Garantindo a Circulação e Unicidade do Token em Algoritmos com Nós Organizados em Anel Sujeitos a Falhas *

Luciana Arantes[1], Julien Sopena[1]

[1] LIP6 - Université de Paris 6 - INRIA Rocquencourt
Paris, France.

Luciana.Arantes@lip6.fr, Julien.Sopena@lip6.fr

Abstract. *We present in this paper an algorithm that provides some few functions which, when called by existing token ring-based distributed algorithms, easily render its token fault tolerant to crashes. It ensures thus the circulation and uniqueness of the token in presence of node crashes. At most k consecutive node crashes are tolerated. The lost of the token is avoided by keeping temporal backup copies of it in k other nodes. Our algorithm scales very well since a node monitors the liveness of at most k other nodes and neither a global election algorithm nor broadcast primitives are used to regenerate a new token. Its token regeneration mechanism is very effective in terms of latency cost. Furthermore, if the token keeps some information, the latter can be easier restored in case of failure of the token holder.*

Resumo. *Apresentamos neste artigo um algoritmo que oferece um conjunto pequeno de funções que, quando chamadas por algoritmos distribuídos do tipo token-ring, fazem com que o token utilizado por estes se torne tolerante a falhas. Ele garante assim a circulação e unicidade to token em presença de falhas por parada dos nós. O número máximo de falhas consecutivas toleradas é limitado a k. A perda do token é evitada mantendo-se cópias temporárias deste em k outros nós. O algoritmo é escalável pois um nó monitora no máximo k nós, não necessita de uma eleição de líder, que envolva todos os nós do sistema e nem a utilização de primitivas de difusão para recriar um novo token. Nossa solução para recriá-lo é bastante eficaz em termos de latência. Além disso, se o token armazena alguma informação, esta pode ser mais facilmente restaurada em caso de falha do nó que detém o token.*

1. Introdução

Inúmeros algoritmos distribuídos baseados em token-ring (exclusão mútua (*mutual exclusion*) [Lann 1977], detecção do término de uma aplicação distribuída (*termination detection*) [Misra 1983] [Dijkstra et al. 1986], eleição do líder (*leader election*) [Chang and Roberts 1979], [Franklin 1982], [Peterson 1982], algoritmos para ordenar a difusão de mensagens (*total order broadcast)* [Défago et al. 2004], gestão de filiação de grupo (*group membership*) [Amir et al. 1995], etc.) são baseados na unicidade de um token que circula ao longo de todos os nós do sistema, organizados logicamente em um anel. A todo momento existe no máximo um nó que possui o token (propriedade de segurança - *safety*) e o seu detentor sempre o envia ao seu nó sucessor no anel lógico, ou seja, o token circula entre todos os nós (propriedade de vivacidade - *liveness*).

*Mais valem $k + 1$ tokens voando do que um na mão.

Entretanto, em caso de falha de um nó, os mecanismos para detecção da perda do token e a sua regeneração podem ser caros e não muito eficazes, principalmente se considerarmos o aspecto escalabilidade, ou seja, o número de nós do anel (larga escala). Soluções existentes geralmente consistem em periodicamente verificar se a configuração do anel mudou (falha de um ou mais nós) através do monitoramento de todos os nós. Se houver alguma mudança, é necessário então executar um algoritmo de eleição global a fim de regerar um novo token e eleger o seu detentor (*token holder*). Em termos de escalabilidade, o ideal seria não envolver todos os nós do sistema para eleger um novo detentor do token (*token holder*) em caso de falha do anterior, não monitorar todos os nós do sistema para detectar falhas de nós e nem utilizar primitivas de difusão (*broadcast*). Por razões de desempenho, a solução deve também evitar reconstruir o anel lógico. Um terceiro ponto importante é que se o token armazenar alguma informação como em [Misra 1983] [Défago et al. 2004], esta deve ser facilmente restaurada quando o novo token é recriado.

Assim, com o objetivo de minimizar os problemas de falta de escalabilidade, desempenho e perda de informação do token acima relacionados, propomos um novo algoritmo que facilmente torna o token tolerante a falhas e que pode ser "plugado" em algoritmos token-ring existentes como os mencionados no início deste artigo. Para tanto, nosso algoritmo oferece as seguintes três funções: $SafeSendToken$, $SafeReceiveToken$ e $UpdateToken$. A perda do token, a criação de um novo token e a detecção de falhas dos nós se tornam assim transparentes à aplicação.

