Eleição de Líder com o vCube para Sistemas Distribuídos *Crash-Recovery* no Modelo GST

Luiz A. Rodrigues[1], Allan Freitas[2] e Elias P. Duarte Jr.[3]

[1]Universidade Estadual do Oeste do Paraná (UNIOESTE)
Programa de Pós-Graduação em Ciência da Computação (PPGComp)
Cascavel – Paraná, Brasil

[2]Instituto Federal da Bahia (IFBA)
Programa de Pós-Graduação em Engenharia de Sistemas e Produtos (PPGESP)
Campus de Salvador, Bahia, Brasil

[3]Universidade Federal do Paraná (UFPR)
Departamento de Informática (DInf)
Curitiba – Paraná, Brasil

Luiz.Rodrigues@unioeste.br, allan@ifba.edu.br, elias@inf.ufpr.br

Resumo. *A eleição de líder é um dos problemas fundamentais de sistemas distribuídos, com um número enorme de aplicações. Neste artigo apresentamos um algoritmo hierárquico e autonômico para eleição de líder baseado na topologia virtual vCube. O vCube é redefinido com um detector de falhas para o modelo crash-recovery, considerando que os processos têm acesso a memória não volátil. O algoritmo elege como líder aquele processo correto entre os mais estáveis (que falharam/recuperaram menos vezes) que tem menor identificador. A correção do algoritmo é discutida, em termos das propriedades de precisão e acordo após um tempo. O algoritmo foi implementado com simulação e resultados comparando com a versão força-bruta demonstram sua escalabilidade.*

1. Introdução

A eleição de líder é um problema fundamental na área de sistemas distribuídos, sendo utilizada em uma variedade de contextos, desde protocolos de redes de computadores até aplicações distribuídas. Os ambientes em que o líder é necessário variam desde sistemas operacionais distribuídos em que um líder pode participar, por exemplo, da gerência de recursos distribuídos até a computação em nuvem. O líder [Santoro 2006] pode desempenhar funções específicas, como a tomada de decisões, ou funcionar como um coordenador, ou ainda ser uma referência para os demais processos, permitindo a coordenação de tarefas e garantindo a consistência, a execução confiável e o desempenho do sistema.

Informalmente, a eleição de líder visa garantir que todos os processos escolham um processo correto como líder [Cachin et al. 2011]. Além disso: todos os processos devem escolher o *mesmo* líder. Quando o modelo *crash* tradicional é adotado, a eleição de líder corresponde a um detector de falhas [Duarte Jr et al. 2023] que retorna, ao invés da lista de processos suspeitos, um único processo correto que é o líder. No modelo *crash-recovery*, processos falham e recuperam, podendo ser inclusive realmente instáveis, por exemplo, com falhas intermitentes. O monitoramento de sistemas instáveis é um

problema relevante [Duarte et al. 2010]. No modelo *crash-recovery* deve-se eleger um líder que seja não apenas correto, mas também o mais estável.

Neste trabalho, propomos uma solução hierárquica para o problema da eleição de líder utilizando a topologia virtual vCube [Bona et al. 2006]. O vCube é um hipercubo virtual quando todos os processos estão corretos e o número de processos é uma potência de 2. Sua principal característica é, no entanto, que na medida em que processos falham e se recuperam o vCube se reorganiza de forma autonômica, mantendo diversas propriedades logarítmicas [Duarte et al. 2009]. O vCube foi definido para oferecer um serviço de detecção de falhas. Recentemente, foi proposta a implementação de um detector de falhas $\diamond P$ sobre o vCube [Stein et al. 2023] considerando o modelo parcialmente síncrono GST (Global Stabilization Time). No modelo GST o sistema começa sem qualquer garantia temporal, mas após um instante – dito GST – passa a se comportar como síncrono.

O presente trabalho tem duas contribuições principais. A primeira delas é que se trata da primeira especificação do vCube dentro do modelo de falhas *crash-recovery*. Apesar de que nas especificações originais o vCube sempre permitiu a recuperação de processos, implicitamente esta recuperação implicava na perda total de todo estado interno do processo falho, exceto o identificador. Desta forma, um processo que recupera é como um processo novo, que deve obter as informações necessárias a partir de outros processos corretos que não falharam. No presente trabalho, cada processo que executa o vCube tem memória secundária não volátil, e é capaz de manter informações de estado, que podem ser recuperadas quando o processo recupera após ter sofrido uma falha.

A outra contribuição do trabalho é justamente o algoritmo hierárquico de eleição de líder sobre o vCube. Cada processo mantém em memória secundária não volátil seu número de *encarnações*. Quando processo inicia a execução do algoritmo está na sua primeira encarnação. A cada vez que falha e recupera o número de encarnações é incrementado de uma unidade. Desta forma, o número de encarnações reflete quantas vezes falhou e recuperou. A eleição de líder é proposta no modelo parcialmente síncrono GST, portanto falsas suspeitas podem acontecer, mas não após o GST. O objetivo é eleger o líder mais "estável", isto é, com menos encarnações. Assim, o critério de eleição é: entre os processos de menor número de encarnações, o líder é aquele com menor identificador.

A corretude do algoritmo é discutida em termos das propriedades clássicas da eleição de líder: precisão eventual e acordo eventual. Informalmente, a precisão eventual determina que todo processo correto eleja como líder um processo correto. O acordo eventual determina que não há dois processos corretos que elejam como líderes dois processos distintos. O algoritmo foi implementado através de simulação. Resultados mostram especialmente a redução significativa no número de mensagens para a eleição em comparação com a versão força-bruta. Também reduz o tempo de execução das rodadas de teste, embora o vCube utilize mais rodadas para a detecção de um evento e convergência do líder único eleito.

