Uma Comparação Experimental de Detectores de Falhas não Confiáveis em Sistemas Parcialmente Síncronos

Luiz Antonio Rodrigues[1], Elias P. Duarte Jr.[2]

[1]UNIOESTE - Universidade Estadual do Oeste do Paraná
Colegiado de Informática
Rua Universitária, 2069. Jardim Universitário.
Caixa Postal 711 - CEP 85819-110 Cascavel, PR
larodrigues@unioeste.br

[2]UFPR - Universidade Federal do Paraná
Departamento de Informática
Centro Politécnico, Jardim das Américas
Caixa Postal 19081 - CEP 81531-980 Curitiba, PR
elias@inf.ufpr.br

Resumo. *Este artigo apresenta resultados experimentais obtidos a partir da simulação dos algoritmos de Chandra e Toueg e Larrea et. al. propostos para a implementatção de detectores de falhas não confiáveis em sistemas parcialmente síncronos. Os detectores de falhas são abstrações propostas originalmente para permitir o consenso em sistemas distribuídos assíncronos. Dada a impossibilidade do consenso em tais sistemas, é frequente o uso de modelos parcialmente síncronos, isto é com limites para os atributos temporais, ainda que estes limites não sejam conhecidos. O simulador Neko foi utilizado na implementação. As métricas de avaliação são a latência de detecção e o número de mensagens utilizadas pelos detectores.*

1 Introdução

Um sistema distribuído consiste de uma coleção de processos independentes que colaboram para a execução de uma tarefa e se comunicam através de troca de mensagens [6,8]. Uma das vantagens dos sistemas distribuídos é a sua redundância intrínseca, propiciada pela utilização de múltiplos nós. A perda de um nó não implica na parada total do sistema, que pode substituir o nó falho por um nó sem falha. Assim, é possível organizar um sistema distribuído para que ele seja tolerante a falhas [7]. Um componente básico de um sistema com essa característica é a detecção de falhas, com a qual o sistema pode tomar decisões para continuar sua execução [1].

Um dos problemas mais importantes em sistemas distribuídos é o acordo ou consenso [10,11]. O consenso tem diversas aplicações práticas, como em banco de dados distribuídos. Um conjunto de processos deve entrar em um acordo sobre a execução de uma determinada tarefa, por exemplo, a modificação de um valor mantido replicado em memória persistente por nós geograficamente dispersos. Lynch e outros [9] provaram que o consenso é impossível em sistemas distribuídos assíncronos sujeitos

à falha de um processo. Em um sistema assíncrono não há limites para os atributos temporais. Em geral são considerados dois desses atributos: a velocidade de execução dos processos e o atraso de mensagens nos canais de comunicação. Ao contrário dos sistemas assíncronos, nos sistemas ditos síncronos há limites conhecidos para esses atributos. Entretanto, a maioria dos sistemas reais não se encaixa no modelo síncrono.

A partir da prova em [9], conhecida como a *impossibilidade FLP*, passaram a ser propostas estratégias que consideram um modelo de sistema entre o síncrono e o assíncrono, chamado de modelo *parcialmente síncrono*. Neste modelo, os atributos temporais têm limites, ainda que não conhecidos. Por exemplo, em [12] são considerados dois tipos de sistemas parcialmente síncronos. No primeiro, os limites dos atributos temporais existem, mas são desconhecidos. No segundo modelo, os limites são conhecidos, mas apenas ocorrem após um intervalo de estabilização, que tem duração desconhecida. Já em [2] é considerado um modelo parcialmente síncrono no qual os atributos temporais têm limites, mas os limites são desconhecidos e só valem após um intervalo de estabilização também desconhecido.

A prova da impossibilidade FLP tem como ponto central o fato do algoritmo do consenso poder levar os nodos a decidirem por um entre dois valores, considerando que um dos nodos está falho. Porém este nodo falho pode, na verdade, estar lento, e pode executar ações que conduzam tanto a uma ou outra decisão. Tendo por objetivo dotar o sistema de um oráculo que forneça aos nodos informações sobre o estado de todos os nodos, Chandra e Toueg propuseram os detectores de falhas de 1996. Eles classificaram os detectores de acordo com sua *completude (completeness)* e *acurácia (accuracy)*, e mostraram que, de acordo com estas propriedades, é possível realizar o consenso em um sistema assíncrono dotado de detectores de falhas.

