Exclusão Mútua Distribuída e Robusta para k Recursos Compartilhados

Luiz A. Rodrigues[1,2], Elias P. Duarte Jr.[2] e Luciana Arantes[3]

[1] Colegiado de Ciência da Computação – Universidade Estadual do Oeste do Paraná
Caixa Postal 801 – 85819-110 – Cascavel – PR – Brasil

[2]Departamento de Informática – Universidade Federal do Paraná
Caixa Postal 19.081 – 81531-980 – Curitiba – PR – Brasil

[3]Laboratoire d'Informatique - Université Pierre et Marie Currie
CNRS/INRIA/REGAL – Place Jussieu, 4 – 75005 – Paris, France

luiz.rodrigues@unioeste.br, elias@inf.ufpr.br, luciana.arantes@lip6.fr

Abstract. *This paper presents a robust k-mutual exclusion solution in distributed systems subject to crash failures. The proposed algorithm is based on Raymond's algorithm. To propagate the request messages in a scalable way, we propose a minimum spanning tree algorithm. The tree is created in a distributed manner, based on information provided by an auxiliary monitoring system. The solution improves the efficiency of the Raymond's algorithm in obtaining resources and works correctly for up to n-1 faulty processes.*

Resumo. *Este trabalho apresenta uma solução robusta de k-exclusão mútua em sistemas distribuídos sujeitos a falhas de crash. O algoritmo proposto é baseado no algoritmo de Raymond. Para propagar as mensagens de requisição de forma escalável, foi desenvolvido um algoritmo de árvore geradora mínima. A árvore é criada de forma distribuída, com base nas informações fornecidas por um mecanismo auxiliar de monitoramento de estados dos processos. A solução proposta melhora a eficiência do algoritmo de Raymond na obtenção de recursos e garante o seu funcionamento para até n-1 processos falhos.*

1. Introdução

Um sistema distribuído consiste de um conjunto finito Π de $n \geq 2$ processos independentes $\{p_0, ..., p_{n-1}\}$ que se comunicam usando troca de mensagens, colaborando para a realização de alguma tarefa. Uma das vantagens dos sistemas distribuídos é o compartilhamento de recursos (dispositivos, programas e dados). No entanto, cada processo pode solicitar o acesso a um recurso compartilhado de forma arbitrária. Na programação concorrente, este acesso é normalmente realizado em uma área de código denominada *seção crítica* [Boehm e Adve 2012]. Uma questão relevante é como organizar o acesso concorrente garantindo duas propriedades principais: a segurança (*safety*), que garante que somente um solicitante obtenha o recurso de cada vez e a propriedade de progressão (*liveness*), na qual todos os interessados em um recurso consigam obtê-lo em um tempo finito. A solução para este problema é chamada de *exclusão mútua* [Lamport 1978, Ricart e Agrawala 1981, Raynal e Beeson 1986].

Existem basicamente duas abordagens clássicas para implementar a exclusão mútua em sistemas distribuídos. A primeira é por passagem de permissão (*token*) e

a segunda é através de solicitação de permissão [Raynal 1991]. Com passagem de permissão, somente o processo que detém o *token* pode acessar o recurso compartilhado [Le Lann 1977, Suzuki e Kasami 1985, Raymond 1989b, Naimi et al. 1996]. A permissão pode circular entre os processos seguindo uma organização lógica em anel, por exemplo. Já na solicitação de permissão, cada processo que deseja fazer uso do recurso deve solicitar a todos os demais a permissão para utilizá-lo [Ricart e Agrawala 1981, Sanders 1987]. Para o caso em que um único recurso é compartilhado, a solução trivial é enviar uma mensagem de solicitação a cada um dos outros $n - 1$ processos do sistema e aguardar as respostas [Bertsekas et al. 1991]. Se todos os processos responderem positivamente, o solicitante obtém a permissão. Uma variação da exclusão mútua é a k-exclusão mútua, na qual k recursos são compartilhados entre os n processos [Raymond 1989b, Bulgannawar e Vaidya 1995]. Neste caso, o processo solicitante precisa aguardar, no mínimo, $n - k$ respostas.

Um fator relevante que tem impacto direto na escalabilidade de um algoritmo de exclusão mútua distribuída é o mecanismo de disseminação de mensagens. Uma abordagem simples, empregada na maioria das propostas citadas anteriormente, é utilizar mensagens de *broadcast*. No entanto, em redes nas quais este mecanismo não está disponível, como a Internet, uma maneira eficiente e escalável é empregar uma solução hierárquica, como uma árvore [Avresky 1999].

Outro fator relevante em uma solução distribuída é a possibilidade de ocorrência de falhas. Na exclusão mútua com pedido de permissão, por exemplo, o solicitante precisa ter informações sobre o estado dos processos para não ficar aguardando indefinidamente por respostas daqueles falhos [Bouillaguet et al. 2008]. O mesmo acontece quando se usa passagem de permissão. No caso de falha do processo que possui o *token*, o sistema precisa identificar o problema e gerar um novo *token*. Nas duas situações, uma solução é utilizar um mecanismo de monitoramento que ofereça informações sobre o estado (falho ou sem-falha) dos processos no sistema [Romano e Rodrigues 2009]. Para o problema da exclusão mutua distribuída é necessário um detector perfeito [Delporte-Gallet et al. 2005].

A principal contribuição deste trabalho é uma solução tolerante a falhas de k-exclusão mútua distribuída baseada no modelo com permissão de [Raymond 1989a]. O algoritmo foi adaptado para melhorar a eficiência na obtenção de recursos na presença de até $n - 1$ processos falhos fazendo uso de um mecanismo auxiliar de monitoramento. A segunda contribuição do trabalho é um algoritmo para construção de uma árvore geradora mínima a partir de um nodo fonte qualquer. O algoritmo proposto permite a construção da árvore de forma totalmente distribuída e adaptativa, também com base nas informações de monitoramento e independente da quantidade de processos falhos. Esta árvore é utilizada para propagar as mensagens de requisição, provendo escalabilidade à solução. Os canais são confiáveis, mas processos podem falhar por *crash* e uma falha é permanente. O total de processos é conhecido por todos no início da execução e o sistema é síncrono.