O texto segue da seguinte forma. A Seção 2 descreve o modelo do sistema utilizado. Seção 3 apresenta o trabalhos relacionados. O algoritmo proposto é apresentado na Seção 4. Os resultados são discutidos na Seção 5. A Seção 6 conclui o trabalho.

2. Modelo do Sistema

Neste trabalho, um sistema distribuído é um conjunto de processos que se comunicam através de troca de mensagens. A composição do sistema é estática, isto é, consiste de N processos, que tem identificadores de 0 a $N-1$. Desta forma, o sistema pode ser representado como o conjunto $\Pi = \{p_0, p_1, ..., p_{N-1}\}$; sendo também possível usar apenas os identificadores dos processos: $\Pi = \{0, 1, ..., N-1\}$. Qualquer par de processos pode se comunicar diretamente entre si, sem necessitar de intermediários, isto é, trata-se de um sistema totalmente conectado, representável por um grafo completo. Os processos podem falhar e recuperar, e o modelo de falhas adotado é o *crash-recovery*. A característica mais importante deste modelo é o fato dos processos manterem informações de estado em memória secundária, não volátil. Estas informações são utilizadas quando processos se recuperam de uma falha.

Os enlaces são perfeitos, assim uma mensagem transmitida por um processo é efetivamente entregue pelo destinatário após um tempo, além disso não há duplicação. Um processo pode estar em um de dois estados possíveis: *correto* ou *falho*. Um processo no estado *falho* não responde a qualquer estímulo. O modelo temporal adotado é parcialmente síncrono, o GST em que o sistema começa sem qualquer garantia temporal e, após um instante de tempo também chamado de GST, passa a se comportar como síncrono. As operações de envio e recebimento de mensagens são atômicas e os enlaces são confiáveis.

A eleição de líder é uma abstração que tem equivalência com os detectores de falhas, definidos a seguir. Um detector de falhas pode ser visto como um conjunto de N módulos de detecção de falhas, cada um em um processo distinto do sistema. O detector de falhas pode ser invocado a qualquer momento pelo processo correspondente, e retorna como saída a lista de processos *suspeitos* de terem falhado. Um detector de falhas pode também ser visto como um sistema de monitoramento do sistema distribuído, que visa determinar quais processos estão *corretos* e quais estão *suspeitos* de terem falhado. Destaca-se que o detector jamais se refere a um processo como propriamente *falho*, pois prevê a possibilidade de engano. Concretamente, um processo correto porém mais lento do que esperado pode ser enganosamente detectado como falho. Desta forma, ao utilizar a palavra “suspeito” ao invés da palavra “falho” um detector define precisamente a classificação que produz dos processos monitorados.

No artigo original em que os detectores de falhas foram propostos [Chandra and Toueg 1996], são definidas duas propriedades: completude e precisão. Informalmente, a completude se refere ao fato de que processos falhos vão ser suspeitados. A precisão se refere ao fato de que processos sem falha não vão ser (incorretamente) suspeitados pelo detector. Na prática, a completude é trivial de se obter em sistemas distribuídos reais. Se um processo sofre uma falha *crash*, não responde a qualquer estímulo e será detectado como tal assim que o procedimento de monitoramento seguinte ao evento de falha for executado. A precisão, por outro lado, é impossível de garantir: se a carga de um processo ou da própria infraestrutura de computação/comunicação ficar elevada por algum motivo a ponto de reduzir significativamente o desempenho de um processo correto, ele pode ser confundido como falho. Assim, a precisão pode acontecer ou não, dependendo da execução do detector.

Os detectores de falhas podem ser classificados em dois tipos, de acordo com a forma com que um processo é monitorado: *pull* ou *push*. No modelo *push* cada processo

correto envia periodicamente mensagens de *heartbeat* aos demais processos, informando que está em execução. No modelo *pull* os processos enviam uma resposta a partir do recebimento de um estímulo, que pode ser considerado um teste. Testes são executados periodicamente em um intervalo de testes. Os processos não tem acesso a um relógio global, nem têm seus relógios sincronizados, assim os intervalos de teste podem ser diferentes entre os processos (apesar de nominalmente idênticos). Para captar essa diferença nos intervalos de testes, é definido o conceito de rodada de testes. Em uma rodada de testes todos os processos corretos executam seus testes assinalados. O teste pode ser tão simples quanto um *heartbeat-request* que deve ser respondido por um *heartbeat-reply*, ou consistir da execução de uma bateria de procedimentos para a verificação de diversos aspectos.

Em [Duarte Jr et al. 2023] são apresentadas três estratégias para a implementação de detectores de falhas *pull*. A primeira é a mais tradicional, denominada força-bruta, na qual todos os processos testam todos os demais processos periodicamente. A segunda estratégia consiste do detector de falhas vRing (virtual Ring), em que os processos formam um anel virtual. O vRing utiliza o número mínimo de testes : N, enquanto a força-bruta necessita de $N^2 - N$ testes. Por outro lado, em termos da latência para um processo correto identificar um novo evento de falha ou recuperação em outro processo pode chegar a N rodadas de testes no vRing, mas é sempre de uma única rodada de testes na força-bruta. Na medida em que falsas suspeitas possam acontecer, o número de testes do vRing aumenta, ficando idêntido à força-bruta no pior caso.