Neste trabalho é apresentada uma comparação experimental obtida a partir da simulação de dois detectores de falhas para sistemas parcialmente síncronos. O primeiro detector é o proposto por Chandra e Toueg em [2] e pode ser considerado como o algoritmo da força-bruta, no qual todos os nodos mandam periodicamente mensagens para todos os demais, permitindo assim que todos os nodos conheçam o estado de todos os outros. O segundo detector foi proposto por Larrea e outros em [3], e utiliza um número bem menor de mensagens: os nodos foram um anel virtual sobre o qual são trocadas informações sobre os estado dos nodos. Foi utilizado o simulador Neko para a implementação dos detectores. As métricas de simulação foram a latência de detecção e o número de mensagens utilizadas pelos detectores.

O restante deste trabalho está organizado da seguinte maneira. Na Seção 2 são descritos os dois detectores de falhas implementados. A Seção 3 apresenta o *framework* de simulação utilizado, o Neko. A implementação e os resultados experimentais são apresentados na Seção 4. A conclusão segue na Seção 5.

2 Detectores de Falhas

Um sistema distribuído pode ser classificado em síncrono, assíncrono ou parcialmente síncrono. Em um sistema síncrono é possível criar detectores de falhas baseados em eventos de temporização (*timeout*). Estes detectores são ditos confiáveis (*reliable*). Já

em um sistema assíncrono, os detectores são classificados como não confiáveis (*unreliable*), pois não há parâmetros de tempo definidos. Por esta razão, não é possível distinguir entre um processo falho e um processo lento. Assim, um detector de falhas não confiável mantém uma lista local de processos possivelmente falhos, isto é, suspeitos. Estas informações são baseadas no monitoramento dos demais processos do sistema. A característica de ser não confiável está ligada à possibilidade de cometer erros, como a suspeita de um processo não falho ou a não suspeita de um processo falho.

Baseado no comportamento dos detectores não confiáveis e considerando as propriedades de completude e acurácia, Chandra e Toueg [2] definiram oito diferentes classes de detectores. A completude exige que todo processo falho seja suspeito pelos processos corretos em algum momento e a acurácia restringe a possibilidade de erros cometidos pelo detector para o estado dos processos. Seguindo essa classificação, Larrea e outros [3] propuseram um algoritmo da classe completude forte (*strong completeness*), em que todo processo falho é permanentemente suspeito por cada processo correto.

2.1 Algoritmo CT (Chandra e Toueg)

De acordo com Chandra e Toueg [2], o requisito para que um algoritmo de detecção seja classificado com completude forte é que, em algum momento, todos os processos que estão falhos serão suspeitos por cada processo correto. Uma solução para satisfazer esta necessidade é fazer com que cada processo *p* envie periodicamente uma mensagem "I-AM-ALIVE" para todos os outros processos (Figura 1). Cada processo, recebendo as mensagens de todos os outros, poderá manter uma lista de estado dos processos.