O restante do texto está organizado nas seguintes seções. A Seção 2 discute a exclusão mútua distribuída e o algoritmo de Raymond para k-exclusão mútua. A Seção 3 apresenta o algoritmo de criação da árvore geradora mínima utilizada para a difusão de mensagens. A Seção 4 descreve o algoritmo de k-exclusão mútua proposto neste trabalho. Uma avaliação experimental é apresentada na seção 5. A Seção 6 aborda trabalhos relacionados e Seção 7 apresenta a conclusão e os trabalhos futuros.

2. Exclusão Mútua em Sistemas Distribuídos

A primeira solução para exclusão mútua distribuída baseada em permissão foi apresentada por [Lamport 1978] e utiliza relógios lógicos para determinar a ordem das solicitações. Quando um processo p_i deseja obter acesso ao recurso, envia uma mensagem por *broadcast* para os outros processos e armazena a solicitação em uma fila local. Um processo p_j que recebe uma mensagem de solicitação de p_i, armazena a mensagem na sua fila e retorna uma mensagem de resposta com o *timestamp* atualizado. Assim, p_i pode acessar o recurso quando receber $n-1$ permissões e o seu pedido é o primeiro da fila. Quando libera o recurso, p_i envia uma mensagem *broadcast* de liberação para que os demais processos retirem a solicitação das suas filas, dando oportunidade às demais solicitações pendentes. Cada fase de requisição, resposta e liberação gera $n-1$ mensagens, totalizando $3(n-1)$ mensagens por rodada. O trabalho de [Ricart e Agrawala 1981] aprimorou a solução de Lamport com um algoritmo que requer $2(n-1)$ mensagens. A implementação também utiliza mensagens de requisição e resposta, mas omite as mensagens de liberação. Isto é possível porque quando um processo recebe uma mensagem de solicitação, mas está utilizando o recurso, ele retém a resposta. Assim, quando o processo libera o recurso, ele envia todas as mensagens de resposta adiadas, permitindo que os demais processos tenham a chance de obter o recurso.

Uma extensão do problema da exclusão mútua é a k-exclusão mútua. Nesta categoria, ao invés de um, existem k recursos compartilhados. O objetivo é garantir que, nó máximo, k processos obtenham acesso aos recursos (*safety*) e que todos os processos que solicitarem recursos consigam obtê-lo em um tempo finito (*liveness*). Cada processo pode obter acesso a um único recurso de cada vez. O algoritmo de [Raymond 1989a] soluciona o problema da k-exclusão mútua utilizando uma abordagem de permissões baseada no algoritmo de 1-exclusão mútua de [Ricart e Agrawala 1981]. Quando um processo deseja utilizar o recurso compartilhado, ele envia mensagens de requisição aos $n-1$ outros processos e aguarda por, no mínimo, $n-k$ mensagens de permissão. Se nenhum processo está utilizando ou solicitando recursos, o total de respostas pode chegar a $n-1$. Portanto, no pior caso, são geradas $2(n-1)$ mensagens por solicitação.

A Figura 1 ilustra o comportamento do algoritmo de Raymond em um sistema com 8 processos e 5 recursos. Inicialmente, no tempo t_0 o processo p_0 envia uma mensagem de requisição para todos os demais, solicitando a permissão. Como nenhum deles está utilizando um recurso ou tentando obtê-lo, todos respondem com mensagens de resposta, permitindo que p_0 utilize o recurso em t_1 (o $*$ marca o momento em que p_i obteve o recurso). Em seguida, em t_2 o processo p_1 efetua o pedido de permissão. Da mesma forma, todos os demais processos respondem positivamente, exceto p_0 que está utilizando um recurso. Neste caso, como p_1 deve aguardar por no mínimo $n-k=8-5=3$ respostas, ele também obtém um dos recursos, ainda que p_0 não tenha enviado a resposta. Por fim, quando p_0 libera o recurso, ele envia as respostas referente às solicitações pendentes. Neste exemplo, apenas para p_1 em t_3. Caso p_1 ainda não tivesse obtido respostas suficientes, esta resposta adiada de p_0 poderia completar o total de respostas necessárias.

Intrinsecamente, o algoritmo de Raymond tolera $k-1$ processos falhos. No entanto, cada processo falho degrada a solução, pois pode ser que o número de processos não falhos que não desejam recursos seja insuficiente, fazendo com que o processo solicitante tenha que aguardar um processo sem-falha liberar o recurso e dar a permissão,

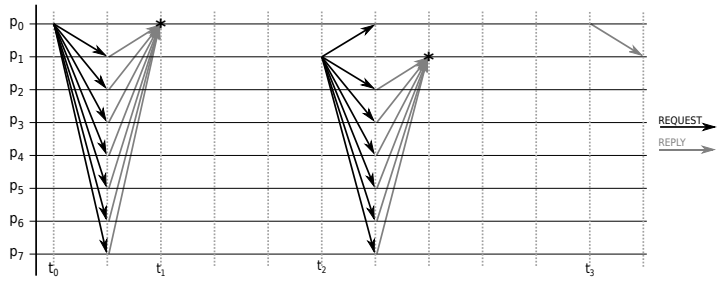


Figura 1. Execução do algoritmo de Raymond para $n = 8$ e $k = 5$ sem falhas.

mesmo que um recurso esteja livre. Voltando ao exemplo anterior, considere que durante a solicitação efetuada por p_1 os processos p_4 a p_7 estejam falhos. O processo p_1 obterá permissão apenas de p_2 e p_3, o que é insuficiente. Neste caso, apenas após a liberação do recurso por p_0 em t_3 é que p_1 conseguirá as três mensagens necessárias para utilizar o recurso, mesmo havendo quatro recursos livres.