O terceiro detector de falhas apresentado é o vCube, que foi originalmente proposto no contexto de diagnóstico em nível de sistema [Duarte and Nanya 1998, Duarte et al. 2014], sendo baseado nas premissas do diagnóstico distribuído. A principal daquelas premissas [Duarte Jr and Cestari 2000] é o fato de que um processo correto é capaz de realizar testes com precisão, isto é, sempre determina sem enganos o estado do processo testado. Mais tarde, o vCube foi especificado para sistemas assíncronos [Jeanneau et al. 2017], permitindo que processos corretos sejam enganosamente detectados como falhos. No vCube os processos formam uma topologia hierárquica virtual, mas a topologia subjacente deve ser *fully-connected*, isto é, representável por um grafo completo. Nesta topologia cada par de processos pode se comunicar diretamente sem depender de intermediários. As arestas virtuais de um vCube correspondem aos testes que os processos corretos executam entre si. Quando o número de processos é uma potência de 2 e não existem processos falhos nem falsas suspeitas, o vCube é um hipercubo. Entretanto, caso processos falhem, o vCube se reorganiza mantendo diversas propriedades logarítmicas, como o número de vizinhos de cada processo e a distância máxima entre dois processos.

O vCube organiza os processos em *clusters* $s = 1, .., \log_2 n$ progressivamente maiores, com 2^{s-1} processos. A Figura 1 ilustra os *clusters* de um vCube de 8 processos. A função $c_{i,s}$ (Equação 1) retorna a lista ordenada de processos de cada *cluster*, onde $\oplus$ é o operador *bitwise* exclusivo (*XOR*).

$$c_{i,s} = \{i \oplus 2^{s-1},\ c_{i \oplus 2^{s-1},\, 1}, \cdots,\ c_{i \oplus 2^{s-1},\, s-1}\} \tag{1}$$

A Tabela 1 mostra a função $c_{i,s}$ para 8 processos. Para determinar as arestas da

topologia virtual, para cada nó i existe uma aresta (j, i), tal que j é o primeiro nó sem falhas em $c_{i,s}$, $s = 1... \log_2 n$. Depois que um processo detecta que qualquer outro processo falhou/recuperou, o conjunto de arestas (testes) é recalculado. Por exemplo, na Figura 1, o processo p_0 originalmente testa o processo p_4 no *cluster* 3, mas depois que p_4 falha, p_0 passa a testar o processo p_5, que é o próximo considerado correto na lista da $c_{0,3}$.

Tabela 1. Resultado da função $c_{i,s}$ para 8 processos.

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

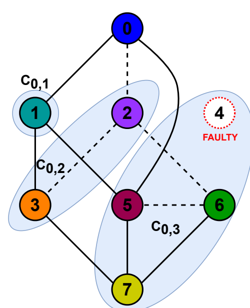


Figura 1. Clusters de um vCube com $2^3 = 8$ processos; p_4 está falho.

Um processo executando o vCube mantém um contador para a troca de estados de todos os processos monitorados, denominado *timestamp*. Utilizando o *timestamp* de um processo testado, é possível determinar cada novo evento que ocorre (suspeição de falha ou recuperação), sendo assim possível diferenciar eventos recentes de eventos mais antigos. Inicialmente, cada processo é considerado correto e o *timestamp* correspondente é zero. Quando há uma suspeita o *timestamp* é incrementado de uma unidade, assim um valor par de *timestamp* indica que o processo é percebido como correto, enquanto a suspeição é indicada por um valor ímpar. Na implementação proposta para sistemas assíncronos, o cenário de recuperação não é considerado. Portanto, os contadores se restringem aos valores zero (correto/sem-falha) e um (falho/suspeito).

O vCube pode ser visto como uma solução escalável para a detecção de falhas, pois apresenta desempenho em termos da latência de execução logarítmico, $log_2 N$ rodadas de testes. Enquanto isso, o número máximo de testes executados é igual a $N log_2 N$. O vCube já foi utilizado para a implementação de diversas abstrações distribuídas, como a exclusão mútua distribuída [Rodrigues et al. 2018a], a formação de quórums dinâmicos e autonômicos [Rodrigues et al. 2016], a difusão causal [Rodrigues et al. 2018b] e a difusão atômica [Ruchel et al. 2024].

3. Eleição de Líder: Trabalhos Relacionados

A eleição de processos instáveis para coordenar ações de um sistema distribuído pode degradar o desempenho. [Aguilera et al. 2000] propõem a percepção da instabilidade de processos por meio da quantidade encarnações (no original o termo usado é *epochs*).

O número de encarnações corresponde à quantidade de vezes que o processo falhou e recuperou. No artigo é construído um detector de falhas no modelo *crash-recovery* que utiliza informações de estabilidade para coordenar ações.

Na literatura é possível encontrar critérios diversos para a escolha do líder mais apropriado. [Biswas and Tripathi 2021] e outros consideram um sistema multi-hop. O critério para escolha do líder pondera capacidade de processamento e memória, além de critérios de conectividade, incluindo o grau e a excentricidade de cada processo. Com base nos múltiplos critérios é definida uma lista ordenada de líderes em potencial. Já [Fernández-Campusano et al. 2017] penalizam na eleição os processos a cada nova encarnação, fazendo que processos instáveis sejam preteridos na escolha do novo líder. Em [Biswas et al. 2021], utiliza-se um fator calculado baseado na taxa de falhas e de carga de cada processo, sendo líder o processo de menor taxa de falhas e menor carga. Em outra abordagem recente, [Wang and Gupta 2023] computa as encarnações para definir quais os nós saudáveis e destes escolher o líder, se for possível. Por fim, [Gómez-Calzado et al. 2013] é outra solução em que um algoritmo baseado em armazenamento estável utiliza a encarnação como critério de eleição, de modo que processos de menor encarnação tem maior prioridade na eleição de líder.