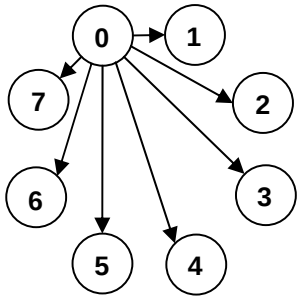


Figura 1. Encaminhamento de Mensagens no Algoritmo CT

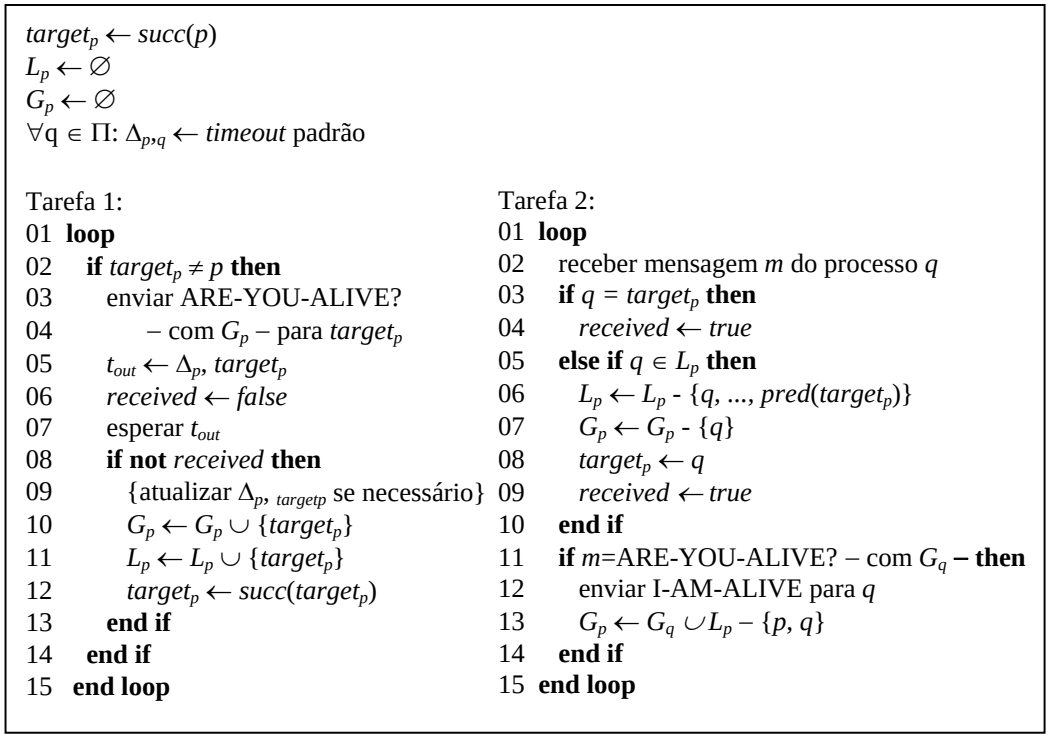
Se *p* não receber a mensagem de *q* dentro de um intervalo de tempo $\Delta_p(q)$, *q* será considerado suspeito por *p*. Se *p* receber a mensagem de um processo *q* que é considerado suspeito, *p* remove *q* da sua lista de suspeitos e incrementa o intervalo de temporização $\Delta_p(q)$. Neste algoritmo, cada processo envia N-1 mensagens, totalizando $N(N\text{-}1)$ mensagens para cada rodada de teste.

2.2 Algoritmo LFA (Larrea, Fernández e Arévalo)

Embora o algoritmo proposto por Chandra e Toueg [2] consiga prover completude forte, ele exige um grande número de troca de mensagens. Por esta razão, Larrea, Fernández e

Arévalo [3] propuseram um novo algoritmo baseado na mesma abordagem, mas que necessita de uma quantidade muito menor de trocas. Este número é reduzido em função da organização do sistema, que passa a ser em anel e não mais em todos-para-todos. Neste algoritmo, cada processo p mantém duas listas: L_p e G_p. L_p é uma lista local de processos suspeitos entre p e o processo que p está monitorando ($target_p$). G_p é uma lista global, utilizada para armazenar todos os processos suspeitos no sistema. É através de G_p que a completude forte é alcançada. No modelo proposto, processos podem falhar por colapso e esta falha é permanente, isto é, um processo em colapso não se recupera.

O algoritmo LFA está reproduzido na Figura 2. A cada rodada, o processo p envia uma mensagem "ARE-YOU-ALIVE?" contendo a sua lista G_p para o processo que está monitorando no anel ($target_p$). Quando $target_p$ recebe esta mensagem, atualiza a sua lista global, considerando a lista recebida e a sua lista local, e envia uma mensagem "I-AM-ALIVE" para p (linhas 11-13, tarefa 2). Se p não receber a mensagem de resposta de $target_p$ dentro de um intervalo pré-definido, este é incluído como suspeito na listas local e global e p passa a monitorar o próximo nó do anel, isto é, o sucessor de $target_p$.



$target_p \leftarrow succ(p)$
$L_p \leftarrow \varnothing$
$G_p \leftarrow \varnothing$
$\forall q \in \Pi$: $\Delta_{p,q} \leftarrow$ *timeout* padrão

Tarefa 1:
01 **loop**
02 **if** $target_p \neq p$ **then**
03 enviar ARE-YOU-ALIVE?
04 – com G_p – para $target_p$
05 $t_{out} \leftarrow \Delta_p$, $target_p$
06 $received \leftarrow false$
07 esperar t_{out}
08 **if not** *received* **then**
09 {atualizar Δ_p, $_{targetp}$ se necessário}
10 $G_p \leftarrow G_p \cup \{target_p\}$
11 $L_p \leftarrow L_p \cup \{target_p\}$
12 $target_p \leftarrow succ(target_p)$
13 **end if**
14 **end if**
15 **end loop**