3. Monitoramento e Difusão de Mensagens

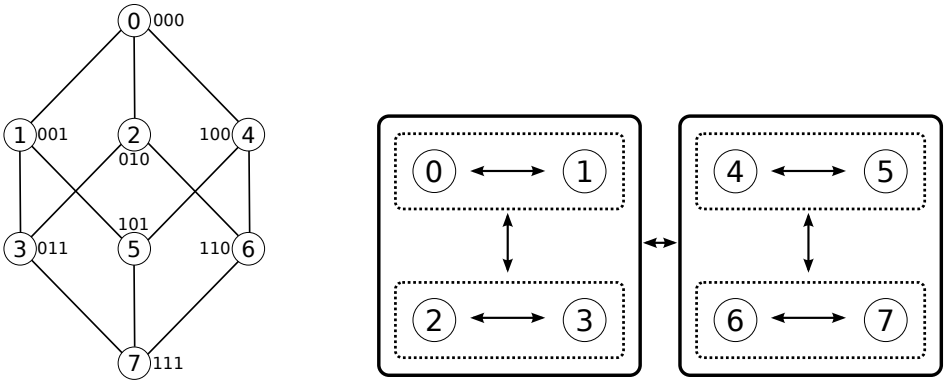
Esta seção está organizada em duas partes. Na primeira é apresentada a estratégia de monitoramento adotada. Na segunda parte é descrito o algoritmo de árvore geradora mínima proposto para a disseminação de mensagens na exclusão mútua.

3.1. Estratégia de Monitoramento

O monitoramento dos processos participantes da exclusão mútua é a base para o funcionamento do mecanismo hierárquico de difusão de mensagens proposto neste trabalho. Além de fazer uso da informação de estado dos processos (falho ou sem-falha), o mecanismo proposto utiliza funções definidas pelo algoritmo de monitoramento Hi-ADSD (*Hierarchical Adaptive Distributed System-Level Diagnosis*) [Duarte Jr. e Nanya 1998].

Considerando um modelo de sistema síncrono, o Hi-ADSD é um algoritmo de diagnóstico distribuído no qual os nodos são interligados logicamente em *clusters* progressivamente maiores, estabelecendo um modelo de testes hierárquico em forma de hipercubo. Um hipercubo de d-dimensões consiste de uma rede com 2^d nodos numerados de 0 a $2^d - 1$. Cada nodo x é identificado pelo código binário $(x_d, ..., x_1)$ do seu identificador. Uma aresta entre dois nodos existe se os seus códigos diferem em um bit, ilustrado pela Figura 2(a).

O modelo de testes do Hi-ADSD é dividido em $\log n$ rodadas, numeradas de 1 a $\log n$. A cada rodada s, um nodo sem-falha testa um nodo no $cluster_s$ e obtém informações sobre os demais nodos naquele $cluster$. Como ilustrado pela Figura 2(b), no primeiro intervalo de testes ($s = 1$), cada nodo testa o *cluster* com um nodo. No segundo intervalo de testes ($s = 2$), o *cluster* com dois nodos. No terceiro intervalo ($s = 3$), com quatro nodos, e assim sucessivamente até que o *cluster* tenha $n/2$ nodos. Em seguida, após mais um intervalo determinado, o processo reinicia. No caso em que todos os nodos são sem-falha, o número de testes executado por cada nodo é $\log n$. A latência de diagnóstico, isto é, o tempo necessário para que todos os nodos sem-falha identifiquem todos os nodos falhos varia entre $\log n$ e $\log^2 n$, no pior caso.



(a) Hipercubo de 3 dim. (b) Organização hierárquica dos testes no Hi-ADSD

Figura 2. Um hipercubo de 3 dimensões e a hierarquia de testes no Hi-ADSD.

Em cada rodada um nodo sem-falha testa o próximo nodo no $cluster_s$ até encontrar um nodo sem-falha ou até testar todos os nodos do *cluster* como falhos. O cálculo dos nodos integrantes de cada *cluster* s e a ordem em que eles são testados por um nodo i qualquer são dados pela função $C_{i,s}$, para $i = 0, .., n-1$. Esta função é definida como:

$$C_{i,s} = < i \oplus 2^{s-1}, C_{i \oplus 2^{s-1}, s-1}, C_{i \oplus 2^{s-1}, s-2}, ..., C_{i \oplus 2^{s-1}, 1} >$$

Como exemplo, o *cluster* 3 de n_0 é dado por:

$$C_{0,3} = < 4, C_{4,2}, C_{4,1} > = < 4, 6, C_{6,1}, 5 > = < 4, 6, 7, 5 >$$

3.2. Algoritmo de Árvore Geradora Mínima

Seja $G = (V, E)$ um grafo conexo e não-direcionado com V vértices e E arestas. Uma árvore geradora (*spanning tree*) é um grafo conexo e acíclico que contém todos os vértices de G. Se as arestas possuem pesos, uma *árvore geradora mínima* é aquela cujo a soma dos pesos das arestas é mínima. Se cada aresta possui um peso diferente, existe uma única árvore mínima. Se todas as arestas possuem o mesmo peso, todas as árvores do grafo são mínimas. Em grafos, os dois algoritmos clássicos para a obtenção de árvores geradoras mínimas são o algoritmo de [Kruskal Jr. 1956] e o proposto por [Prim 1957].

O algoritmo de Kruskal inicialmente cria uma *floresta* na qual cada vértice é uma árvore. A cada passo, as árvores são conectadas entre si através das arestas de menor peso. As arestas que não interligam duas árvores são descartas, evitando ciclos. Ao final, uma única componente conexa é gerada e esta constitui a árvore geradora mínima do grafo. O algoritmo de Prim utiliza uma abordagem diferente, que emprega cortes mínimos para escolher as arestas de menor peso e incluí-las na árvore. Além destas soluções centralizadas, algumas propostas foram definidas para soluções distribuídas, baseadas em troca de mensagens. A primeira delas foi definida por [Gallager et al. 1983]. O processo é semelhante ao utilizado por Kruskal. Inicialmente cada nodo é uma árvore. A cada nível, um nodo é eleito líder e uma aresta de peso mínimo que o interliga a um nodo em outra árvore é adicionada. O processo é repetido até formar uma única componente conexa.

O algoritmo proposto neste trabalho dispensa os cálculos dos algoritmos descritos anteriormente, pois faz uso da topologia em hipercubo utilizada pelo Hi-ADSD. Trata-se de um algoritmo distribuído que propaga as mensagens da aplicação através de uma

árvore geradora mínima, construída com base na estrutura hierárquica de *clusters* obtida pela função $C_{i,s}$. O Algoritmo 1 apresenta o pseudocódigo da solução. Ele está dividido em duas tarefas principais: a primeira é realizada pelo nodo inicial que deseja propagar uma mensagem e a segunda pelos demais nodos no hipercubo.