Basicamente, nossa solução evita a perda do token devido à falha de nós, criando cópias temporárias do token em outros nós que não aquele que detém o token. Sempre que o nó S_i, detentor do token, envia uma mensagem $\langle$*TOKEN*$\rangle$ ao seu sucessor direto S_{i+1} a fim de lhe passar o token, S_i também envia, de forma transparente para a aplicação, uma cópia desta mensagem aos k nós que sucedem S_{i+1} no anel. Ao receber uma mensagem $\langle$*TOKEN*$\rangle$, o nó em questão começa a monitorar um subconjunto de nós que receberam também a mesma cópia da mensagem. Um nó tem o direito exclusivo de utilizar o token ou quando ele recebe uma mensagem $\langle$*TOKEN*$\rangle$ que informa que ele é o próximo detentor do token, ou quando o mecanismo de monitoramento deste nó detecta que todos os nós que ele monitora falharam.

Nosso algoritmo tolera no máximo k falhas consecutivas em relação à ordem dos nós no anel. Os pontos críticos levantados anteriormente são evitados em nossa solução: ela é escalável pois um nó monitora no máximo k nós e o monitoramento inicia (resp. encerra) quando o token está (resp. não está) na vizinhança do nó; o token não é regerado com um algoritmo de eleição que envolva todos os nós; não é necessário reconstruir logicamente o anel; como k nós recebem um cópia do token, a informação que ele armazena pode ser mais facilmente recuperada se o nó detentor do token falhar.

Vale ressaltar que nossa estratégia para regerar o token é bastante eficaz em termos de latência quando comparada com outras em que a detecção da perda do token envolve todos os nós do anel e que utilizam uma eleição global ou protocolo de gestão de filiação de grupo para a escolha do novo detentor do token. Na nossa solução, o token é regerado instantaneamente graça às cópias backup do token. Quando a falha de um nó é detectada, o nó com o direito de recriar o token não precisa se comunicar com nenhum outro para fazê-lo. Uma segunda observação importante é que, conjuntamente com um protocolo que mantém a circulação virtual do token sobre um grafo (e.g. *depth-first token circulation protocol*), nosso algoritmo pode ser utilizado em qualquer grafo que repre-

sente uma dada topologia. Além disso, em caso de falhas (no máximo k consecutivas no anel lógico correspondente), o grafo não precisa ser reconstruído. Um último ponto é que nosso algoritmo pode suportar mais de k falhas, ou seja, tantas falhas enquanto estas não constituírem um conjunto de k falhas consecutivas.

Pode-se argumentar que nossa solução gera k mensagens adicionais para cada envio de uma mensagem $\langle \textit{TOKEN} \rangle$. Entretanto, ela conserva a mesma ordem de complexidade de mensagens $\mathcal{O}(N)$ que o algoritmo original. Além do que, nosso mecanismo de detecção de falhas apresenta um custo muito menor em termos de mensagens quando comparado com a maioria das soluções existentes em que cada nó monitora todos os outros, principalmente em sistemas com um número grande de nós.

O restante do artigo está organizado da seguinte maneira. A seção 2 especifica o modelo computacional. Nosso algoritmo, que oferece funções que permitem ao token se tornar resiliente às falhas, se encontra descrito na seção 3. Esta também inclui um esboço da prova de correção e exemplos de como algoritmos baseados em token circulando em anel podem facilmente utilizar as funções oferecidas pelo nosso algoritmo a fim de evitar a perda do token em caso de falhas. Uma comparação com trabalhos relacionados é feita em 4. Finalmente, a seção 5 conclui o trabalho.

2. Modelo de Sistema

Consideramos um sistema distribuído formado por um conjunto finito Π de $N > 1$ nós, $\Pi = \{S_0, S_2, \ldots, S_N - 1\}$ que se comunicam apenas por mensagens. Os canais são considerados confiáveis (*reliable*) e bidirecionais; eles não alteram, não criam, nem perdem mensagens. Entretanto, mensagens podem ser entregues fora da ordem de suas respectivas emissões. Há um processo por nó; os termos nós e processos são análogos neste artigo.