O algoritmo proposto em [Santoro 2006] se beneficia de uma rede de sobreposição em um hipercubo completo para eleição de líder com um número reduzido de passos em face às propriedades logarítmicas do hipercubo, mas não há suporte ao cenário *crash-recovery*.

Nossa proposta de eleição é semelhante a diversas destas mencionadas acima, utilizando o número de encarnações como principal critério de escolha do líder. Na Seção 5 o algoritmo é comparado com a estratégia tradicional descrita em [Cachin et al. 2011], na qual todos os processos monitoram todos os demais. Aquele algoritmo é baseado em heartbeats. Se dentro de um intervalo de monitoramento não é recebido um *heartbeat* de um processo, então ele é considerado suspeito e é retirado do conjunto de candidatos a líder. O modelo temporal também é o GST, e a cada vez que um processo recupera, o intervalo de monitoramento é aumentado de forma que, quando o sistema fica síncrono, o *timeout* é grande o suficiente para receber heartbeats de todos os processos efetivamente corretos.

4. O Algoritmo de Eleição de Líder Hierárquica

Nesta seção, o algoritmo hierárquico para eleição de líder é descrito e especificado. Este algoritmo apresenta uma versão do vCube no modelo *crash-recovery*, em que cada processo mantém em memória secundária não volátil informações que pode usar após recuperar de uma falha. O Algoritmo 1 apresenta o pseudocódigo da solução de eleição de líder com o vCube. As variáveis de estado mantidas pelos processos são:

- $Correct_i$: lista dos processos considerados corretos pelo processo i.
- $timestamp_i[]$: vetor como os *timestamps* de detecção de falhas do vCube. Valores pares indicam que o processo está em um estado considerado correto e valores ímpares indicam que o processo está suspeito de falha. É utilizado para identificar a validade da informação recebida;
- $leader_i$: identificador do processo considerado líder pelo processo i.

- $epoch_i[]$: armazena as informações de encarnação de cada processo no sistema. É atualizado em cada rodada de testes e utilizado na escolha o processo líder.

Inicialmente, todos os processos consideram o processo zero como líder (linha 8). Em seguida, cada processo executando o vCube realiza seus testes assinalados - linha 11 - e, caso o processo testado responda adequadamente ao teste (linha 12), as informações recebidas são atualizadas localmente, incluindo informações sobre as encarnações processos. Após completar a rodada de testes, o processo executa o procedimento `CheckLeader()` para atualizar a informação sobre o processo líder no sistema. O líder é o processo correto de menor identificador entre aqueles com o menor número de encarnações.

Em caso de recuperação do processo após ter sofrido falha, o evento `recover` é executado (linha 28). O processo recupera a sua última informação de encarnação da memória não-volátil, incrementa o contador e salva o valor atualizado. Em seguida, reinicia o processo de monitoramento no vCube.

Como exemplo, considere a execução de um cenário sem falhas com 8 processos, conforme ilustrado na Figura 2. Inicialmente, cada processo i testa o seu vizinho no cluster $c_{i,1}$, ou seja, p_0 testa p_1, p_2 testa p_3, p_4 testa p_5 e p_6 testa p_7, e vice-versa. Na próxima rodada, cada processo testa o seu vizinho no cluster $c_{i,2}$. Assim, p_0 testa p_2 e obtém informação sobre p_2 e p_3 (lembre-se que p_3 foi testado por p_2 na rodada anterior). O mesmo ocorre para os demais processos. Na terceira e última rodada, cada processo testa o vizinho direto no cluster $c_{i,3}$. No exemplo, p_0 testa p_4 e obtém informações sobre p_4, p_5, p_6 e p_7, que já foram testados por p_4 nas rodadas anteriores. Com isso, após $\log_2 N$ rodadas, todos os processos possuem informação sobre os demais processos no sistema e podem decidir sobre a liderança. Neste caso, o processo p_0 será eleito o líder.

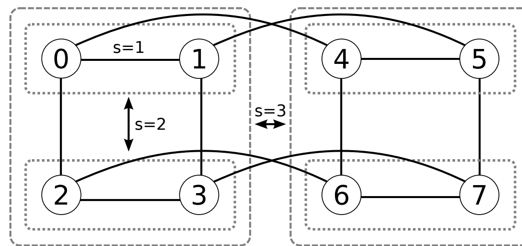


Figura 2. Clusters de um vCube com $2^3 = 8$ processos.

Em um segundo exemplo, considere que o processo p_0 falhou. Na primeira rodada, p_1 testa p_0 como falho. Na segunda rodada, p_2 executa o teste em p_0 e suspeita dele. Na mesma rodada, p_3 testa p_1 e obtém informação sobre a falha de p_0 a partir dele. Na última rodada, p_4 executa o teste em p_0 e identifica a sua falha, assim como p_5 faz a descoberta a partir de p_1, e assim todos os demais a partir dos respectivos vizinhos no *cluster* $c_{i,3}$. Portanto, todos os processos receberão a informação sobre a falha de p_0 e poderão eleger p_1 como líder.

