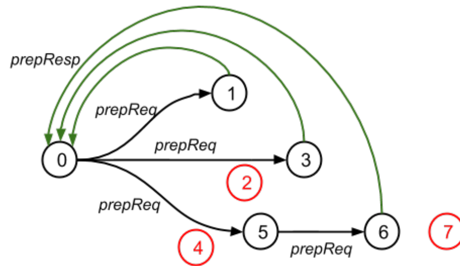
Esta seção apresenta a implementação através de simulação do algoritmo proposto, bem como resultados obtidos, incluindo uma comparação com o Ring Paxos.



(a) Exemplo com todos os processos sem-falha e $n = 8$.



(b) Exemplo com um processo falho (4) e $n = 8$.



(c) Exemplo com três processos falhos (2, 4, 7) e $n = 8$.

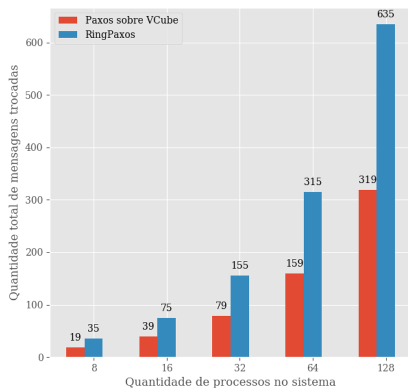
Figura 6. Exemplos de execução do algoritmo para a Fase 1 do Paxos.

5.1. Implementação

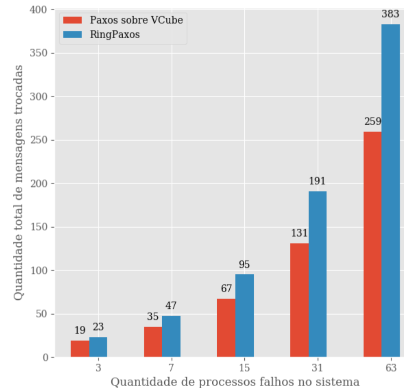
A implementação do Paxos sobre o VCube foi feita utilizando as linguagens de programação C/C++ e a biblioteca de simulação SMPL (*Simulation Programming Language*) [MacDougall 1987].

Ao fazer um *propose*, cada proposta deve ter identificador único. Para garantir que diferentes propostas geradas em nodos diferentes tenham identificados únicos, é utilizada a seguinte alternativa. Inicialmente, a primeira proposta de um processo é o identificador do próprio nodo: $propo_id \leftarrow node_id$. A cada nova proposta, é feito o cálculo $propo_id \leftarrow propo_id + n$. A mensagem contendo número de rodada (*prepare request*) é enviada através da difusão de melhor esforço para os processos *acceptors* no sistema. A difusão também foi configurada como um evento na simulação. Primeiramente o processo origem da mensagem (o *proposer*) entrega (*deliver*) a mensagem caso ele seja também um *acceptor*, em seguida envia para os processos *acceptors* do sistema, que estão organizados no VCube, começando pelo seu maior *cluster*.

O processo *acceptor*, ao receber a mensagem, entrega para a aplicação e acrescenta o seu identificador a um vetor de *accepts*, além de atualizar uma variável que representa o maior número de rodada recebido e aceito pelo processo. Quando o processo é um nodo folha na árvore do VCube, envia para o processo origem da proposta uma mensagem do tipo *prepare response* contendo a contagem de *accepts*, além do número da rodada $propo_id$. No caso de o processo já ter aceito um valor v, esse valor é retornado no *prepare response* no campo $propo_val = v$, caso ainda não tenha aceito nenhum valor o campo é preenchido com $propo_val = -1$.



(a) Cenário com todos os processos sem-falha.



(b) Cenário com $n/2 - 1$ processos falhos.

Figura 7. Quantidade total de mensagens trocadas em cada algoritmo.

Quando o processo *proposer* recebe um *prepare-response*, ele verifica se a quantidade de *accepts* recebidos até o momento são suficientes para avançar para a próxima fase do Paxos. Quando a quantidade de *accepts* recebidos é $qtd_accepts >= n/2$, sendo n a quantidade de *acceptors* no sistema, o processo interrompe a propagação das mensagens e avança para a segunda fase do Paxos através do escalonamento de um evento.

No início da segunda fase, é gerada uma mensagem do tipo *accept-request*, contendo o número da proposta e também o valor associado, além dos demais parâmetros que são gerados de forma automática, como por exemplo o *timestamp*. A mensagem é disseminada através da topologia VCube para os *acceptors* da mesma forma que durante a Fase 1. Quando um processo recebe o *accept-request*, verifica se já aceitou alguma outra proposta com número maior e também se já aceitou algum valor anteriormente (se $nodo_reg = -1$, então não aceitou nenhum valor ainda). No caso de não ter aceitado um valor, o processo aceita e altera o seu registrador, atualizando para o valor que foi aceito. A mensagem percorre a topologia e, assim como na Fase 1, quando um processo é folha na árvore, envia para a origem a contagem de processos que aceitaram o valor até então. Com isso, o consenso é finalizado.

5.2. Resultados Experimentais

Para avaliar o impacto do nosso algoritmo, foi também implementada uma versão do Ring Paxos. Ao final de cada simulação foram coletadas informações sobre o número de mensagens trocadas. A quantidade total de mensagens trocadas até atingir o consenso, tanto para um cenário sem-falhas quanto para um cenário com falhas, pode ser vista na Figura 7. Informações mais detalhadas são descritas adiante.