Tarefa 2:
01 **loop**
02 receber mensagem m do processo q
03 **if** $q = target_p$ **then**
04 $received \leftarrow true$
05 **else if** $q \in L_p$ **then**
06 $L_p \leftarrow L_p$ - {q, ..., $pred(target_p)$}
07 $G_p \leftarrow G_p$ - {q}
08 $target_p \leftarrow q$
09 $received \leftarrow true$
10 **end if**
11 **if** m=ARE-YOU-ALIVE? – com G_q **– then**
12 enviar I-AM-ALIVE para q
13 $G_p \leftarrow G_q \cup L_p - \{p, q\}$
14 **end if**
15 **end loop**

Figura 2. Algoritmo LFA [3]

A Figura 3 apresenta um cenário com 8 nós obtido após a falha dos nós 1 (n_1), 4 (n_4) e 5 (n_5). No exemplo, a lista local L_0 do nó n_0 indica que entre n_0 e $target_0$ (n_2) o nó n_1 é suspeito. Já a lista global G_0 informa que os nós suspeitos em todo o sistema são n_1, n_4 e n_5. Neste exemplo é possível verificar que, embora cada nó mantenha uma lista local diferente, a lista global deverá ser a mesma, garantindo a completude forte.

Neste algoritmo, cada processo testa periodicamente um outro processo. Como este teste utiliza duas mensagens, são necessárias 2*N* mensagens a cada rodada.

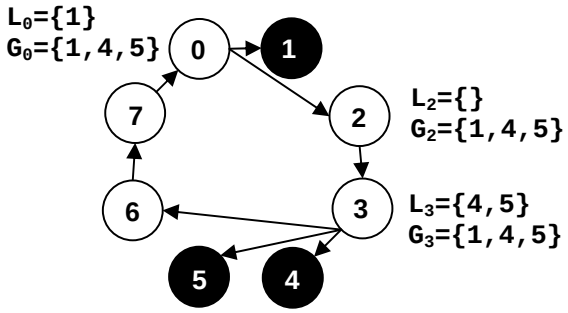


Figura 3. Exemplo de Conjunto de Testes e Estrutura de Dados no LFA

Detectado um nó falho, a latência do diagnóstico dependerá do procedimento adotado. A Figura 4 apresenta três modelos de encaminhamento de mensagens: (a) simétrico, (b) assimétrico e (c) *multicast*. As setas no sentido horário representam as mensagem de teste e as tracejadas as mensagens de diagnóstico.

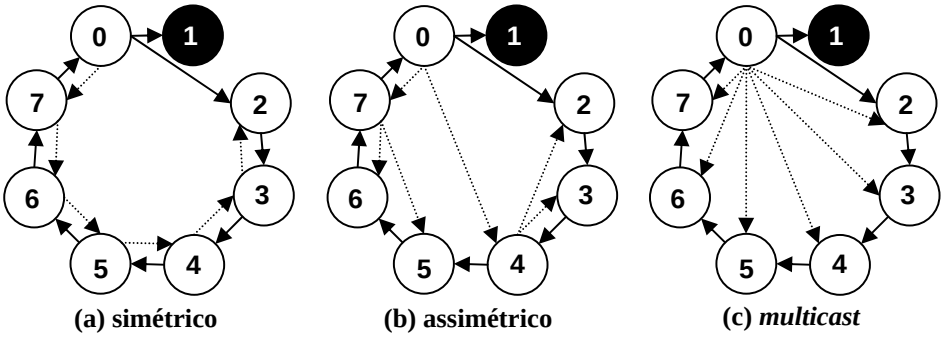


Figura 4. Modelos de Encaminhamento das Mensagens do Algoritmo LFA

No modelo simétrico, o resultado do teste é encaminhado na direção reversa dos testes. Este mecanismo utiliza um número mínimo de mensagens, mas possui a maior latência (tempo entre o evento de falha e a detecção em todos os processos corretos). O modelo assimétrico envia mensagens utilizando caminhos redundantes e diferentes da direção dos testes. Isto aumenta o número de mensagens, mas reduz significativamente a latência. Por fim, tem-se o modelo de *multicast*, que reduz ao mínimo a latência, mas exige um grande número de mensagens. Esta quantidade de mensagens pode ser minimizada com o uso de recursos da rede, como o *multicast* da *Ethernet*. Entretanto, este serviço é restrito às redes locais e não está disponível quando duas ou mais redes estão interligadas por roteadores.