Em um último cenário de exemplo, considere que p_0 se recuperou da falha. Neste caso, o seu contador de encarnação é atualizado para 1 (um). Assim, embora p_0 venha a ser considerado correto por todos os demais processos, p_1 ainda é eleito líder, pois o seu contador de encarnação continua em zero.

Algoritmo 1 Eleição de Líder Hierárquica

/* Executado pelo processo i do sistema $\Pi = \{0, .., n-1\}$ */

1: **procedure** INIT( )
2: $timestamp_i[p] \leftarrow 0, \forall p \in \Pi$ ▷ timestamps de detecção dos processos falhos
3: $epoch_i[p] \leftarrow 0, \forall p \in \Pi$ ▷ contador de encarnações de cada processo
4: STORE($epoch_i[i]$)
5: STARTMONITOR( )

6: **procedure** STARTMONITOR( )
7: $Correct_i \leftarrow P$
8: $leader_i \leftarrow 0$
9: **repeat**
10: **for** $s \leftarrow 1$ to $log_2 N$ **do**
11: **for all** $j \in c_{i,s} \mid i$ é o primeiro processo correto $\in c_{j,s}$ **do**
12: TEST(j)
13: **if** j está correto, i.e., respondeu corretamente ao teste **then**
14: Obtém informação de j: $timestamp_j[]$ e $epoch_j[]$
15: **if** $timestamp_i[j] \mod 2 = 1$ **then** ▷ j está como suspeito, mas retornou!
16: $Correct_i \leftarrow Correct_i \cup \{j\}$
17: $timestamp_i[j] + +$
18: **else**
19: **if** $timestamp_i[j] \mod 2 = 0$ **then** ▷ $j \in Correct_i$, mas agora será suspeito
20: $Correct_i \leftarrow Correct_i \setminus \{j\}$
21: $timestamp_i[j] + +$
22: CHECKLEADER( )
23: Aguardar até o próximo intervalo de testes
24: **forever**

25: ▷ Entre os processos corretos com menor encarnação, o líder é de menor id
26: **procedure** CHECKLEADER( )
27: $leader \leftarrow p \in Correct_i \mid p = min(epoch_i[p], p)$

28: **procedure** RECOVER( )
29: $timestamp_i[p] \leftarrow 0, \forall p \in \Pi$
30: $epoch_i[p] \leftarrow 0, \forall p \in \{\Pi \setminus i\}$
31: RETRIEVE($epoch_i[i]$)
32: $epoch_i[i] + +$
33: STORE($epoch_i[i]$)
34: STARTMONITOR( )

A correção do algoritmo pode ser elaborada em termos das propriedades clássicas da eleição de líder: precisão eventual e acordo eventual. Informalmente, a precisão eventual determina que todo processo correto eleja como líder um processo correto. O acordo eventual determina que não há dois processos corretos que elejam como líderes dois processos distintos. A precisão eventual é garantida pelo fato de que o vCube sempre atende à propriedade de completude forte, isto é: todo processo falho é eventualmente suspeito. Desta forma, todo processo correto sempre elege como líder um processo correto. Observar que se há processos que ficam falhando e recuperando (mesmo após o GST) eles não serão eleitos. Se houver pelo menos um processo que não falha após o GST, seu número de encarnação ficará menor que os dos processos instáveis que falham e recuperam, garantindo a eleição de um processo correto. O acordo eventual só pode ser garantido após o GST, isto é, antes do sistema apresentar as propriedades do modelo síncrono, podem ser eleitos múltiplos líderes. Entretanto, após o GST, o conjunto de processos corretos converge para o mesmo em todo o vCube. Como o critério de seleção de líder é determinístico, todos os processos elegem o mesmo líder.

5. Implementação e Resultados de Simulação

Esta seção apresenta a avaliação empírica executada através de simulação. Inicialmente são descritos o ambiente, os parâmetros e as métricas utilizadas. Em seguida são apresentados os resultados obtidos. O vCube foi comparado com uma solução clássica todos-para-todos (ALL) [Cachin et al. 2011], na qual cada processo correto testa todos os demais processos em todas as rodadas.

5.1. Ambiente de Simulação

As simulações foram realizadas utilizando o Neko [Urban et al. 2001], que é um *framework*[1] Java desenvolvido com o objetivo de permitir a simulação de algoritmos distribuídos. Sua arquitetura é dividida em dois níveis principais: aplicação e rede. Uma aplicação é construída na forma de microprotocolos. Os microprotocolos são registrados nos processos (*containers*), que são instâncias da classe NEKOPROCESS. No nível da aplicação, os processos comunicam-se utilizando troca de mensagens. As mensagens são enviadas e recebidas através dos métodos SEND e DELIVER, respectivamente. O segundo componente da arquitetura do Neko é a rede, que pode ser simulada ou real. Nos testes realizados, foi utilizada a rede BASICNETWORK, que utiliza um parâmetro `lambda` para gerar atrasos fixos de transmissão.

A simulação de falhas no Neko é realizada através da adaptação sugerida em [Rodrigues 2006]. Um mecanismo de colapso inicia e finaliza intervalos de falha, de acordo com o arquivo de configuração, que é o mesmo utilizado para as demais configurações do Neko. A aplicação envia mensagens para a classe de suporte de simulação a falhas, que verifica se o processo está em colapso. Se estiver, a mensagem é descartada. O mesmo ocorre para mensagens recebidas da rede. A aplicação pode fazer uma consulta ao estado do processo e, em caso de falha, parar sua execução através de uma *flag* `crashed`.

