Utilizando o vCube para Detecção de Falhas em Sistemas Assíncronos

Gabriela Stein[1], Luiz Antonio Rodrigues[1] e Elias Procópio Duarte Jr.[2]

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

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

{Gabriela.Stein1, Luiz.Rodrigues}@unioeste.br, elias@inf.ufpr.br

Abstract. *This paper presents a solution for failure detection in asynchronous distributed systems. Each pair of system processes can execute tests on each other, but the testing graph is maintained based on the vCube virtual topology. Since there are no limits on the execution time of the processes and the communication delays, false suspicions may occur. To improve the accuracy of the detector, a process exits the system when it learns that it has been suspected by another. The proposed algorithm was compared with the traditional all-to-all solution. The results show that although the failure/false suspicion diagnosis latency is higher, the number of messages and the execution times decrease (comparatively) when the number of processes increases.*

Resumo. *Este trabalho apresenta uma solução para a detecção de falhas em sistemas distribuídos assíncronos. Qualquer par de processos do sistema pode executar testes mutuamente entre si, mas o grafo de testes é mantido com base na topologia virtual vCube. Dada a propriedade de não haver limites para o tempo de execução dos processos e do atraso de comunicação, falsas suspeitas podem ser sinalizadas. Para melhorar a acurácia do detector, quando um processo identifica que foi suspeito por outro, ele deixa o sistema. O algoritmo proposto foi comparado com uma solução típica todos-para-todos. Os resultados mostram que, embora a latência de diagnóstico de falhas/falsas suspeitas seja maior, o número de mensagens e o tempo de execução reduzem (comparativamente) na medida que o número de processos aumenta.*

1. Introdução

O monitoramento de sistemas computacionais e a identificação de falhas já eram tratados desde a década de 1960 no contexto do diagnóstico em nível de sistema [Masson et al. 1996]. Na década de 1990, os detectores de falhas não-confiáveis foram propostos [Chandra and Toueg 1996] como uma abstração para permitir a solução do problema do consenso em sistemas assíncronos, dada a impossibilidade de distinguir um processo lento de um processo falho quando limites temporais de comunicação e processamento são desconhecidos [Fischer et al. 1985].

A principal diferença entre as duas abordagens é que o diagnóstico em nível de sistema original assume que um processo correto é capaz de identificar corretamente o estado de qualquer processo que testa. Portanto, o diagnóstico assume implicitamente o modelo síncrono [Duarte Jr. et al. 2022]. Embora um modelo anterior de diagnóstico em nível de sistema tenha assumido testes que não são perfeitos [Camargo and Duarte 2018], apenas no trabalho recente de [Duarte Jr et al. 2022] os algoritmos de diagnóstico são especificados como detectores de falhas.

Tanto no diagnóstico quanto nos detectores de falhas, cada processo possui um módulo monitor que pode ser consultado para obter informações sobre o estado dos demais processos. Um processo pode estar em um de dois estados: correto ou falho, sendo assumido modelo de falhas por parada (*crash*). No caso dos detectores de falhas, a qualidade das informações fornecidas é medida a partir de duas propriedades: completude (*completeness*) e acurácia (*accuracy*) [Chandra and Toueg 1996]. Informalmente, a acurácia reflete a capacidade do detector de identificar processos que realmente falharam. A precisão, por outro lado, é a capacidade do detector de não suspeitar de processos corretos (falsas suspeitas).

No entanto, detectores de falha não-confiáveis podem fornecer informações imprecisas ao suspeitar incorretamente de processos corretos e/ou não suspeitar de nós com falha. Porém, ainda que flexibilizadas as propriedades de completude e acurácia, é possível propor detectores de falhas que auxiliam na resolução de problemas que exijam acordo entre os processos [Sergent et al. 2001]. Em [Chandra et al. 1996], por exemplo, é demonstrado que um detector com completude e acurácia fracas pode ser utilizado para resolver o consenso em sistemas com maioria correta de processos.

A topologia virtual vCube, utilizada neste trabalho, foi originalmente proposta no contexto de diagnóstico hierárquico adaptativo distribuído em nível do sistema (Hi-ADSD) [Duarte Jr. et al. 2022]. Embora todos os processos possam se comunicar diretamente (topologia *fully-connected*), os processos são organizados em uma rede de sobreposição virtual escalável. Vale ressaltar que existem outras abordagens do vCube baseadas em topologia arbitrária [Jr. and Mattos 2000] e em testes sincronizados [Brawerman and Jr. 2001]. No contexto de aplicações, o vCube já foi utilizado em soluções de violação de integridade [Ziwich et al. 2005], difusão de mensagens [Rodrigues et al. 2014], exclusão mútua [Rodrigues et al. 2018] e Publish/Subscribe [de Araujo et al. 2017], entre outras.