O intervalo de latência no modelo simétrico é de NT_r, onde T_r representa o intervalo de tempo entre cada rodada de teste. Esta latência pode ser diminuída se cada processo, ao receber informações sobre um processo falho, encaminhar imediatamente o novo vetor de estados ao seu vizinho, sem aguardar até a próxima rodada de testes. Outra forma de diminuir a latência sem aumentar significativamente a quantidade de

mensagens é enviar a lista de suspeitos nos dois sentidos do anel. Isso reduz a latência aproximadamente pela metade.

3 O Framework Neko

Neko é um *framework* 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 (Figura 5). 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.

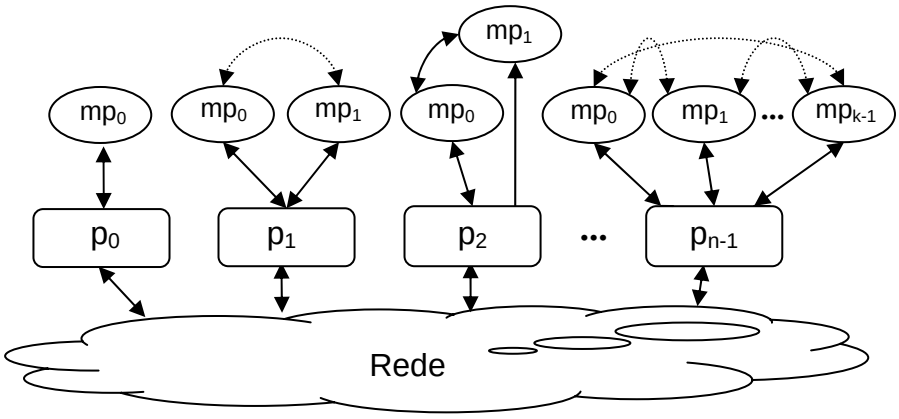


Figura 5. Arquitetura do Neko

O segundo componente da arquitetura do Neko é a rede. A infra-estrutura de comunicação pode ser controlada de diferentes maneiras, podendo ser instanciada a partir de uma coleção de redes previamente definidas (como as redes TCP/IP reais ou *Ethernet* simulada) ou ainda por meio da extensão e adição de novos modelos de rede.

3.1 Suporte a Simulação de Defeitos

A simulação de defeitos no Neko é realizada através da adaptação sugerida em [4] e ilustrada na Figura 6. 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 esta classe, que verifica se o processo está em colapso. Se estiver, a mensagem é descartada. O mesmo acorre para mensagens recebidas da rede. A aplicação pode fazer uma consulta ao estado do

processo e, em caso de falha, parar sua execução através do método `wait`. Quando o processo retornar da falha, o método `notify` o avisará para continuar sua execução. Neste momento, cabe ao desenvolvedor da aplicação definir que tipo de colapso será simulado: por pausa, com amnésia parcial ou com amnésia total. Por pausa, o processo continua a sua execução do ponto em que estava antes da falha, mantendo o estado original. Com amnésia parcial, o processo mantém alguns parâmetros e com amnésia total, o processo volta ao estado inicial.

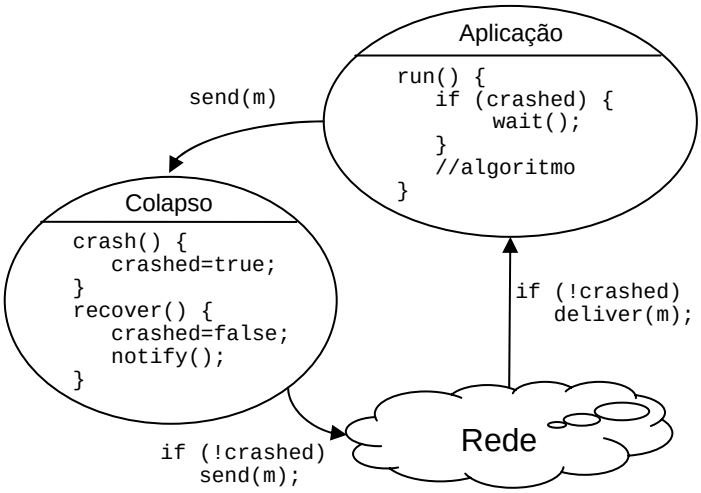


Figura 6. Simulação de Defeitos de Colapso no Neko

Além dos defeitos de colapso citados, as adaptações de [4] permitem simular colapso por parada (*crash*, quando o processo não volta a execução), omissão de mensagens e particionamento de rede.