Os N nós são organizados em um anel lógico. Todo nó S_i é identificado de maneira única e conhece a identificação. S_i se comunica com seus respectivos $k + 1$ sucessores e predecessores mais próximos. Para evitar complicar a notação, nós denominamos que o sucessor de S_i é S_{i+1} e não $S_{(i+1)\%N}$.

Inicialmente, um certo nó possui o token. Este circula numa dada direção. Denominamos S_t o nó corrente detentor do token e este sempre o libera ao seu sucessor direto dentro de um limite de tempo.

Um processo pode falhar por parada (*crash*). Um processo *correto* é um processo que não falha durante a execução do sistema; senão ele é considerado *falho*. Seja k ($k < N - 1$), valor conhecido por todos os processos, o número máximo de falhas consecutivas toleradas no anel.

O sistema é síncrono, o que significa que a velocidade relativa dos processadores e os atrasos nas entregas das mensagens pelos canais de comunicação atendem a limites estabelecidos e conhecidos. Esta hipótese é necessária para garantir a unicidade to token, ou seja, um processo não pode ser suspeitado erroneamente de se encontrar falho.

3. Algoritmo em Anel Tolerante a Falhas do Token

Apresentamos nesta seção nosso algoritmo que torna o token, que circula por processos organizados logicamente em um anel, tolerante a falhas. Ele oferece três funções

à aplicação: $SafeSendToken$, $SafeReceiveToken$ e $UpdateToken$. A aplicação deve então substituir as suas funções originais utilizadas para enviar o token ao nó sucessor no anel e receber o token do predecessor pelas funções $SafeSendToken$ e $SafeReceiveToken$ respectivamente. A função $UpdateToken$ pode ser utilizada pela aplicação quando, em caso de falha, as informações guardadas pelo token precisarem ser atualizadas (veja seção 3.4). Consideramos que a aplicação se comporta corretamente, ou seja, um nó utiliza o token pour um intervalo de tempo limitado e depois o envia ao seu sucessor direto no anel. Consideramos que inicialmente a aplicação sempre atribui o token a S_0.

Tanto a versão original do algoritmo da aplicação quanto a sua versão que utiliza as funções oferecidas por nosso algoritmo para tolerar falhas do token precisam assegurar as seguintes propriedades:

- **segurança** (*safety*): A todo instante, existe, no máximo, um token no sistema.
- **vivacidade** (*liveness*): A todo instante, todo processo correto receberá o token num intervalo de tempo limitado.

Note que, momentaneamente, pode acontecer que não exista nenhum token no sistema pois ele está sendo regerado. No caso do nosso algoritmo, as cópias do token não são consideradas como token enquanto não substituírem o verdadeiro.

3.1. Variáveis e Mensagens

O nó S_i possui as variáveis locais $count_{S_i}$ e $token_{S_i}$. Aquela é utilizada para evitar que um nó considere uma mensagem $\langle$*TOKEN*$\rangle$ antiga como válida enquanto que esta controla se S_i detém o token, uma cópia deste, ou nenhum dos dois. Os valores $REAL$, $BACKUP$ e $NONE$ podem ser atribuídos à variável $token_{S_i}$: (1) $token_{S_i}$ possui o valor $REAL$ sempre que S_i tiver o direito de utilizar o token, ou seja, o processo da aplicação que executa em S_i pode usá-lo. Num dado instante t, apenas um processo tem sua variável $token$ igual a $REAL$, o que assegura a unicidade do token; (2) o valor $BACKUP$ é atribuído a $token_{S_i}$ se S_i é um dos k sucessores diretos do nó S_t e detém uma cópia válida do token. Se S_i não falhar, haverá um momento em que o valor de $token_{S_i}$ será substituído por $REAL$; (3) O valor $NONE$ é atribuído à variável $token_{S_i}$ quando S_i não possui o token nem uma cópia dele.