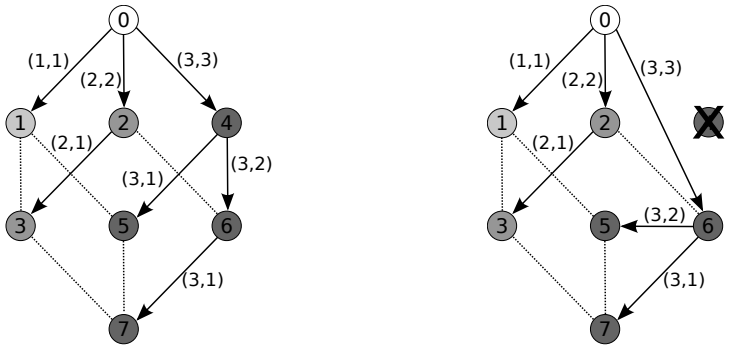
Algoritmo 1 STA(msg): Algoritmo de disseminação das mensagens

```
// Inicialização
 1: req_queue_i ← ∅ // fila de mensagens de requisição pendentes
 2: ∀j ∈ N : last_msg_i[j] ← ⊥ // última mensagem recebida de cada processo

|| ao receber uma nova requisição msg =REQUEST(i, last_i)
 3: req_queue_i ← req_queue ∪ {msg}

|| Tarefa 1: executada constantemente
 4: obtém próxima msg da fila req_queue_i
 5: repita
 6:     para s = 1 até log n faça
 7:         se ∃j ∈ C_{i,s} sem-falha então
 8:             enviar TREE(i, cluster = s, hops = s, msg) para n_j
 9:     aguardar ACK(j, m) ∀j ≠ i, n_j ∉ FD_i.suspects ou timeout
10: até que receba ACK(j, m) ∀j ≠ i, n_j ∉ FD_i.suspects

|| Tarefa 2: executada pelos demais nodos n_i
11: ao receber TREE(r, cluster, hops, msg) de algum n_j
12: se last_msg_i[j] ≠ msg então
13:     last_msg_i[j] ← msg
14:     entregar msg para n_i
15:     enviar ACK(i, msg) para n_r
16: enquanto (hops > 1) faça
17:     hops ← hops − 1
18:     se ∃k ∈ C_{i,s=hops} sem-falha então
19:         enviar TREE(r, cluster, hops, msg) para n_k
```

Considere inicialmente uma execução sem falhas. No primeiro passo, o nodo solicitante envia uma mensagem TREE para cada um dos seus *clusters* (linhas 6-8), totalizando $\log n$ mensagens. Cada mensagem carrega o identificador do solicitante, o *cluster* destino e a quantidade de saltos (*hops*) que a mensagem deverá percorrer. Como cada *cluster* possui uma quantidade de nodos progressivamente maior, quanto maior o seu identificador, maior a quantidade de saltos. Ao receber uma mensagem, o nodo verifica o total de saltos restantes e, se após decrementá-lo, $hops > 1$, encaminha a mensagem para os *clusters* internos subsequentes, iniciando pelos de maior valor. Para cada mensagem enviada, a quantidade de saltos é decrementada em uma unidade (linha 17).

Como exemplo, considere o hipercubo de três dimensões da Figura 3(a). O nodo n_0 inicia o processo enviando uma mensagem para cada um dos *clusters* $C_{0,1} = <1>$, $C_{0,2} = <2,3>$ e $C_{0,3} = <4,6,7,5>$. O nodo n_1 recebe a mensagem, verifica que $hops = 1$ e não a repassa. O nodo n_2 verifica que $hops = 2$, efetua o decremento e repassa a mensagem para n_3, que é o nodo do seu *cluster* $C_{2,1}$. Quando n_3 recebe a mensagem não faz o repasse, já que $hops = 1$. Por fim, o nodo n_4 recebe a mensagem com $hops = 3$ e a repassa para os seus *clusters* $C_{4,2} = <6,7>$ com $hops = 2$ e $C_{4,1} = <5>$ com $hops = 1$. A última mensagem é enviada de n_6 para n_7. Com isso, todos os nodos recebem uma única cópia da mensagem enviada inicialmente por n_0.

Para tratar os casos com falhas, antes de enviar um mensagem o nodo verifica



(a) sem-falhas (b) nodo 4 falho

Figura 3. Árvore geradora no hipercubo de 3 dimensões.

no vetor de estados se existe um nodo sem-falha no *cluster* destino (linhas 7 e 18). Se todos os nodos do *cluster* estão falhos, $hops$ é decrementado, mas a mensagem não é enviada, passando para a próxima tentativa no *cluster* subsequente. A Figura 3(b) ilustra um exemplo com falha. Como o nodo n_4 está falho, a mensagem inicial é enviada para o n_6 com $hops = 3$, que é o próximo no *cluster* $C_{0,3}$ de n_0. O valor de $hops$ é decrementado em n_6. No entanto, como n_4 pertence ao *cluster* $C_{6,2} = < 4, 5 >$ de n_6 e n_4 está falho, n_6 envia o par $(3, 2)$ para n_5. n_5 recebe a mensagem, decrementa $hops$ e verifica que não existe nodo sem-falha no seu *cluster* $C_{5,1}$ e, portanto, não repassa a mensagem. Na última etapa, $hops$ é decrementado em n_6 e o par $(3, 1)$ é enviado para n_7.

O parágrafo anterior discutiu a solução para o caso de falhas detectadas antes do início da disseminação. No entanto, é possível que uma falha ocorra durante o envio das mensagens. Considere, por exemplo, o caso da Figura 3(a). No primeiro passo, n_0 envie a mensagem para n_1, n_2 e n_4. Se n_4 falha após o envio por n_0, a mensagem não será repassada para n_5,n_6 e, consequentemente, para n_7. Esta perda de mensagem, além de reduzir a eficiência, pode levar o algoritmo de exclusão mútua a um estado de interblocagem (*deadlock*). Para garantir que todos os processos recebam a mensagem de requisição, uma mensagem ACK de confirmação é enviada diretamente para o mecanismo de disseminação do processo solicitante (linha 15). Assim, se após um intervalo de tempo determinado não forem recebidos os ACKs de todos os processos sem-falha (espera ativa da condição na linha 9), a disseminação é refeita. Para evitar mensagens de reconhecimento e resposta duplicadas, cada processo mantém localmente um vetor de informações com o *timestamp* da última mensagem recebida de cada processo que solicitou um recurso. Mensagens recebidas com *timestamp* diferente são reconhecidas e entregues à aplicação (condição da linha 12) e as repetidas são apenas repassadas na árvore.