A solução proposta neste trabalho considera a execução da detecção de falhas do vCube em um ambiente assíncrono. Para lidar com as falsas suspeitas, que podem deixar a visão que os processos corretos têm dos outros processos em um estado temporariamente inconsistente, quando um processo detecta que foi suspeito, ele deixa o sistema, em definitivo. Resultados de simulação mostram especialmente a redução significativa no número de mensagens, o que também reduz o tempo de execução das rodadas de teste.

O texto segue da seguinte forma. A Seção 2 apresenta algumas propriedades dos detectores de falhas distribuídos. A Seção 3 descreve o modelo do sistema utilizado. O algoritmo proposto é apresentado na Seção 4. Os resultados são discutidos na Seção 6. A Seção 7 apresenta o trabalhos relacionados. Na Seção 5 é descrito o *framework* e o ambiente utilizado para simulação dos testes. A Seção 8 conclui o trabalho.

2. Detectores de Falhas Distribuídos

Um detector de falha pode ser visto como um conjunto de n módulos de detecção de falhas, cada um ligado a um processo diferente no sistema. Esses módulos cooperam para satisfazer as propriedades exigidas do detector de falhas [Larrea et al. 2000].

Considere as seguintes propriedades de completude definidas em [Chandra and Toueg 1996]:

- **Completude forte**: inevitavelmente, todo processo falho é permanentemente suspeito por todo processo correto;
- **Completude fraca**: inevitavelmente, todo processo falho é permanentemente suspeito por algum processo correto.

Se analisada separadamente, a completude forte é trivial de se obter forçando cada processo a suspeitar permanentemente de todos os outros processos no sistema, o que não é desejável. No entanto, considere as seguintes variações da acurácia, também definidas por [Chandra and Toueg 1996]:

- **Acurácia forte garantida**: inevitavelmente, nenhum processo correto é suspeito por nenhum outro processo correto;
- **Acurácia fraca garantida**: inevitavelmente, algum processo correto nunca é suspeito por nenhum processo correto.

Combinando estas propriedades em pares, podem ser definidas quatro classes de detectores de falha, apresentadas na Tabela 1.

Tabela 1. Classes de detectores com completude e acurácia fracas (*eventual*).

	Acurácia forte garantida	**Acurácia fraca garantida**
Completude forte	Inevitavelmente perfeito $\Diamond P$	Inevitavelmente forte $\Diamond S$
Completude fraca	Inevitavelmente quase-perfeito $\Diamond Q$	Inevitavelmente fraco $\Diamond W$

Os detectores $\Diamond P$ e $\Diamond S$ garantem que todos os processos falhos serão detectados por todos os processos corretos (completude forte) e que, em um tempo finito, as falsas suspeitas deixarão de acontecer para todos ou algum processo correto, respectivamente.

3. Modelo do Sistema

Considera-se um sistema distribuído como um conjunto finito P com $n > 1$ processos $\{p_0, .., p_{n-1}\}$ que se comunicam por troca de mensagens. As operações de envio e recebimento são atômicas e os enlaces são confiáveis.

A rede é representada por um grafo completo, ou seja, cada par de processos pode se comunicar diretamente sem depender de intermediários. No entanto, os processos são organizados em uma topologia virtual, chamada vCube [Duarte et al. 2014]. Quando não existem processos falhos, o vCube é um hipercubo completo. Com isso, a topologia oferece propriedades logarítmicas importantes, como o número de vizinhos de cada processo e a distância máxima entre dois processos.

O sistema admite falhas crash permanentes. Os processos falhos não enviam mais mensagens. Um processo que nunca falha e responde corretamente às mensagens do detector é considerado *correto* ou *sem-falha*. Caso contrário, ele é dito *falho* ou *suspeito*.

O modelo de sistema é assíncrono, isto é, não há limites nos atrasos de transmissão de mensagens e nas velocidades de processamento relativas dos processos [Chandra and Toueg 1996].

4. O vCube para Sistemas Assíncronos

A topologia virtual criada pelo vCube é hierárquica e corresponde a um hipercubo quando todos os processos estão corretos. Os processos de um vCube com dimensão $d > 0$ têm identificadores que consistem em d bits. Em um cenário sem falhas, dois processos estão virtualmente conectados se seus endereços binários diferem por um único bit. No exemplo da Figura 1, o processo 1 (001) se conecta com os processos 0 (000), 3 (011) e 5 (101).

As arestas virtuais de um vCube correspondem aos testes que os processos corretos executam entre si. O vCube permite que os processos obtenham informações de diagnóstico de cada processo testado corretamente. As informações obtidas são marcadas com contadores incrementais, que funcionam como *timestamps*, para permitir que os processos diferenciem eventos recentes de eventos mais antigos. Inicialmente, cada nó é considerado correto e o contador correspondente é zero. Originalmente, depois que um evento é detectado, ou seja, um nó correto tornou-se falho ou vice-versa, o contador correspondente é incrementado em um. 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 organiza os processos em clusters $s = 1, .., \log_2 n$ progressivamente maiores, com 2^{s-1} 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 2 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, 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, o processo p_0 passa a testar o processo p_5, que é o próximo considerado correto na lista da $c_{0,3}$.