4 Implementação e Resultados

Os algoritmos CT e LFA foram implementados em Java e simulados no Neko. A arquitetura do sistema está representada na Figura 7. Cada nó possui uma instância do algoritmo e, para os nós que apresentarão falhas, é incluída uma instância da classe de defeitos de colapso por parada. Com isso, processos falham e não mais se recuperam.

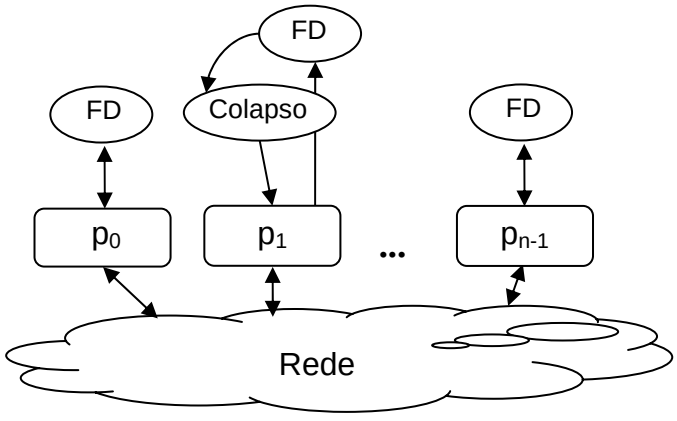


Figura 7. Modelo de Simulação dos Algoritmos no Neko

Foram realizados dois testes: o primeiro com os algoritmos CT e LFA simétrico e o segundo com o LFA *multicast*. Foram utilizados 8 nós e a duração dos testes foi de 5000s. O intervalo entre cada rodada dos algoritmos foi fixado em 60s. Uma falha foi configurada nos nós *p1*, *p4* e *p5* para ser iniciada 500s após o início da simulação. Considerando o intervalo de testes de 60s, a 9ª rodada aconteceu no intervalo *t*=540s, ou seja, 40s após a falha. Definiu-se um intervalo de temporização fixo de 3s. A rede utilizada acrescenta um tempo de propagação fixo de 0,3s às mensagens enviadas.

A Figura 8 ilustra, através da ferramenta *LogView* do Neko, o comportamento do algoritmo CT no primeiro teste. À esquerda está a troca de mensagens na 2ª rodada (execução sem falha) e à direita as mensagens enviadas na 9ª rodada, isto é, a 1ª após a falha de *p1*, *p4* e *p5*. A detecção dos processos falhos foi realizada pelos demais processos ao final desta rodada (*t*=503).

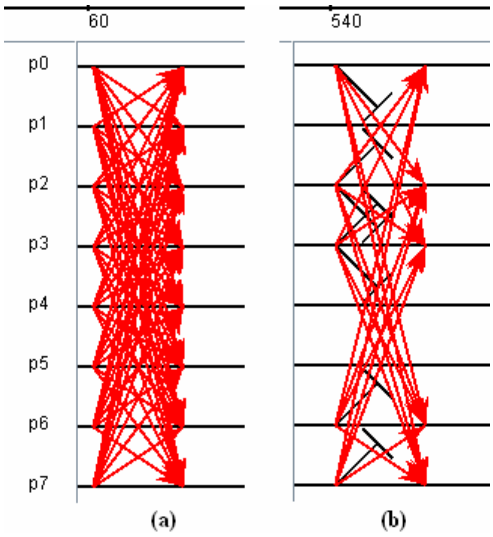


Figura 8. Teste1 - Execução do Algoritmo CT

A Figura 9 ilustra a execução do algoritmo LFA em 3 momentos: (a) execução normal, (b) após a falha e (c) no diagnóstico. Após a ocorrência da falha em *t*=500, *p0* suspeita *p1* em *t*=543. Neste mesmo instante *p3* suspeita *p4*. A próxima rodada foi realizada em *t*=600s. Neste momento, *p0* testa *p2* enviando sua lista de suspeitos, que inclui *p1*. Da mesma forma, *p3* testa *p5* que embora esteja em colapso, é o próximo nó no anel depois de *p4*. Em seguinda, *p2* recebe a informação de falha de *p1* em *t*=600,3. A falha de *p5* é detectada por *p3* em *t*=603. Na próxima rodada, *p3* testa *p6* e o avisa das falhas de *p4* e *p5*. O processo continua até que em *t*=780,3 todos recebem a mensagem com informação de falha dos nós *p1*, *p4* e *p5*.