4. Algoritmo de k-Exclusão Mútua Proposto

O algoritmo de k-exclusão mútua proposto neste trabalho é uma adaptação do algoritmo de [Raymond 1989a]. O objetivo das modificações é aumentar a eficiência na obtenção de recursos em cenários com falhas de processos. Assim como na versão original, o algoritmo faz uso apenas de mensagens de requisição (REQUEST) e resposta (REPLY) para solicitar e dar permissão, respectivamente. Para tratar os casos de falha, o solução em questão faz uso de um mecanismo auxiliar de monitoramento distribuído, que informa

o estado (falho ou sem-falha) de cada processo no sistema. O Algoritmo 2 ilustra o pseudocódigo do algoritmo proposto. As variáveis locais mantidas pelos processos são:

- $state_i$: armazena o estado atual do processo, que pode ser $not\_requesting$, $requesting$ ou $executing$;
- $clock_i$: usada como relógio lógico local. Inicialmente em zero, é atualizado sempre que uma mensagem de requisição é recebida, passando a armazenar o maior valor entre o relógio local e o *timestamp* da mensagem recebida;
- $last_i$: o valor de *timestamp* da última mensagem de requisição enviada;
- $perm\_count_i$: o total de permissões recebidas (mensagens de REPLY) desde a última tentativa de obtenção do recurso;
- $reply\_count_i[n]$: vetor que armazena a quantidade de respostas esperadas de cada processo;
- $defer\_count_i[n]$: contador que armazena a quantidade de respostas adiadas para cada processo. Após a liberação do recurso, o processo envia todas as permissões pendentes baseando-se nestas informações.

O algoritmo possui duas funções principais: uma função $Request()$ para iniciar o processo de requisição do recurso e uma função $Release()$ para quando deseja liberar o recurso obtido. O processo de requisição inicia com a mudança de estado do processo para $requesting$ (linha 7). Esta mudança garante que se p_i receber um pedido de permissão de um outro processo p_j, ela só será dada caso o relógio de p_j tenha um valor maior que o de p_i ou, em caso de empate, que o identificador de p_j seja menor (linha 21). Em seguida, uma mensagem marcada com o relógio local de p_i é enviada para os demais processos utilizando o algoritmo de *spanning tree* apresentado na Seção 3.

Originalmente, para obter acesso ao recurso é preciso que p_i obtenha permissão de $n - k$ processos. Na proposta em questão, considerando que processos podem falhar, foi adicionada uma espera ativa que consulta o algoritmo de monitoramento para verificar dinamicamente a quantidade de processos falhos (linha 12). Este valor é descontado do total de respostas esperadas, permitindo que o processo obtenha acesso aos recursos com mais eficiência. Ao receber a quantidade de permissões mínima, p_i obtém acesso a recurso e é colocado no estado $executing$. Quando uma falha é detectada durante o $requesting$, o algoritmo verifica se já recebeu a permissão do processo falho (linhas 29-30). Se já recebeu, $perm\_count$ é decrementada para não interferir na condição da linha 12, garantindo a propriedade de *safety* (embora o código do monitor não seja apresentado, é preciso garantir que a sua lista de suspeitos seja atualizada apenas após a execução deste ajuste). A liberação de um recurso por um processo p_i implica na mudança de estado do processo para $not\_requesting$ e no envio de todas as mensagens de requisição recebidas de outros processos que foram retidas e contabilizadas na estrutura $defer\_count$ (linhas 14-18). Com isso, os processos que estão aguardando permissões podem verificar a condição mínima da linha 12 e obter o recurso.

Para garantir o correto funcionamento do algoritmo, duas propriedades precisam ser satisfeitas. Em primeiro lugar, em cada instante de tempo, no máximo k processos diferentes podem estar utilizando os k recursos existentes, caracterizada pela propriedade de *safety* e, em segundo lugar, mas não menos importante, é preciso garantir que se um processo correto solicita um recurso, ele o obterá em um intervalo de tempo finito, de acordo com a propriedade de *liveness*. A prova formal será omitida devido a limitação de espaço, mas as duas propriedades são garantidas.

Algoritmo 2 Algoritmo de k-exclusão mútua robusta

// Inicialização

1: $state_i \leftarrow not\_requesting$
2: $\forall j \in N : reply\_count_i[j] \leftarrow 0$
3: $\forall j \in N : defer\_count_i[j] \leftarrow 0$
4: $perm\_count_i \leftarrow 0$
5: $clock_i \leftarrow 0$
6: $last_i \leftarrow 0$

Request(): n_i quer obter um recurso livre

7: $state_i \leftarrow requesting$
8: $last_i \leftarrow clock_i + 1$
9: $perm\_count_i \leftarrow 0$
10: STA(REQUEST(i, $last_i$))
11: $\forall j \neq i, j \notin FD_i.suspects : reply\_count_i[j] + +$
12: aguardar até que ($perm\_count_i \geq N - |FD_i.suspects| - k$)
13: $state_i \leftarrow executing$