Tabela 2. 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

O Algoritmo 1 apresenta o pseudocódigo do vCube adaptado de [Duarte et al. 2014] para uso em um sistema assíncrono. O processo i mantém registros de *timestapms* para o estado de todos os outros processos no vetor $STATE_i[]$.

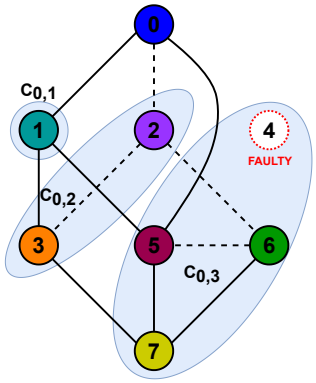


Figura 1. Clusters de um vCube com $2^3 = 8$ processos; p_4 está falho.

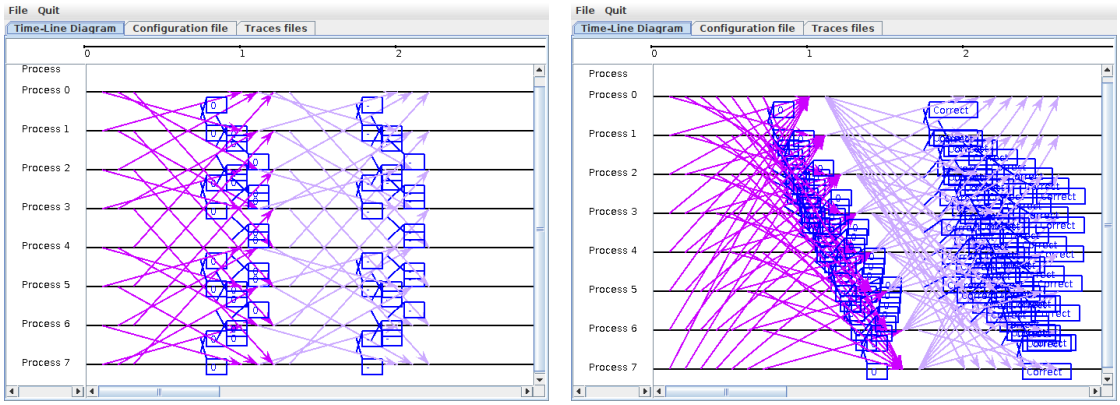
O algoritmo prevê que no caso de uma falsa suspeita, ou seja, se um processo i foi suspeitado erroneamente por um processo j, o processo i, ao receber esta informação, para de executar e deixa o sistema (linha 8). O resultado é um detector de falha $\Diamond P$ que é inevitavelmente perfeito [Chandra and Toueg 1996] se os processos corretos permanecerem em um único componente conectado, ou seja, se não houver partições no grafo de teste.

Algoritmo 1 Detector de falhas vCube assíncrono no processo i

1: $STATE_i[j] \leftarrow 0, \forall j = 0, .., n-1$
2: **repeat**
3: **for** $s \leftarrow 1$ **to** $\log_2(n)$ **do**
4: **for all** $j \in c_{i,s} \mid i$ é o primeiro processo correto $\in c_{j,s}$ **do**
5: TEST(j)
6: **if** $j \in correct$ **then**
7: **if** $STATE_j[i] \mod 2 = 1$ **then** $\triangleright$ i foi suspeito por j (falsa suspeita)
8: **Terminar a execução e deixar o sistema**
9: **else**
10: Atualizar as informações de $STATE_j[]$ a partir de $STATE_j[]$
11: Aguardar até o próximo intervalo de testes

A Figura 2 ilustra uma rodada de testes com o vCube na Figura 2(a) e uma implementação clássica de detecção todos-para-todos (Figura 2(b)). No caso do vCube, cada processo é testado por, no máximo, $\log_2 n$ vizinhos, totalizando $n * log_2 n$ testes no pior caso. Na estratégia todos-para-todos, cada processo testa todos os $n-1$ vizinhos diretamente, totalizando $n(n-1)$ testes, ou seja, quadrático. A primeira sequência de mensagens representa as solicitações ARE_YOU_ALIVE e a segunda, as respostas I_AM_ALIVE. Neste caso, o número de mensagens é o dobro do número de testes.

A latência para o vCube detectar a falha de um processo é uma função logarítmica do número total de processos do sistema. Mais tarde, foi demonstrado que o algoritmo original leva a um número quadrático de testes em alguns casos específicos. Como resultado, foi proposto o algoritmo vCube [Duarte et al. 2014], o que garante que o número de testes seja uma função logarítmica do número total de processos, variando $\log_2 N$ rodadas



(a) vCube (b) All-to-All

Figura 2. Troca de mensagens de testes no vCube e no Todos-para-Todos com 8 processos (geradas com a ferramenta LogView do Neko [Urban et al. 2001]).