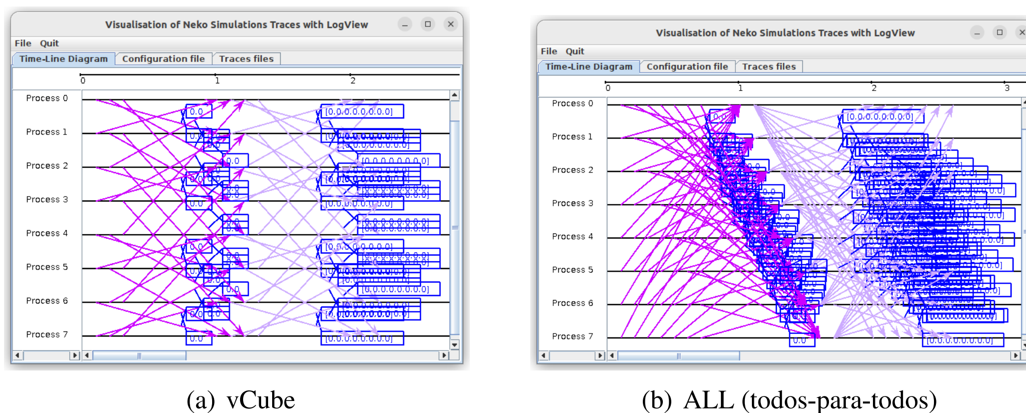
[1] Código-fonte disponível em: `https://github.com/arluiz/neko`

5.2. Métricas de Desempenho

O desempenho de algoritmos distribuídos é comumente medido com base em duas métricas: complexidade de tempo e complexidade de mensagem [Urban et al. 2000]. A complexidade de tempo reflete a latência do algoritmo, isto é, o tempo de execução. A complexidade de mensagem consiste em contar o número total de mensagens geradas pelo algoritmo.

Neste trabalho, a latência foi avaliada em dois aspectos tanto considerando o sistema sem falhas e com falhas. Na situação em que não há falhas, a latência é o intervalo de tempo para propagar as informações de monitoramento e de encarnação para todos os processos corretos. Na situação com falhas *crash* a latência é o tempo necessário para que todos os processos corretos detectem a ocorrência de uma falha e o líder seja eleito. No segundo caso, foi avaliada ainda a latência dos estados "inconsistentes", isto é, após a falha do líder, é o tempo para que o novo líder seja eleito por todos os processos corretos. O mesmo vale para os casos em que o líder por direito entra no sistema, mas ainda não foi eleito pelos demais. Por se tratar de casos semelhantes, este último cenário não foi avaliado explicitamente na simulação.

Para cada cenário, foram utilizados sistemas com $n = 2^d$ processos, para $d = 3, 4, .., 9$, i.e., $N = 8, 16, .., 512$. Os parâmetros de falhas estão descritos nos cenários a seguir, quando aplicável. O modelo de rede do Neko utilizado foi o BASICNETWORK. Para cada mensagem enviada, é considerado um tempo de envio de 0.1 intervalos de tempo e um tempo de transmissão da mensagem pela rede de 0.9. Assim, o intervalo de tempo total entre o envio e o recebimento de uma mensagem é 1.0, mas para cada mensagem enviada em sequência, o tempo de envio é deslocado em 0.1 (como ilustrado na Figura 3).



(a) vCube

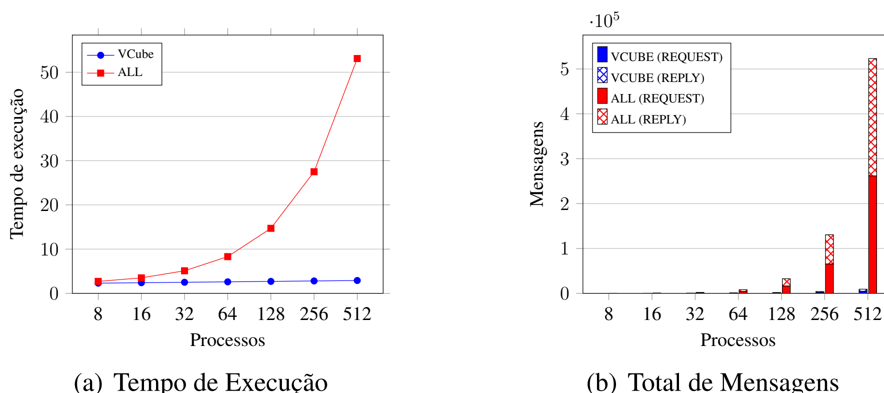
(b) ALL (todos-para-todos)

Figura 3. Troca de mensagens de testes no vCube e ALL com 8 processos.

5.3. Cenários Sem Falhas

Neste primeiro cenário de testes, não há falhas em nenhum dos processos. Visando estabelecer um parâmetro de comparação e considerando que não há eventos de falha ou recuperação, apenas uma rodada de testes foi realizada por cada algoritmo. Na Figura 4, é possível observar o tempo de execução e o número de mensagens das abordagens vCube e ALL. O número de mensagens também é detalhado na Tabela 2. O tempo de execução

varia em ambos os algoritmos, com um aumento maior para ALL conforme o número de processos aumenta. O número de mensagens de ALL é visivelmente maior (N^2), ao passo que vCube envia $N \log_2 N$ mensagens por rodada. Cada teste realizado por um processo em outro utiliza uma estratégia de requisição (REQUEST) e resposta (REPLY) – modelo *pull*.



(a) Tempo de Execução

(b) Total de Mensagens

Figura 4. Execução sem falhas na rede BASICNETWORK.