Foram realizados experimentos com número de processos no sistema igual a 8, 16, 32, 64 e 128. O processo coordenador é sempre o processo de identificador igual a $n/2-1$, ou seja, o processo que está no centro da topologia. A escolha do coordenador não interfere nos resultados. Para os cenários com falhas, foram programadas falhas para os nodos de número ímpar do sistema, exceto o processo de número $n/2-1$ (o coordenador). Assim, totalizam-se $n/2 - 1$ falhas, o número máximo de falhas suportadas, pois ainda

resta uma maioria de processos sem-falha no sistema. Em todos os cenários, todos os processos são *acceptors*, inclusive o processo coordenador (que é também *proposer*).

Para os experimentos, o Ring Paxos utiliza um algoritmo tradicional de difusão de melhor esforço durante a primeira fase do algoritmo. Na segunda fase, o Ring Paxos dispõe os processos em um anel, de forma que são enviadas apenas n mensagens durante a segunda fase, sendo n o número de processos sem-falha no sistema. Contudo, após o consenso terminar, são enviadas mensagens contendo a decisão para todos os processos sem-falha. O Paxos sobre o VCube por sua vez apresenta uma redução no número de mensagens para as duas fases do algoritmo, quando a mensagem precisa chegar apenas até uma maioria de *acceptors*. A quantidade de cada um dos tipos de mensagens enviadas por cada um dos algoritmos pode ser observada a seguir. As Tabela 1 e 2 consideram um cenário com todos os processos sem-falha e um cenário com $n/2 - 1$ falhas, respectivamente. No caso do VCube, consideramos que o número de respostas de *acceptors* é um limite teórico correspondente ao caso em que todos são folhas. Na verdade o número de folhas será sempre menor que $n/2$.

Algoritmo/Qtd msg	***prepare req***	***prepare resp***	***accept req***	***accept resp***	**decisão**
Ring Paxos	$n-1$	$n-1$	$n-1$	1	$n-1$
Paxos sobre VCube	$n/2$	$n/2$	$n/2$	$n/2$	$n/2$

Tabela 1. Quantidade de mensagens de cada algoritmo para cenário sem-falhas.

Algoritmo/Qtd msg	***prepare req***	***prepare resp***	***accept req***	***accept resp***	**decisão**
Ring Paxos	$n-1$	$n/2$	$n/2$	1	$n/2$
Paxos sobre VCube	$n/2$	$n/2$	$n/2$	$n/2$	$n/2$

Tabela 2. Quantidade de mensagens em cada algoritmo para cenário com $n/2-1$ falhas.

Pode-se notar que em ambos os cenários, o algoritmo Paxos sobre o VCube tem vantagem em relação ao número de mensagens trocadas. Isso porque, na primeira fase, envia mensagem apenas até obter uma maioria e em ambas as fases as mensagens respostas são enviadas para a origem apenas quando o processo é folha na árvore.

6. Conclusão

Neste trabalho foi apresentado um algoritmo hierárquico que implementa o Paxos na topologia virtual VCube. O Paxos é um dos algoritmos mais importantes de consenso, desta forma reduzir seu custo pode trazer benefícios para o grande número de aplicações distribuídas que são baseadas neste algoritmo. O VCube, e portanto o Paxos sobre VCube, são escaláveis por definição, apresentando diversas propriedades logarítmicas. O algoritmo foi especificado e implementado através de simulação, resultados mostram o ganho no número de mensagens utilizadas em comparação com o Ring Paxos.

A principal vantagem do Ring Paxos é dispor os processos em um anel e, quando há várias instâncias executando, o *pipeline* gera um *throughput* elevado. Como os resultados obtidos para esse trabalho são para apenas uma instância dos algoritmos, fica como trabalho futuro a extensão para permitir múltiplas instâncias consecutivas.

Referências

Cachin, C., Guerraoui, R., and Rodrigues, L. (2011). *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267.

Duarte, E. P., Bona, L. C. E., and Ruoso, V. K. (2014). Vcube: A provably scalable distributed diagnosis algorithm. In *2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 17–22.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

Jalili Marandi, P., Primi, M., Schiper, N., and Pedone, F. (2017). Ring Paxos: High-Throughput Atomic Broadcast†. *The Computer Journal*, 60(6):866–882.

Jeanneau, D., Rodrigues, L. A., Arantes, L., and Jr., E. P. D. (2017). An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detection. *J. Braz. Comp. Soc.*, 23(1):15:1–15:14.

Lamport (2001). Paxos made simple. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 32.

Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169.

Lamport, L. (2006a). Fast paxos. *Distributed Computing*, 19(2):79–103.

Lamport, L. (2006b). Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125.

MacDougall, M. H. (1987). *Simulating Computer Systems: Techniques and Tools*. MIT Press, Cambridge, MA, USA.

Moraru, I., Andersen, D. G., and Kaminsky, M. (2013). There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA. ACM.

Parisa Jalili Marandi, Primi, M., Schiper, N., and Pedone, F. (2010). Ring paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 527–536.

Rodrigues, L. A., Duarte Jr, E. P., and Arantes, L. (2014). Árvores geradoras mínimas distribuídas e autonômicas. In *Anais do 32º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos – SBRC 2014*, pages 1–14.