(melhor caso) a $\log_2^2 N$ (pior caso).

5. Ambiente de Simulação

O Neko [Urban et al. 2001] é um *framework*[1] Java desenvolvido com o objetivo de permitir a simulação de algoritmos distribuídos e avaliar o funcionamento dos mesmos. 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. Os microprotocolos podem se comunicar de duas formas: através da rede, quando estão registrados em processos diferentes, ou através da invocação do método apropriado por meio de referência direta, quando pertencem ao mesmo processo. Quando a mensagem é enviada pela rede, assim que é recebida pelo processo destino é entregue diretamente para o microprotocolo indicado no cabeçalho da mensagem. Em uma simulação todos os processos estão localizados em uma única máquina. Em uma execução distribuída, cada processo pode estar localizado em uma máquina diferente.

O segundo componente da arquitetura do Neko é a rede, que pode ser simulada ou real. As redes simuladas incluem duas implementações principais:

- BASICNETWORK: utiliza um parâmetro `lambda` para gerar atrasos fixos de transmissão;
- RANDOMNETWORK: gera atrasos de transmissão aleatórios (RANDOM do Java) com base em um parâmetro `lambda`, que varia entre 0 e 1. O parâmetro `seed` pode ser utilizado para reproduzir um experimento específico.

Uma rede real utiliza a comunicação por *sockets* TCP e deve ser instanciada a partir da execução de módulos em cada hospedeiro. Com isso, o tempo de processamento e comunicação serão obtidos a partir do ambiente real.

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

5.1. Suporte a Simulação de Falhas e Falsas Suspeitas

A simulação de falhas no Neko é realizada através da adaptação sugerida em [Rodrigues 2006] e ilustrada na Figura 3. 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`.

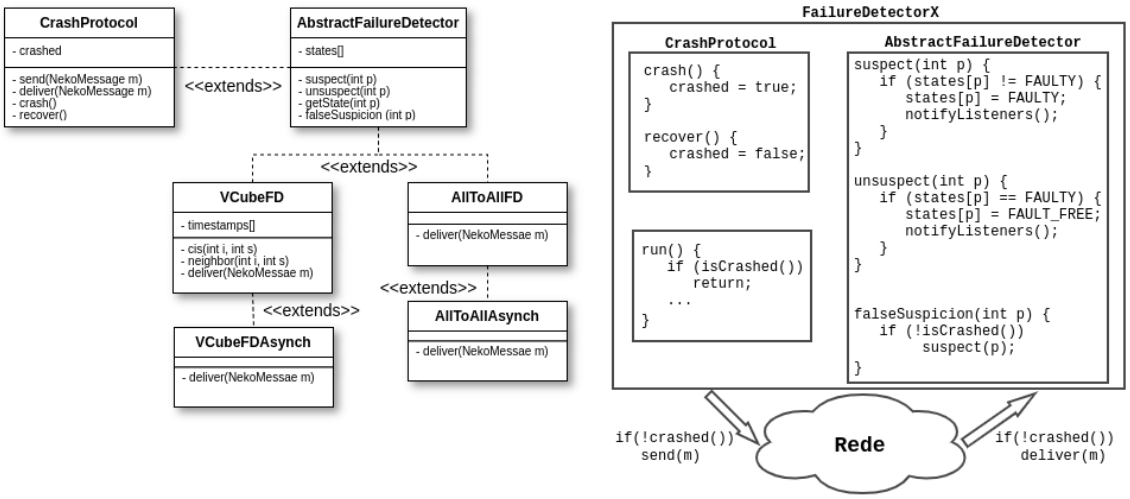


Figura 3. Modo de simulação de falhas de crash e falsas suspeitas no Neko.

Neste trabalho, foram incluídos os mecanismos para a simulação de falsas suspeitas. A classe `AbstractFailureDetector` passou a disponibilizar um método `falseSuspicion(int p)`, que recebe como parâmetro o identificador do processo p que será suspeito pelo processo i. O método verifica se o processo está em um estado de execução válido, isto é, não está falho no contexo do simulador, e invoca o método `suspect(p)`, que já fazia parte do Neko adaptado por [Rodrigues 2006]. A invocação deste método é feita por meio de um parâmetro `process.i.false_suspect.process.p = <tempo>`, que deve ser incluído no arquivo de configuração geral da simulação. O código-fonte e um exemplo de configuração estão disponíveis em `https://github.com/arluiz/vcubefd-asynch`.

6. Resultados

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 mede 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: 1) falhas *crash*: latência de diagnóstico da falha *crash*, isto é, após a falha, é o tempo necessário para todos os

demais processos serem notificados; 2) falsas suspeitas: a) tempo para o processo identificar que foi suspeito incorretamente; b) latência de detecção da falsa suspeita: é o tempo para todos os processos identificarem um processo como falho a partir da falsa suspeita reportada pelo primeiro processo.