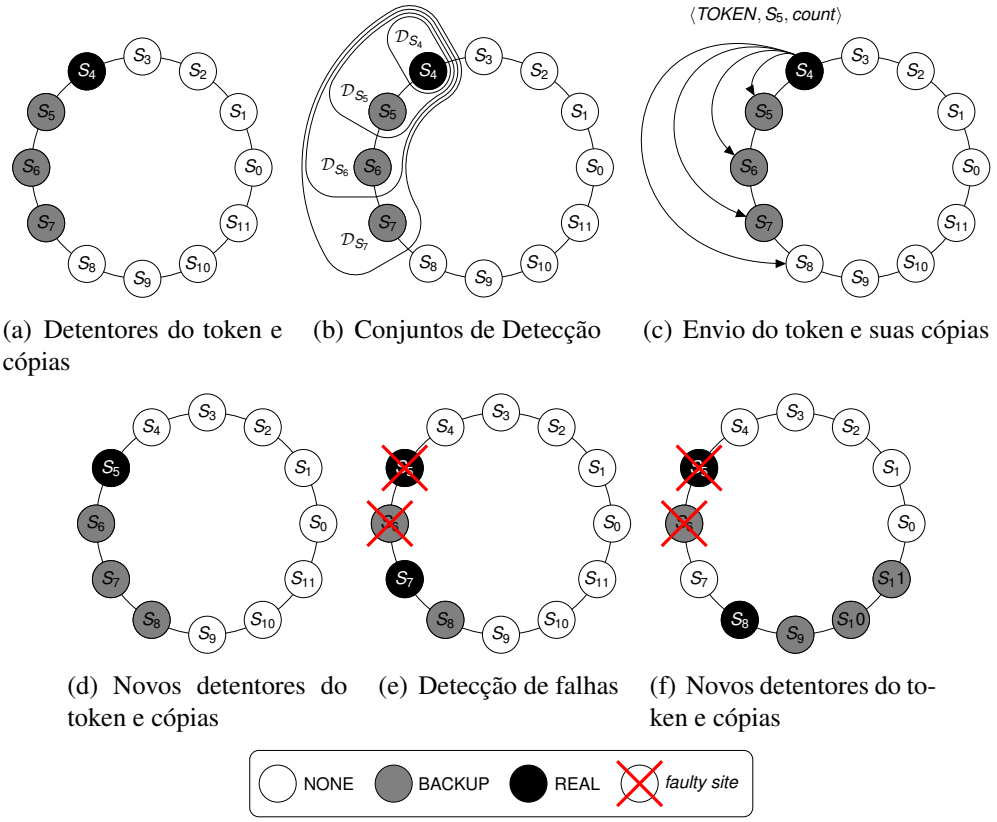
Os seguintes dois conjuntos são utilizados por S_i:

- $\mathcal{D}_{S_i}$ (Conjunto de detecção, *Detection set*): Conjunto que inclui os nós que S_i precisa monitorar para detectar as respectivas falhas além do próprio S_i. Ele é composto por $\{S_t \ldots S_i\}$, ou seja, o conjunto de nós entre S_t e S_i na ordem crescente do anel, incluindo ambos os nós. Ele possui então no máximo $k+1$ nós.
- $\mathcal{F}_{S_i}$ (Conjunto de nós falhos, *Faulty set*): Conjunto de nós que S_i detectou como falhos.

Se o valor de $token_{S_i}$ for igual a $REAL$ ou $BACKUP$, então $\mathcal{D}_{S_i} \neq \emptyset$. Os $k+1$ conjuntos de detecção são construídos de forma que cada um difere do outro e um inclui o outro (*nested detection sets*). O nó que detém o token está presente em todos os $k+1$ conjuntos. A vantagem de tal construção é o baixo custo em termos de mensagens quando comparado a um sistema de monitoramento no qual cada um dos $k+1$ nós monitora os outros k. Além disso, em caso de falha, a eleição do novo detentor do token não requer nenhum envio de mensagem.

A Figura 1 ilustra $N = 12$ nós organizados em anel e $k = 3$. Na Figura 1(a), o nó S_4 é o detentor do token e os nós $S_5 \ldots S_7$ possuem cópias do token enquanto que na Figura 1(b), podemos observar o conjunto de detecção dos nós $S_4 \ldots S_7$.

A mensagem $\langle TOKEN \rangle$ contém a identificação do sucessor do nó emissor da mensagem além do valor corrente da variável $count$ deste nó. O token pode conter outros dados relacionados ao algoritmo que utiliza nossas funções.