Tabela 2. Número de mensagens do vCube e ALL em cenários sem falhas.

Processos	ALL	vCube	Diferença
8	112	48	57,14%
16	480	128	73,33%
32	1.984	320	83,87%
64	8.064	768	90,48%
128	32.512	1.792	94,59%
256	130.560	4.096	96,86%
512	523.264	9.216	98,28%

5.4. Cenários com Falhas *Crash*

Para simular uma falha *crash*, utilizou-se o mecanismo do Neko no qual foi possível definir um sinal de falha do processo 0 no tempo 0.0 da simulação. Neste caso, cada execução considerou $\log_2(n)$ rodadas, que é latência máxima de diagnóstico do vCube no pior caso. O intervalo entre rodadas de testes para os dois algoritmos foi configurado em 30,0.

Por se tratar de uma única falha, o tempo de execução é muito próximo do cenário sem falhas, visto que cada execução inclui mais de uma rodada de testes e a latência de detecção é diluída no tempo total. O mesmo acontece com o número de mensagens, ainda que seja ligeiramente menor a medida que os processos detectam a falha e deixam de testar o processo falho. A Figura 5 ilustra o comportamento do vCube e ALL frente à detecção da falha do processo p_0 em um sistema com $N = 8$ processos. Note que no vCube, são necessárias três rodadas para que todos os processos identifiquem a falha. Isto se deve à distância entre o processo falho e o processo p_7 que equivale ao diâmetro do vCube ($\log_2 8 = 3$). No ALL, como o teste é realizado em todos os processos diretamente, a latência é menor, variando de uma rodada de testes (se a falha iniciar antes da rodada – caso do exemplo), a duas rodadas de teste (se a falha iniciar no meio da rodada).

```
#vCube
2,300 p3 messages Leader remains: 0
2,300 p5 messages Leader remains: 0
2,300 p6 messages Leader remains: 0
2,300 p7 messages Leader remains: 0
4,400 p1 messages suspect 0
4,400 p1 messages New Leader: 1
6,500 p2 messages suspect 0
6,500 p2 messages New Leader: 1
6,600 p4 messages suspect 0
6,600 p4 messages New Leader: 1
..
34,400 p3 messages suspect 0
34,400 p5 messages suspect 0
34,500 p6 messages suspect 0
34,600 p3 messages New Leader: 1
34,600 p5 messages New Leader: 1
34,600 p6 messages New Leader: 1
34,600 p7 messages Leader remains:
    0
..
66,700 p7 messages suspect 0
66,900 p7 messages New Leader: 1
```

```
#ALL
4,800 p1 messages suspect 0
4,800 p1 messages New Leader: 1
4,800 p2 messages suspect 0
4,800 p2 messages New Leader: 1
4,800 p3 messages suspect 0
4,800 p3 messages New Leader: 1
4,800 p4 messages suspect 0
4,800 p4 messages New Leader: 1
4,800 p5 messages suspect 0
4,800 p5 messages New Leader: 1
4,800 p6 messages suspect 0
4,800 p6 messages New Leader: 1
4,800 p7 messages suspect 0
4,800 p7 messages New Leader: 1
```

Figura 5. Log de execução das três primeiras rodadas para p_0 falho ($N = 8$).

A Figura 6 demonstra a latência para detecção da falha *crash* iniciada no tempo 0.0, isto é, o intervalo de tempo entre a falha de um processo e a identificação da falha em todos os processos corretos. Como o processo 0 é o primeiro a ser testado por todos os processos no ALL, o tempo é proporcional ao número de processos e está atrelado ao *timeout* do teste. Este valor poderia ser proporcionalmente maior se o processo falho fosse o último a ser testado devido ao atraso de transmissão das mensagens sequenciais. Ainda assim, nos cenários avaliados, a detecção aconteceria em uma mesma rodada de testes. A latência de diagnóstico no vCube está vinculada diretamente ao número de rodadas de testes. Na primeira rodada, apenas os vizinhos ligados virtualmente ao processo falho identificam o evento. Na segunda rodada, os vizinhos com dois saltos de distância identificam a falha e, assim sucessivamente. Note que a latência de diagnóstico é ligeiramente menor (vCube-Diagnóstico) se comparada ao tempo necessário para completar todos os testes da rodada (vCube-Total).

Assim, embora o número de mensagens seja muito maior na estratégia todos-para-todos (semelhante ao cenário sem falhas), a latência de diagnóstico é sempre ligeiramente maior no vCube. O aumento é de 1 rodada de testes para $log_2 N$ rodadas (no pior caso). Este resultado é comparável para a recuperação de um processo falho, já que a propagação do evento segue a mesma estratégia. Por esta razão, a recuperação não foi simulada.

6. Conclusão

O presente artigo apresenta um novo algoritmo hierárquico para a eleição de líder em sistemas distribuídos, utilizando a topologia virtual vCube. O algoritmo proposto é projetado para lidar com falhas no modelo *crash-recovery*, considerando que os processos têm acesso a memória não volátil. Ele seleciona o líder com base na estabilidade dos processos, além do identificador. Resultados de simulação demonstram que a solução é eficiente e escalável, em comparação com a abordagem tradicional em que todos os processos monitoram todos os demais. Apesar de que o número de rodadas de testes para completar a

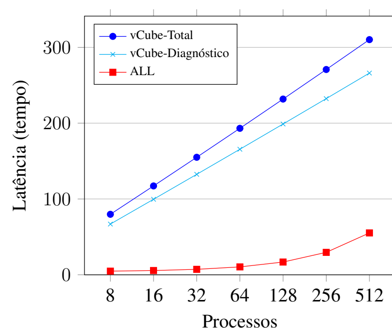


Figura 6. Latência de diagnóstico com uma falha (p_0) no tempo 0.0

eleição aumenta logaritmicamente, o número de testes (mensagens) necessários é significativamente menor. Trabalhos futuros incluem permitir a aplicação da eleição hierárquica para redes dinâmicas [Kumar et al. 2015], o que implica em portar o algoritmo para um modelo em que a composição do sistema varia com o tempo.