Os algoritmos foram avaliados em três cenários: a) sem falhas ou falsas suspeitas; b) com falhas *crash*; e c) sem falhas, mas com falsas suspeitas. Para cada cenário, foram utilizados sistemas com $n = 2^d$ processos, para $d = 2, 3, .., 8$, i.e., $n = 4, 8, 16, .., 256$. Os parâmetros de falhas estão descritos nos cenários a seguir, quando aplicável.

Os dois modelos de rede do Neko utilizados foram o BASICNETWORK e o RANDOMNETWORK, descritos na Seção 5. Para cada mensagem enviada, é considerado um tempo de envio de 0.1 intervalos de tempo, um tempo de transmissão da mensagem pela rede e um tempo de recebimento de 0.1. Nas simulações com a rede BASICNETWORK, o intervalo de tempo 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 2). Como o recebimento é controlado pelo simulador, o tempo de transmissão na rede foi configurado pelo parâmetro `BasicNetwork.lambda=0.9`. Na rede RANDOMNETWORK, o tempo de transmissão varia aleatoriamente de acordo com um parâmetro `RandomNetwork.lambda`.

O vCube foi comparado com uma solução clássica todos-para-todos (ALL), na qual cada processo correto envia um teste individual para todos os demais processos considerados corretos. Para uma comparação justa, em caso de falsa suspeita, o processo suspeito também deixa o sistema.

Todos os testes foram realizados em $\log_2^2(n)$ rodadas, que é latência máxima de diagnóstico do vCube no pior caso, de modo a garantir que todos os processos serão testados por todos os outros. O intervalo de testes para os dois algoritmos foi configurado em 30.0. A cada intervalo de testes, o vCube testa apenas os vizinhos.

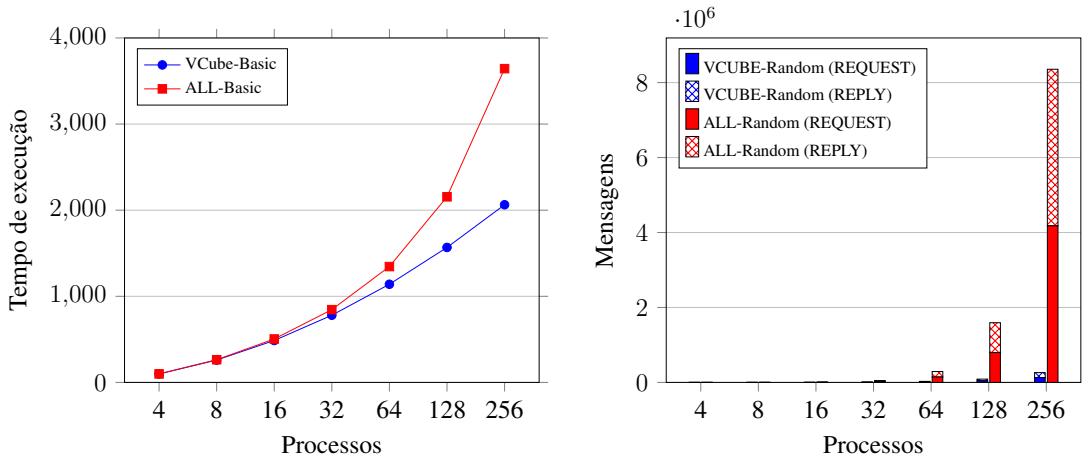
6.1. Sem falhas ou falsas suspeitas

Neste primeiro cenário de testes, utilizou-se dos modelos de rede BASICNETWORK e RANDOMNETWORK para simular cenários nos quais não houvessem falhas em nenhum dos processos.

Na Figura 4, é possível observar o tempo de execução de todas as rodadas e número de mensagens do algoritmo vCube e o todos-para todos (ALL), após ser executado em um modelo de rede BASICNETWORK. 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 é visivelmente maior, sendo $n \log_2 n$ testes por rodada para o vCube e n^2 para ALL. Cada teste utiliza uma estratégia de requisição (REQUEST) e resposta (REPLY).

O segundo cenário testado foi um modelo de rede RANDOMNETWORK com parâmetro de atraso de transmissão `lambda=0.1`, que gera atrasos aleatórios, porém sem grandes variações, evitando assim as falsas suspeitas. O resultado é exatamente o mesmo do cenário anterior, exceto pelas pequenas variações nos atrasos de transmissão. Por uma questão de espaço, os gráficos foram omitidos.

A Tabela 3 exibe-se os valores referentes ao número de mensagens dos dois



(a) Tempo de Execução (b) Total de Mensagens

Figura 4. Execução sem falhas na rede BASICNETWORK**.**

cenários sem falhas. Independente do tipo de rede, o total de mensagens é o mesmo.

Tabela 3. Número de mensagens do vCube e ALL em cenários sem falhas.