(a) Detentores do token e cópias (b) Conjuntos de Detecção (c) Envio do token e suas cópias

(d) Novos detentores do token e cópias (e) Detecção de falhas (f) Novos detentores do token e cópias

Figura 1. Exemplo execução algoritmo tolerante a falhas do token com k=3

3.2. Algoritmo

A Figura 2 descreve o pseudo-código para o nó S_i do nosso algoritmo. Por razões de simplificação, o monitoramento e a detecção da falha dos nós que pertencem a $\mathcal{D}_{S_i}$ não foram incluídos na figura. Sempre que $\mathcal{D}_{S_i}$ é atualizado, a função $updateDectection(\mathcal{D}_{S_i})$ desativa o monitoramento dos processos do antigo conjunto $\mathcal{D}_{S_i}$ e ativa o monitoramento dos processos contidos no novo $\mathcal{D}_{S_i}$. Quando a falha de S_j é detectada, S_i recebe o evento *Suspected* (função $ReceiveSuspected$). Como o sistema é síncrono não há nunca falsas suspeitas de falhas. Uma segunda observação é que S_i pertence ao seu próprio conjunto $\mathcal{D}_{S_i}$ não para se auto-monitorar mas apenas para decidir mais facilmente que ele detém o $REAL$ token quando todos os nós que ele monitora falharem. Um terceiro ponto a ressaltar é que em termos de implementação, não é necessário que S_i monitore todos os de seu $\mathcal{D}_{S_i}$ ao mesmo tempo. Ele pode monitorar apenas o predecessor não falho mais perto dele no anel. Se este também falhar, ele então passa a monitorar o predecessor deste nó, que pertence ao seu conjunto $\mathcal{D}_{S_i}$ e assim sucessivamente.

Na fase de inicialização (linhas 3-15), consideramos que S_0 detém o token. Consequentemente, os nós S_1 a S_k possuem uma cópia do token (token = $BACKUP$) e seus respectivos $\mathcal{D}$s são inicializados a $\{S_0, \ldots S_i\}$. Cada um destes nós começa então, com exceção dele mesmo, a monitorar os processos contidos no seu $\mathcal{D}$ (linha 15). Todos os outros nós do anel não possuem nenhuma cópia válida do token ($token = NONE$).

A função $SafeSendToken(< TOKEN >)$ permite a $S_i = S_t$ enviar o token a seu sucessor S_{i+1} e uma cópia deste a $\{S_{i+2} \ldots S_{t+k+1}\}$, indicando que o próximo proprietário do $REAL$ token é o nó S_{i+1}, i.e. $S_{i+1} = S_t$ (Figuras 1(c) e 1(d)). Como S_i não possui mais o token ($token = NONE$), ele não mais monitora os processos de seu $\mathcal{D}_{S_i}$ (linha 21). Vale lembrar que consideramos que S_i chama a função $SafeSendToken(< TOKEN >)$ somente se ele detiver o $REAL$ token e ao fazê-lo ele não pode mais utilizá-lo até adquiri-lo novamente.

Ao receber uma mensagem $TOKEN$ de S_j (função $SafeReceiveToken(< TOKEN, S_t, count_r >)$) que não seja antiga, S_i atualiza sua variável $count_{S_i}$ com o valor ($count_r$) contido na mensagem $TOKEN$ (linha 24). Ele então atribui ao seu $\mathcal{D}_{S_i}$ os nós entre S_t e ele próprio, incluindo ambos os nós (linha 25). Assim, (1) se S_i é o próximo detentor do token, ele atribui o valor $REAL$ à sua variável $token_{S_i}$ e entrega o token à aplicação (linhas 27-28); (2) Se S_i detecta a falha de todos os nós que ele monitora, ele se torna o detentor do $REAL$ token executando para tanto a função $UseBackup()$ (linhas 39-42; Figura 1(e)). Ele entrega da mesma forma que em (1) o token à aplicação; (3) senão S_i afeta o valor $BACKUP$ à sua variável $token_{S_i}$ (linha 32). Nos três casos, ele passa a monitorar os nós do seu novo $\mathcal{D}_{S_i}$ (linha 33).

Quando S_i é informado ($ReceiveSuspected(S_j)$, linha 34) da falha de S_j, um dos nós monitorado por S_i, este atualiza o conteúdo de seu conjunto de nós falhos $\mathcal{F}_{S_i}$. Se S_i detectar a falha de todos os nós que ele monitora, S_i chama a função $UseBackup()$, a fim de se tornar o novo detentor do $REAL$ token.