Release(): n_i vai liberar o recurso

14: $state_i \leftarrow not\_requesting$
15: **para todo** ($j \neq i : j \notin FD_i.suspects$) **faça**
16: **se** ($defer\_count_i[j] \neq 0$) **então**
17: enviar REPLY($defer\_count_i[j]$) para n_j
18: $defer\_count_i[j] \leftarrow 0$

|| ao receber REQUEST(j, $last_j$) de n_j

19: $clock_i \leftarrow \max(clock_i,\ last_j)$
20: **se** ($n_j \notin FD_i.suspects$) **então**
21: **se** ($state_i = executing$ **ou** ($state_i = requesting$ **e** $(last_i,\ i) < (last_j,\ j)$)) **então**
22: $defer\_count_i[j] + +$
23: **senão**
24: enviar REPLY(1) para n_j

|| ao receber REPLY($count$) de n_j

25: **se** ($j \notin FD_i.suspects$) **então**
26: $reply\_count_i[j] \leftarrow reply\_count_i[j] - count$
27: **se** ($state_i = requesting$ **e** $reply\_count_i[j] = 0$) **então**
28: $perm\_count_i + +$

|| ao receber CRASH(j) de FD_i

29: **se** ($state_i = requesting$ **e** $reply\_count_i[j] = 0$) **então**
30: $perm\_count_i - -$

Em relação a propriedade de *safety*, considere que, em um dado instante de tempo, k processos estejam utilizando os k recursos disponíveis. Em seguida, um processo p_i inicia a operação de REQUEST, enviando mensagens para os demais processos. Cada processo p_j ao receber a solicitação pode tomar uma das seguintes decisões:

- Se p_j está no estado *not_requesting* ele responde imediatamente com um REPLY para p_i;
- Se p_j está no estado *executing* ele retém a resposta até que libere o recurso e, só então, envia REPLY para p_i;
- Se p_j está no estado *requesting* e $(last_i, i) < (last_j, j)$, ele envia imediatamente REPLY para p_i, pois neste caso p_i tem prioridade. Caso contrário, retém a resposta até que ele consiga obter o recurso e, só depois de liberá-lo, envia REPLY para p_i.

De acordo com a especificação do algoritmo, no máximo $n - k - 1$ processos irão tomar a primeira decisão e permitir imediatamente o uso do recurso, uma vez que k processos estão utilizando os recursos (e adiarão a resposta) e o processo solicitante não contribui para a decisão. Assim, p_i não terá acesso a um recurso e terá que aguardar por uma resposta adiada, que será enviada apenas quando um dos processos que estão utilizando recursos efetuar a liberação. Com isso, garante-se que, no máximo, k processos estarão utilizando os k recursos em cada instante de tempo.

A única exceção que comprometeria a propriedade de *liveness* seria o caso em que o processo solicitante ficasse bloqueado indefinidamente na linha 12 do algoritmo, aguardando pela quantidade mínima de respostas exigida. No entanto, se existe um recurso livre, em algum momento as respostas de autorização serão recebidas pelo processo solicitante, de acordo com as possíveis decisões já discutidas anteriormente. Se algum processo falhar durante a operação, em algum momento o algoritmo de monitoramento irá detectar a falha e atualizar a lista de suspeitos, permitindo que o solicitante não mais aguarde pela sua resposta. Se dois ou mais processos disputam um recurso, a propriedade em questão é garantida pela ordenação do relógio lógico. Logo, todo processo que solicitar um recurso o conseguirá em um tempo finito, de acordo com a ordem de solicitação ou, em caso de empate, com a prioridade do seu identificador.

5. Avaliação Experimental

O algoritmo de k-exclusão mútua proposto foi implementado no *framework* Neko [Urbán et al. 2002]. O Neko é uma ferramenta desenvolvida para a simulação e emulação de algoritmos distribuídos baseado em microprotocolos e troca de mensagens. Cada protocolo é instanciado em um processo (que pode representar um *nodo*) e utiliza uma rede real ou simulada para se comunicar com protocolos em outros processos. No contexto das simulações elaboradas, os algoritmos de exclusão mútua e o algoritmo de monitoramento são protocolos executados em cada processo. Uma camada intermediária foi inserida entre o o algoritmo de exclusão mútua e a rede para transportar as mensagens de REQUEST com base na *spanning tree*.

O algoritmo hierárquico de monitoramento (Hi-ADSD) foi implementado utilizando-se as classes de detecção disponíveis no pacote de tolerância a falhas do Neko. As falhas de processo foram geradas com o mecanismo de *crash* proposto em [Rodrigues e Jansh-Pôrto 2008]. A Figura 4 representa a arquitetura utilizada na construção do ambiente simulado. O algoritmo de k-exclusão mútua (Mutex) envia as mensagens *broadcast* de REQUEST que são interceptadas pela camada responsável pela propagação na árvore (Span). O algoritmo Hi-ADSD envia e recebe mensagens de ARE-YOU-ALIVE e I-AM-ALIVE diretamente na rede. Além disso, toda vez que o estado de um dos processos monitorados é modificado, o Mutex é notificado.

Para o estudo de caso, foram utilizados 16 processos e 5 recursos. A Figura 5 apresenta o hipercubo e a árvore correspondente para o caso em que todos os nodos estão sem-falha. O algoritmo de Raymond e a solução proposta foram configurados para enviar solicitações de recurso periodicamente. Após a obtenção do recurso, o processo aguarda um intervalo de 0,8 para liberá-lo. Em seguida, aguarda outro intervalo de 0,1 e executa uma nova solicitação. A rede do Neko utilizada foi a BasicNetwork, que é simulada e foi configurada com um tempo de transmissão constante de 0,01. O *timeout* para espera

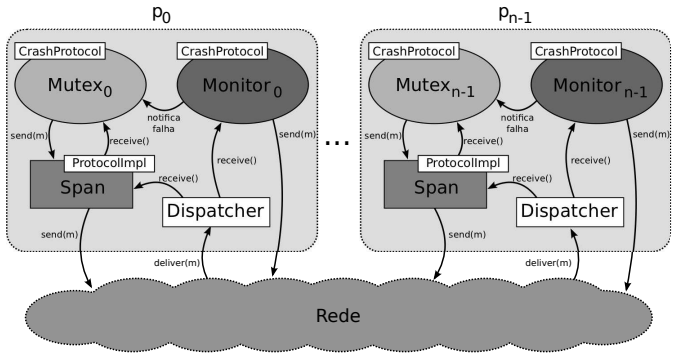
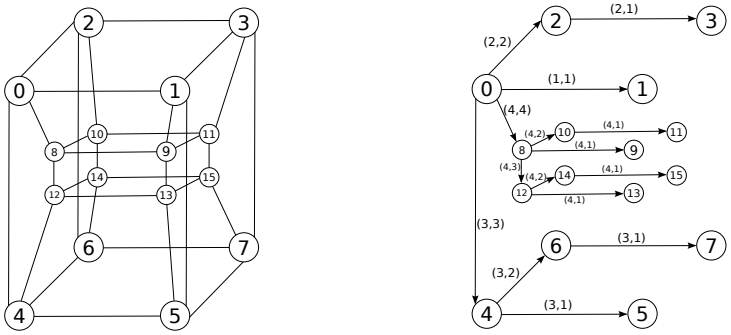


Figura 4. Organização dos módulos de simulação no Neko.