Com base nestes valores, o intervalo de tempo entre a ocorrência da falha e a detecção foi o mesmo nos dois algoritmos, sendo determinado pelo *timeout*, que neste teste foi fixado em 3s. A latência da detecção, isto é, o intervalo de tempo entre a detecção e a suspeita das falhas em todos os processos foi de 3s no algoritmo CT e de 237,3s no LFA. No CT o diagnóstico depende apenas do *timeout*. No LFA, além do *timeout*, deve-se considerar o número de rodadas necessárias até que a informação se

propague no anel. Este tempo aumenta linearmente de acordo com o número de nós envolvidos. No teste realizado, foram necessárias 5 rodas para um total de 5 nós não falhos.

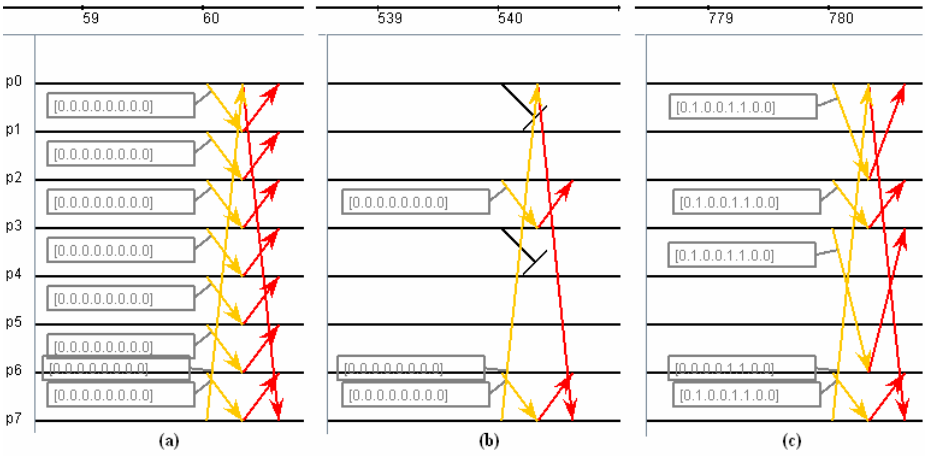


Figura 9. Teste1 - Execução do Algoritmo LFA Simétrico

Em relação à quantidade total de mensagens enviadas, o algoritmo LFA apresentou um ganho significativo se comparado ao CT. Foram enviadas 891 mensagens pelo LFA e 3129 no CT, isto é, 5 vezes menos mensagens no LFA para um total de 83 rodadas.

Para comparar a eficiência do algoritmo LFA foi realizado um segundo teste (Teste2). Neste teste, quando um nó detecta a falha do outro nó ele atualiza a lista local e global e envia imediatamente uma mensagem a todos os outros processos com a sua lista atualizada. Esse procedimento visa melhorar a latência de diagnóstico do algoritmo. A Figura 10 ilustra a seqüência de mensagens desde a detecção (t=503) até o diagnóstico (t=603,3). A latência no LFA *multicast* foi de 57,3s. Isto representa um ganho de 75,9% em relação à latência do simétrico. A quantidade de mensagens foi ligeiramente superior (924), mas ainda muito menor que no CT.