A função $UseBackup()$ adiciona a $count_{S_i}$ o número de nós que S_i detectou terem falhados (linha 39), o que garante a coerência de $count_{S_i}$ e que mensagens antigas que possam chegar mais tarde a S_i sejam descartadas. Ela também altera o valor da variável $token_{S_i}$ de $BACKUP$ para $REAL$ (linha 40). Antes de entregar o token à aplicação, S_i pode atualizar a informação armazenada no token (linha 41), se necessário. Por exemplo, o algoritmo de detecção de término da aplicação de Misra [Misra 1983] guarda um contador no token e este precisa ser atualizado antes que o token seja entregue à aplicação se o novo detentor do token não o recebeu de seu nó predecessor. Para mais detalhes veja a seção 3.4. Vale ressaltar que a criação de um novo token é bastante eficaz pois basicamente consiste em considerar uma das cópias $BACKUP$ do token como o $REAL$ token.

3.3. Esboço da Prova de Correção

Alguns dos principais argumentos da prova de que o algoritmo da Figura 2 satisfaz as propriedades de segurança (*safety*) e vivacidade (*liveness*) são apresentados nesta subseção.

Definições: Consideramos que o tempo é discretizado pela chamada às funções $SafeSendToken$, $SafeReceiveToken$ e $UseBackup$. $t = 0$ é estabelecido pela chamada da função $Initialisation$. $\mathcal{C}$ denota o tempo discretizado enquanto que $\mathcal{C}_d$, o tempo discretizado referente às chamadas das funções $UseBackup$. Processos não têm

```
1  /* 𝒟: Detection set */
2  /* ℱ: Faulty set    */
3  Initialisation ()
4      count ← 0
5      ℱ ← { }
6      case i = 0
7          token ← REAL
8          𝒟 ← {S_0}
9      case 0 < i ≤ k
10         token ← BACKUP
11         𝒟 ← {S_0, .., S_i}
12     case k < i
13         token ← NONE
14         𝒟 ← { }
15     UpdateDetection(𝒟)

16 SafeSendToken (⟨TOKEN⟩)
   to S_{i+1}
17     count ← count + 1
18     Send ⟨TOKEN,S_{i+1},count ⟩
       to   {S_{i+1}, .., S_{i+k+1}}
19     token ← NONE
20     𝒟 ← { }
21     UpdateDetection(𝒟)

22 SafeReceiveToken (⟨TOKEN,S_t,count_r⟩) from S_j
23     if count < count_r then
24         count ← count_r
25         𝒟 ← {S_t, .., S_i}
26         if S_i = S_t then
27             token ← REAL
28             DeliverToken(⟨TOKEN⟩)
29         else if 𝒟/ℱ = {S_i} then
30             UseBackup()
31         else
32             token ← BACKUP
33         updateDetection(𝒟)

34 ReceiveSuspected (S_j)
35     ℱ ← ℱ ∪ S_j
36     if 𝒟/ℱ = {S_i} then
37         UseBackup()

38 UseBackup ()
39     count ← count + ( #(𝒟) - 1 )
40     token ← REAL
41     UpdateToken(⟨TOKEN⟩)
42     DeliverToken(⟨TOKEN⟩)
```

Figura 2. Algoritmo tolerante à perda do token

acesso a $\mathcal{C}$ e nem a $\mathcal{C}_d$. Eles são introduzidos apenas por uma questão de conveniência da apresentação da prova.

Para expressar o valor de certas variáveis de nosso algoritmo em relação a $\mathcal{C}$, definimos algumas funções. Para um dado $t \in \mathcal{C}$ e um dado site S, cada uma destas funções retorna o valor da respectiva variável.

Denotamos $\mathcal{P}(\Pi)$ o conjunto de potência de π. As funções que respectivamente retornam o valor das variáveis *token*, $count$ e $\mathcal{D}$ para o nó S a t são:

$Token(S,t):\ \pi \times \mathcal{C} \rightarrow \{NONE, BACKUP, REAL\}$
$Count(S,t):\ \pi \times \mathcal{C} \rightarrow \mathbb{N}$
$\mathcal{D}(S,t):\ \pi \times \mathcal{C} \rightarrow \mathcal{P}(\Pi)$

$\langle Pend_{REAL} \rangle$ e $\langle Pend_{BACKUP} \rangle$ denotam uma mensagem $\langle$*TOKEN,*S_i*,count*$\rangle$ pendente que é endereçada respectivamente a S_i e um dos k sucessores de S_i.