Processos	ALL	vCube	Diferença
4	96	64	33%
8	1.008	432	57%
16	7.680	2.048	73%
32	49.600	8.000	84%
64	290.304	27.648	90%
128	1.593.088	87.808	94%
256	8.355.840	262.144	97%

6.2. 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 (zero) da simulação. Os algoritmos vCube e ALL foram executados no modelo de rede BASICNETWORK. 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 demonstra a latência de diagnóstico de uma falha *crash* iniciada no tempo 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 constante 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, 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.

Portanto, a latência de diagnóstico é sempre maior no vCube se comparado com todos-para-todos. Este resultado é comparável para falsas suspeitas, já que a propagação do evento segue a mesma estratégia.

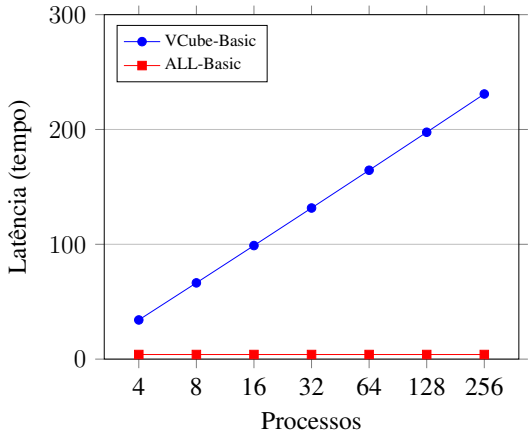


Figura 5. Latência de diagnóstico com uma falha no tempo 0 em uma rede BASICNETWORK**.**

6.3. Falsas suspeitas

O cenário com introdução de falsas suspeitas utilizando o novo recurso implementado no Neko possui comportamento análogo ao cenário de *crash*. A Figura 6 ilustra um exemplo de *log* gerado para um sistema com 4 processos. O processo 0 foi configurado para suspeitar do processo 1 no tempo 0. Na primeira linha, o processo 0 já sinaliza que suspeitou do processo 1. Portanto, passa a testar diretamente o processo 2 (linha 3). O processo 1 testa o processo 0 e recebe a informação que está suspeito por ele (linhas 4 e 10, respectivamente). Ao receber esta informação, o processo 1 deixa o sistema (linha 11) e, na linha 12, uma sinalização de *crash* é criada por ele para parar a sua execução.

```
1  0,000 p0 messages falsely suspect p1
2  0,000 p0 messages suspect 1
3  0,100 p0 messages e s p0 p2 ARE_YOU_ALIVE 0
4  0,100 p1 messages e s p1 p0 ARE_YOU_ALIVE 0
5  ..
6  1,000 p0 messages e r p1 p0 ARE_YOU_ALIVE 0
7  ..
8  1,100 p0 messages e s p0 p1 I_AM_ALIVE true
9  ..
10 2,000 p1 messages e r p0 p1 I_AM_ALIVE true
11 2,000 p1 messages p0 suspect me: die!
12 2,000 p1 messages crash started at crash-all2all-fd
```

Figura 6. Recorte do *log* gerado para uma falsa suspeita do processo 1 pelo processo 0 na rede BASICNETWORK**.**

Uma segunda forma de testar falsas suspeitas é utilizar a rede RANDOMNETWORK. Neste caso, foi utilizado o `lambda = 0.9`, que gera grandes variações

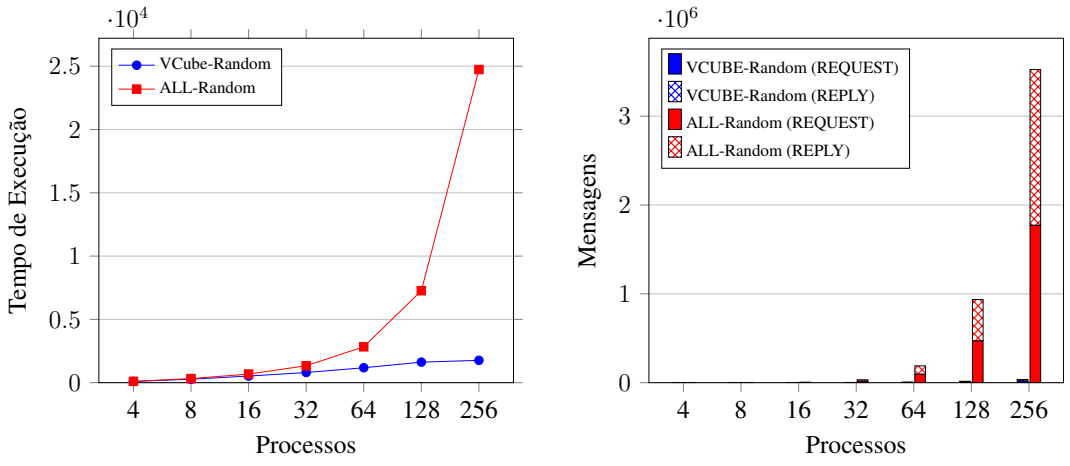
no tempo de transmissão da rede, ocasionando *timeouts* prematuros e, consequentemente, falsas suspeitas.