Agradecimentos

O presente trabalho foi parcialmente apoiado pelo projeto CNPq 308959/2020-5.

Referências

Aguilera, M. K., Chen, W., and Toueg, S. (2000). Failure detection and consensus in the crash-recovery model. *Distributed computing*, 13(2):99–125.

Biswas, A., Maurya, A. K., Tripathi, A. K., and Aknine, S. (2021). Frlle: a failure rate and load-based leader election algorithm for a bidirectional ring in distributed systems. *The Journal of Supercomputing*, 77:751–779.

Biswas, A. and Tripathi, A. K. (2021). Preselection based leader election in distributed systems. In *International Symposium on Intelligent and Distributed Computing*, pages 261–271. Springer.

Bona, L. C., Duarte Jr, E. P., Mello, S. L., and Fonseca, K. V. (2006). Hyperbone: Uma rede overlay baseada em hipercubo virtual sobre a internet. *XXIV SBRC*.

Cachin, C., Guerraoui, R., and Rodrigues, L. (2011). *Introduction to reliable and secure distributed programming*. Springer Science & Business Media.

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

Duarte, E. P., Albini, L. C., Brawerman, A., and Guedes, A. L. (2009). A hierarquical distributed fault diagnosis algorithm based on clusters with detours. In *The 6th IEEE Latin American Network Operations and Management Symposium*, pages 1–6. IEEE.

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.

Duarte, E. P., Garrett, T., Bona, L. C., Carmo, R., and Züge, A. P. (2010). Finding stable cliques of planetlab nodes. In *DSN 2010*, pages 317–322. IEEE.

Duarte, E. P. and Nanya, T. (1998). A hierarchical adaptive distributed system-level diagnosis algorithm. *IEEE Transactions on computers*, 47(1):34–45.

Duarte Jr, E. P. and Cestari, J. M. A. (2000). O agente chinês para diagnóstico de redes de topologia arbitrária. In *Anais do II WTF*, pages 88–93. SBC.

Duarte Jr, E. P., Rodrigues, L. A., Camargo, E. T., and Turchetti, R. C. (2023). The missing piece: a distributed system-level diagnosis model for the implementation of unreliable failure detectors. *Computing*, 105(12):2821–2845.

Fernández-Campusano, C., Larrea, M., Cortiñas, R., and Raynal, M. (2017). A distributed leader election algorithm in crash-recovery and omissive systems. *Information Processing Letters*, 118:100–104.

Gómez-Calzado, C., Larrea, M., Soraluze, I., Lafuente, A., and Cortiñas, R. (2013). An evaluation of efficient leader election algorithms for crash-recovery systems. In *2013 21st Euromicro*, pages 180–188.

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. Comput. Soc.*, 23(1):15:1–15:14.

Kumar, N., Pathan, A.-S. K., Duarte, E. P., and Shaikh, R. A. (2015). Critical applications in vehicular ad hoc/sensor networks. *Telecommunication Systems*, 58:275–277.

Rodrigues, L. A. (2006). Extensão do suporte para simulação de defeitos em algoritmos distribuídos utilizando o neko. Master's thesis, UFRGS.

Rodrigues, L. A., Arantes, L., and Duarte, E. P. (2016). An autonomic majority quorum system. In *2016 IEEE 30th AINA*, pages 524–531. IEEE.

Rodrigues, L. A., Duarte, E. P., and Arantes, L. (2018a). A distributed k-mutual exclusion algorithm based on autonomic spanning trees. *JPDC*, 115:41–55.

Rodrigues, L. A., Duarte, E. P., de Araujo, J. P., Arantes, L., and Sens, P. (2018b). Bundling messages to reduce the cost of tree-based broadcast algorithms. In *2018 8th LADC*, pages 115–124. IEEE.

Ruchel, L. V., de Camargo, E. T., Rodrigues, L. A., Turchetti, R. C., Arantes, L., and Duarte Jr, E. P. (2024). Scalable atomic broadcast: A leaderless hierarchical algorithm. *JPDC*, 184:104789.

Santoro, N. (2006). *Design and analysis of distributed algorithms*. John Wiley & Sons.

Stein, G., Rodrigues, L. A., Duarte Jr, E. P., and Arantes, L. (2023). Diamond-p-vcube: An eventually perfect hierarchical failure detector for asynchronous distributed systems. In *Proc. 12th LADC*, pages 40–49.

Urban, P., Defago, X., and Schiper, A. (2000). Contention-aware metrics for distributed algorithms: comparison of atomic broadcast algorithms. In *9th Int'l Conf.on Computer Communications and Networks (ICCCN)*, pages 582–589.

Urban, P., Defago, X., and Schiper, A. (2001). Neko: a single environment to simulate and prototype distributed algorithms. In *15th Int'l Conf. Info. Networking*, pages 503–511.

Wang, J. and Gupta, I. (2023). Churn-tolerant leader election protocols. In *43rd IEEE ICDCS*, pages 96–107. IEEE.