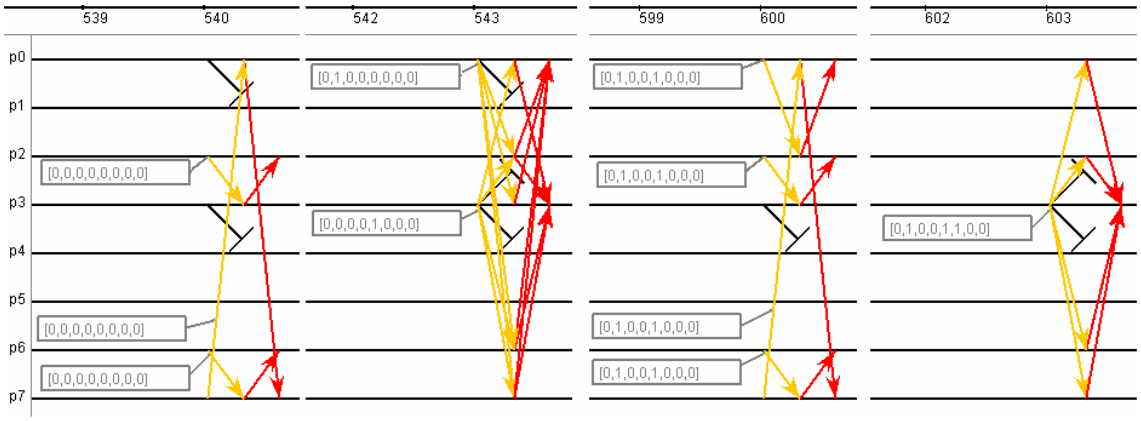


Figura 10. Teste2 - Execução do Algoritmo LFA *Multicast*

5 Conclusão

Neste trabalho foi descrita uma comparação experimental de dois detectores de defeitos não confiáveis propostos pra sistemas parcialmente síncronos. O *framework* Neko foi utilizado para realizar a simulação dos detectores em um ambiente de rede composto por oito nós. Os resultados experimentais permitem uma comparação dos dois detectores distribuídos com relação à eficiência na detecção dos processos falhos e a latência no diagnóstico. Os dois algoritmos mostraram-se igualmente eficientes para detectar falhas em um único nó. Entretanto, no algoritmo CT a falha de múltiplos nós não impacta na latência de detecção, ao contrário do LFA, em que a detecção de todos os nós falhos dependerá da quantidade de nós não falhos no anel. Essa latência pode ser minimizada com a utilização de *multicast*, o que reduz significativamente a latência do LFA e necessita de uma quantidade reduzida de mensagens extras.

Referências

[1] Duarte Jr., E. P., Nanya, T., "A Hierarchical Adaptive Distributed System-Level Diagnosis Algorithm", *IEEE Transactions on Computers*, Vol. 47, No. 1, pp. 34-45, 1998.

[2] Chandra, T. D., Toueg, S., "Unreliable Failure Detectors for Reliable Distributed Systems", *Journal of the ACM*, Vol. 43, No. 2, pp. 225-267, 1996.

[3] Larrea, M., Fernández, A., Arévalo, S., "On the Implementation of Unreliable Failure Detectors in Partially Synchronous Systems", *IEEE Transactions on Computers*, Vol. 53, No. 7, pp. 815-828, 2004.

[4] Rodrigues, L., Jansch-Porto, I., "Ampliação do Framework Neko para a Simulação de Defeitos em Algoritmos Distribuídos", *7th International Information and Telecommunication Technologies Symposium*, Foz do Iguaçu, 2008.

[5] Paxson, V, Allman, M., "Computing TCP's Retransmission Timer", *RFC 2988*, 2000.

[6] Guerraoui R., Rodrigues L., Introduction to Reliable Distributed Programming, Springer-Verlag, 2006.

[7] Jansch-Porto, I. "Fundamentos de Tolerância a Falhas," *Revista de Informática Teórica e Aplicada (RITA)*, Vol. 1, No. 3, 1991.

[8] Greve, F. "Protocolos Fundamentais para o Desenvolvimento de Aplicações Robustas," *Minicurso do Simpósio Brasileiro de Redes de Computadores (SBRC)*, 2005.

[9] Fischer, M., Lynch, N., Paterson, M. "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the ACM*, Vol. 32, No. 2, pp. 374-382, 1985.

[10] Turek D., Shasha M, "The Many Faces of Consensus in Distributed Systems," *IEEE Computer*, Vol. 25, No. 6, pp. 8-17, 1992.

[11] Pease, M., Shostak, R., Lamport, L. "Reaching Agreement in the Presence of Faults," *Journal of the ACM*, Vol. 27, No. 2, pp. 228-234, 1980.

[12] Dwork, C., Lynch, N., Stockmeyer, L. "Consensus in the Presence of Partial Synchrony," *Journal of the ACM*, Vol. 35, No. 2, pp. 288-323, 1988.