(a) ligações lógicas (b) árvore a partir de n_0

Figura 5. Hipercubo de 4 dim. e a árvore quando todos os nodos estão sem-falha.

dos ACKs foi estabelecido em $2 * \log n$ vezes o tempo de ida e volta, ou seja, 0,08. O Hi-ADSD também é executado periodicamente em intervalos de 1,0 com *timeout* de 0,04.

Falhas *crash* foram injetadas a cada 5,0 intervalos de tempo a partir do tempo 5,0. Em cada intervalo um processo falha, iniciando do processo 16 (p_{15}) até o processo 1 (p_1). Cinco processos (p_0 a p_4) solicitam os recursos periodicamente. As próximas seções apresentam os resultados de número de mensagens e eficiência para o cenário descrito acima, comparando o algoritmo de Raymond com o algoritmo proposto.

5.1. Número de Mensagens

No gráfico da Figura 6 é possível perceber nos intervalos iniciais que o número de mensagens enviadas pelo algoritmo proposto (já com os reconhecimentos do STA) é 50% maior que na solução de Raymond, sem contabilizar as mensagens de monitoramento. Em valores absolutos, o algoritmo proposto enviou 8884 mensagens (3.007 requisições, 2.919 respostas e 2.958 reconhecimentos) ao passo que o algoritmo de Raymond enviou 2.690 mensagens (1.425 req. e 1.265 resp.). Isto representa uma sobrecarga significativa em relação à solução de Raymond, sem considerar as 7.448 mensagens de diagnóstico (5.100 req. e 2.348 resp.). Entretanto, essa diferença deve-se ao fato de que o algoritmo de Raymond ficou bloqueado a partir da quinta falha, deixando de enviar mensagens de requisição após $t = 25,0$. Além disso, pode-se notar que na solução proposta o total de mensagens permanece equilibrado a medida que as falhas são injetadas.

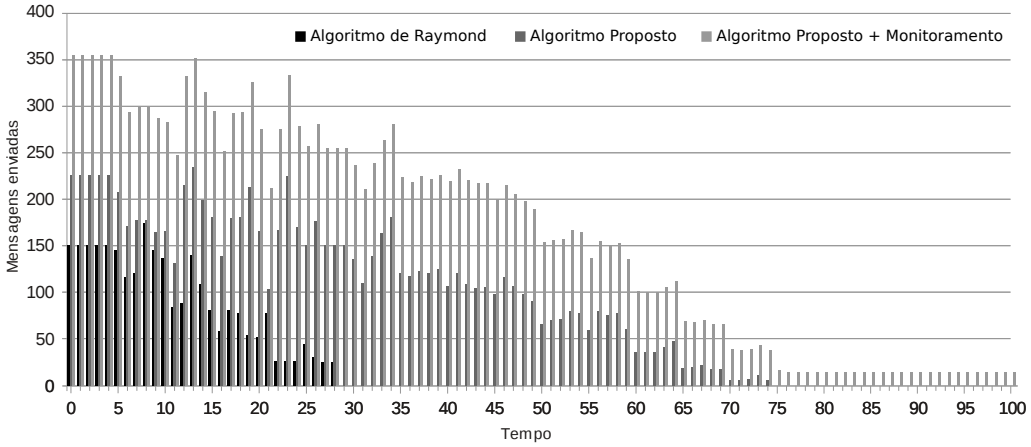


Figura 6. Comparativo de quantidade de mensagens enviadas.

5.2. Eficiência

A Figura 7 mostra a eficiência do algoritmo proposto. Como esperado, após $k - 1 = 4$ falhas o algoritmo de Raymond não consegue mais obter recursos. Isso ocorre após a injeção da quinta falha em $t = 25,0$. Já o algoritmo proposto, com base nas informações fornecidas pelo algoritmo de monitoramento, consegue melhorar significativamente a eficiência até a falha de $n - 1$ processos. Após a falha de $n - k$ processos em $t = 60,0$ ocorre uma degradação constante, que é justificada pela inicio da falha dos cinco processos solicitantes. Após o tempo 75.0 apenas o processo 0 (p_0) continua em execução.

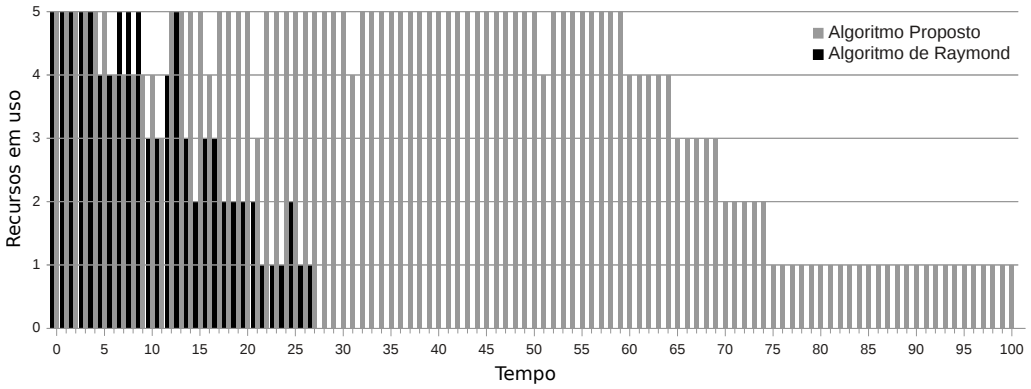


Figura 7. Comparativo de eficiência na obtenção dos recursos.