A Figura 7 apresenta o tempo total de execução e o número de mensagens para os dois algoritmos testados. O tempo total de execução das $\log_2^2 n$ rodadas em cada cenário é menor que nas execuções anteriores, especialmente para o vCube. Isso se dá pela quantidade de processos que foram suspeitos e deixaram o sistema, conforme pode ser observado na Tabela 4.

Tabela 4. Quantidade de processos suspeitos na rede RANDOMNETWORK **com** `lambda=0.9`**.**

Processos	ALL	vCube
4	1	1
8	3	4
16	4	11
32	6	29
64	13	62
128	31	126
256	94	255

Pela mesma razão, na Figura 7(b), é possível identificar que há uma redução significativa na quantidade de mensagens trocadas pelos processos. Isto se dá porque este cenário produz muitos processos suspeitos, que deixam a rede ao se identificarem como suspeitos. Com isso, menos testes são realizados tanto pelos que deixaram a rede, quanto pelos que detectaram a falha e deixaram de testar o processo suspeito. Embora seja o comportamento da rede, não uma comparação justa porque no vCube teve muito mais processos suspeitos que no ALL.



(a) Tempo de Execução (b) Total de Mensagens

Figura 7. Execução com mais de uma falsa suspeita na rede RANDOMNETWORK **com** `lambda=0.9`**.**

7. Trabalhos Relacionados

Em [Chandra and Toueg 1996], os autores apresentaram um algoritmo de detecção por *timeouts* que implementa um detector $\Diamond P$. O algoritmo utiliza mensagens de *heartbeat*

trocadas periodicamente entre todos os processos, o que exige um número quadrático de mensagens e um sistema parcialmente síncrono.

Visando reduzir o número de mensagens de detecção, [Larrea et al. 1999] propuseram uma solução baseada em anel para sistemas parcialmente síncronos. Cada processo monitora apenas o próximo processo correto no anel, i.e., o seu sucessor direto. Assim, o número de mensagens enviadas periodicamente é linear. É um detector da classe $\Diamond P$, isto é, processos falhos são permanentemente suspeitos, mas podem ocorrer falsas suspeitas durante um período de instabilidade.

Posteriormente, [Larrea et al. 2000] propuseram uma nova versão do algoritmo que implementa um $\Diamond S$. A organização em anel foi mantida, mas ao invés de cada processo testar o seu sucessor, o sistema busca estabelecer um processo correto comum a todos os demais. Neste caso, apenas o processo correto comum a todos envia mensagens "I_AM_ALIVE" periodicamente. Se o processo confiável deixar de enviar mensagens, o próximo processo no anel passa a ser o confiável. O monitoramento é realizado por *time-outs* e o intervalo de temporização varia para cada processo monitorado e é incrementado sempre que uma falsa suspeita é detectada. A versão foi comparada com a solução de [Chandra and Toueg 1996] e [Larrea et al. 1999] e mostrou-se mais eficiente em número e tamanho das mensagens.

SWIM [Das et al. 2002] é uma solução de detecção de falhas que busca aumentar a escalabilidade dos protocolos de detecção. Ao contrário dos protocolos baseados em *heartbeats* tradicionais, o SWIM separa as funcionalidades de detecção de falha e disseminação de atualização de membros do protocolo de *membership*. Os processos são monitorados por meio de um de sondagem aleatório ponto-a-ponto. O tempo esperado para a primeira detecção de cada falha de processo e o carregamento de mensagem esperado por membro não variam com o tamanho do grupo. As informações sobre alterações de associação, como junções de processos, desistências e falhas, são propagadas por meio de carona em mensagens de *ping* e *acks*.

Em [Urban et al. 2003] dois algoritmos de difusão atômica foram comparados usando detectores de falhas não confiáveis e protocolos de gerenciamento de grupos. Para os testes com detector de falhas, foi utilizado o algoritmo clássico de Chandra e Toueg em conjunto com o detector $\Diamond S$, definidos em [Chandra and Toueg 1996].

Falcon [Leners et al. 2011] é um detector de falhas que utiliza uma rede de módulos de espionagem ou espiões que usam informações internas para saber se os módulos do sistema estão ativos. Se um módulo parece ter falhado, os espiões o interrompem definitivamente para que o detector possa relatar que o processo está falho sem cometer equívocos. O sistema foi implementado e avaliado no monitoramento de aplicação, sistema operacional, máquina virtual e *switches*.

8. Conclusão

Este trabalho apresentou uma proposta de implementação do vCube para diagnóstico de falhas em sistemas assíncronos. Considerando a possibilidade de falsas suspeitas em razão da variação nos atrasos de execução dos processos e/ou da transmissão de mensagens pela rede, o processo correto que se identifica suspeito por outro processo interrompe a sua execução e deixa o sistema. Esta estratégia melhora a acurácia dos detectores, pro-

priedade apresentada por [Chandra and Toueg 1996], favorecendo a solução de diversos problemas clássicos em sistemas distribuídos.

O vCube foi comparado com uma versão clássica de todos-para-todos. Para simular as falsas suspeitas usando o *framework* Java Neko, um novo mecanismo foi incorporado ao simulador. Tal mecanismo permite configurar quando um processo deve suspeitar de outro, independente do seu estado atual. Falsas suspeitas também pode ser geradas variando-se os tempos de transmissão da rede simulada com os mecanismos já existentes no simulador. Por utilizar uma estratégia de testes hierárquica, o vCube possui uma maior latência de propagação de eventos no sistema (diagnóstico). Porém, em termos de tempo de execução e número de mensagens, ficou demonstrado que o vCube utiliza um número consideravelmente menor de mensagens, o que reduz também o tempo de execução das rodadas de teste.

Como trabalhos futuros, pretende-se comparar o vCube com outras soluções descritas na Seção 7. Além disso, espera-se poder utilizar o detector proposto nas soluções de difusão de mensagem apresentada em [Rodrigues et al. 2014] e de exclusão mútua distribuída proposta em [Rodrigues et al. 2018] para avaliar o impacto no desempenho.

Agradecimentos

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

Referências

Brawerman, A. and Jr., E. P. D. (2001). An isochronous testing strategy for hierarchical adaptive distributed system-level diagnosis. *J. Electron. Test.*, 17(2):185–195.

Camargo, E. T. d. and Duarte, E. P. (2018). Running resilient mpi applications on a dynamic group of recommended processes. *Journal of the Brazilian Computer Society*, 24:1–16.

Chandra, T. D., Hadzilacos, V., and Toueg, S. (1996). The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722.

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

Das, A., Gupta, I., and Motivala, A. (2002). Swim: scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks*, pages 303–312.

de Araujo, J. P., Arantes, L., Duarte, E. P., Rodrigues, L. A., and Sens, P. (2017). A publish/subscribe system using causal broadcast over dynamically built spanning trees. In *29th SBAC-PAD*, pages 161–168.

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 Jr., E., Rodrigues, L., Camargo, E., and Turchetti, R. (2022). O elo perdido: Um modelo de diagnóstico distribuído para a implementação de detectores de falhas não confiáveis. In *Anais do XXIII WTF*, pages 29–42, Porto Alegre, RS, Brasil. SBC.

Duarte Jr, E. P., Rodrigues, L. A., Camargo, E. T., and Turchetti, R. (2022). A distributed system-level diagnosis model for the implementation of unreliable failure detectors. *arXiv preprint arXiv:2210.02847*.

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

Jr., E. D. and Mattos, G. (2000). Diagnóstico em redes de topologia arbitrária: Um algoritmo baseado em inundação de mensagens. In *Anais do II Workshop de Testes e Tolerância a Falhas*, pages 82–87, Porto Alegre, RS, Brasil. SBC.

Larrea, M., Arevalo, S., and Fernndez, A. (1999). Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In Jayanti, P., editor, *Distributed Computing*, pages 34–49, Berlin, Heidelberg. Springer Berlin Heidelberg.

Larrea, M., Fernandez, A., and Arevalo, S. (2000). Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000*, pages 52–59.

Leners, J. B., Wu, H., Hung, W.-L., Aguilera, M. K., and Walfish, M. (2011). Detecting failures in distributed systems with the falcon spy network. In *23rd ACM Symposium on Operating Systems Principles*, SOSP '11, page 279–294.

Masson, G. M., Blough, D. M., and Sullivan, G. F. (1996). *System Diagnosis*, page 478–536. Prentice-Hall, Inc.

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

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

Rodrigues, L. A., Duarte Jr, E. P., and Arantes, L. (2014). Arvores geradoras mınimas distribuıdas e autonômicas. In *Anais do XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos (SBRC)*, Porto Alegre, RS, Brasil. SBC.

Sergent, N., Defago, X., and Schiper, A. (2001). Impact of a failure detection mechanism on the performance of consensus. In *Proceedings 2001 Pacific Rim International Symposium on Dependable Computing*, pages 137–145.

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 *Proceedings 15th International Conference on Information Networking*, pages 503–511.

Urban, P., Shnayderman, I., and Schiper, A. (2003). Comparison of failure detectors and group membership: performance study of two atomic broadcast algorithms. In *Int'l Conf. on Dependable Systems and Networks*, pages 645–654.

Ziwich, R., Duarte, E., and Albini, L. (2005). Distributed integrity checking for systems with replicated data. In *11th International Conference on Parallel and Distributed Systems (ICPADS'05)*, volume 1, pages 363–369 Vol. 1.