6. Trabalhos Relacionados

O trabalho de [Naimi 1996] avalia duas abordagens de exclusão mútua em hipercubos. A primeira utiliza pedido de permissão e requer $d2^d$ mensagens, sendo d, a dimensão do hipercubo . A segunda é baseada em passagem de permissão e requer $2d$ mensagens no pior caso. [Raymond 1989a] apresenta um algoritmo baseado em passagem de permissão que utiliza uma *spanning tree*. O número médio de mensagens é $O(\log n)$, mas pode chegar a $2(n - 1)$ no pior caso. Estas propostas, embora otimizadas em termos de quantidade de mensagens, não tratam a falha de processos.

O algoritmo de [Agrawal e El Abbadi 1989] faz uso do conceito de círculo social (*coterie*) introduzido por [Garcia-Molina e Barbara 1985] e de uma organização hierárquica em árvore para resolver a exclusão mútua em $\log n$, sendo n o número de processos. Outra proposta de [Agrawal e El Abbadi 1991] apresenta um algoritmo semelhante e tolerante a falhas. A dificuldade das soluções baseadas neste modelo é a complexidade de construir os círculos.

Mais recentemente, o trabalho de [Bouillaguet et al. 2008] utilizou a abordagem de permissão baseada na proposta de [Raymond 1989b] e que tolera até $n - 1$ processos falhos com o auxilio de um detector de falhas. Posteriormente, [Bouillaguet et al. 2009] propôs uma solução similar que dispensa o uso de detectores de falhas e de mensagens extras para detecção de nodos falhos. As informações de estado dos processos são integradas às mensagens do próprio algoritmo de exclusão mútua, que tolera até $k - 1$ falhas.

7. Conclusão

Este trabalho apresentou uma solução de k-exclusão mútua distribuída com suporte a falhas por $crash$ de até $n - 1$ processos. Um algoritmo distribuído e hierárquico de árvore geradora mínima foi proposto para propagar as mensagens de solicitação de permissão de forma escalável. Os dois algoritmos fazem uso de um mecanismo auxiliar de monitoramento do estado dos processos. O resultado dos experimentos mostrou que, embora as mensagens enviadas pelo monitoramento aumentem significativamente o total de mensagens final, tal custo justifica-se pelo aumento também significativo da eficiência na obtenção de recursos em cenários com falhas.

Como trabalhos futuros, pretende-se avaliar experimentalmente o impacto de falhas no algoritmo da árvore geradora mínima. Além disso, será feita uma comparação da solução proposta com o trabalho de [Bouillaguet et al. 2008].

Referências

Agrawal, D. e El Abbadi, A. (1989). Efficient solution to the distributed mutual exclusion problem. In *Proc. of the 8th ACM Symp. on Princ. Distr. Comp.*, pages 193–200.

Agrawal, D. e El Abbadi, A. (1991). An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 9:1–20.

Avresky, D. R. (1999). Embedding and reconfiguration of spanning trees in faulty hypercubes. *IEEE Transactions on Parallel and Distributed Systems*, 10(3):211–222.

Bertsekas, D. P., Ozveren, C., Stamoulis, G. D., Tseng, P. e Tsitsiklist, J. N. (1991). Optimal communication algorithms for hypercubes. *J. P. Distr. Com.*, 11:263–275.

Boehm, H.-J. e Adve, S. V. (2012). You don't know jack about shared variables or memory models. *Commun. ACM*, 55(2):48–54.

Bouillaguet, M., Arantes, L. e Sens, P. (2008). Fault tolerant k-mutual exclusion algorithm using failure detector. In *Int'l Symp. on Parallel and Distr. Comp.*, pages 343–350.

Bouillaguet, M., Arantes, L. e Sens, P. (2009). A timer-free fault tolerant k-mutual exclusion algorithm. In *4th Latin-American Symp. on Dep. Comp.*, pages 41–48.

Bulgannawar, S. e Vaidya, N. (1995). A distributed k-mutual exclusion algorithm. In *Proceedings of the 15th Int'l Conf. on Distr. Comp. Systems*, pages 153–160.

Delporte-Gallet, C., Fauconnier, H., Guerraoui, R. e Kouznetsov, P. (2005). Mutual exclusion in asynchronous systems with failure detectors. *J. P. Distr. Comp.*, 65:492–505.

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

Gallager, R. G., Humblet, P. A. e Spira, P. M. (1983). A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5:66–77.

Garcia-Molina, H. e Barbara, D. (1985). How to assign votes in a distributed system. *J. ACM*, 32:841–860.

Kruskal Jr., J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proc. of the American Mathematical Society*, pages 48–50.

Lamport, L. (1978). Time clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565.

Le Lann, G. (1977). Distributed systems - towards a formal approach. In *IFIP Congress*, pages 155–160.

Naimi, M. (1996). Distributed mutual exclusion on hypercubes. *SIGOPS Oper. Syst. Rev.*, 30:46–51.

Naimi, M., Trehel, M. e Arnold, A. (1996). A log (n) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34:1–13.

Prim, R. C. (1957). Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389–1401.

Raymond, K. (1989a). A distributed algorithm for multiple entries to a critical section. *Inf. Process. Lett.*, 30:189–193.

Raymond, K. (1989b). A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7:61–77.

Raynal, M. (1991). A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Oper. Syst. Rev.*, 25:47–50.

Raynal, M. e Beeson, D. (1986). *Algorithms for mutual exclusion*. MIT Press, Cambridge, MA, USA.

Ricart, G. e Agrawala, A. K. (1981). An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24:9–17.

Rodrigues, L. A. e Jansh-Pôrto, I. E. (2008). Ampliação do framework neko para a simulação de defeitos em algoritmos distribuídos. In *Proc. 7th I2TS*, pages 1–8.

Romano, P. e Rodrigues, L. (2009). An efficient weak mutual exclusion algorithm. In *8th Int'l Symp. on Parallel and Distributed Computing*, pages 205 –212.

Sanders, B. A. (1987). The information structure of distributed mutual exclusion algorithms. *ACM Trans. Comput. Syst.*, 5:284–299.

Suzuki, I. e Kasami, T. (1985). A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 3:344–349.

Urbán, P., Défago, X. e Schiper, A. (2002). Neko: A single environment to simulate and prototype distributed algorithms. *Journal of Inf. Science and Eng.*, 18(6):981–997.