Denotamos S_d o nó que chama a função $UseBackup$.

Para auxiliar a prova de nosso algoritmo, introduzimos as seguintes propriedades:

- $\boldsymbol{PSafetyCond(t)}$: Existe no máximo um processo correto que detém o $REAL$ token ou que é o receptor de $\langle Pend_{REAL} \rangle$.
  Se $PSafetyCond(t)$ é satisfeita, nós denotamos:
  - $Holder(t)$: o nó que satisfaz $PSafetyCond(t)$.
  - $HCount(t)$: equivale a $Count(Holder(t), t)$, se $Token(Holder(t), t) = REAL$ ou ao valor de *count* de $\langle Pend_{REAL} \rangle$, caso contrário.
  - $\mathcal{T}(t)$: o conjunto token a t, ou seja, o conjunto ordenado de $k+1$ nós composto de $Holder(t)$ e seus k sucessores.
- $\boldsymbol{PHolderCount(t)}$: O detentor do token possui o maior valor de *count* entre todos os nós não falhos e mensagens $\langle$*TOKEN*$\rangle$ pendentes.
- $\boldsymbol{PHolderMonitored(t)}$: Se um nó não falho monitora outros nós, então $Holder(t)$ está presente entre estes nós.
- $\boldsymbol{PNestedMonitoring(t)}$: Se dois nós não falhos monitoram outros nós, ao menos um deles monitora o outro.

Definimos que $S_i \prec_t S_j$, se S_i precede S_j em $\mathcal{T}(t)$.

Assumimos que o algoritmo que utiliza as funções oferecidas por nosso algoritmo as chamam corretamente e que o algoritmo original (sem as chamadas às referidas funções) satisfazem as propriedades de segurança e vivacidade.

Hip 1 (Hipótese de Uso Correto) *Um nó pode chamar a função SafeSendToken à condição que ele possua o $REAL$ token. Depois de chamá-la ele não detém mais o token.*

Lema 1 *A $t = 0$, todas as propriedades acima descritas são satisfeitas.*

Prova. A função *Initialisation* é chamada a $t = 0$. $PSafetyCond(0)$: S_0 é o único detentor do token e não há nenhuma mensagem $\langle$*TOKEN*$\rangle$ pendente; $PHolderCount(0)$: o valor da variável *count* de todos os nós é igual a 0; $PHolderMonitored(0)$ e $PNestedMonitoring(0)$: os nós com um conjunto de detecção não vazio, $S_0 \ldots S_k$, são sucessivos no anel e monitoram nós entre S_0, o detentor do token, e ele próprio.

Lema 2 $\forall t \in \mathcal{C}, PSafetyCond(t) \wedge PHolderCount(t) \implies PHolderCount(t+1)$.

Prova.

- Se $t+1 \notin \mathcal{C}_d$: $PSafetyCond(t)$ assegura que existe no máximo um nó S_i com $Token(S_i, t) = REAL$ enquanto $PHolderCount(t)$, que S_i possui o maior valor de $count$. Além disso, Hyp.1 garante que S_i é o único que pode executar $SaveSendToken$ e incrementar *count* (line 17) a $t+1$. Consequentemente, quando S_i enviar a mensagem $\langle$*TOKEN*$\rangle$ a $t+1$, esta contém o maior valor possível de *count* e o novo detentor do token atribuirá este valor à sua própria variável *count* quando da recepção desta mensagem (linha 24).
- Se $t+1 \in \mathcal{C}_d$: Quando o nó S_d chama a função $UseBackup$, $\#(\mathcal{D}(S_d, t+1)) - 1$ é adicionado à sua variável *count* (linha 39). Como S_d monitorava $Holder(t)$ a t:

$$Count(Holder(t), t) - Count(S_d, t+1) < \#(\mathcal{D}(S_i, t)) - 1 \\ \implies Count(Holder(t), t) < Count(S_d, t+1).$$

  Logo, como $PHolderCount(t)$ assegura que $Holder(t)$ possui o maior valor de *count* a t, S_d detém o maior valor de *count* a $t+1$.

Lema 3 $\forall t \in \mathcal{C}, PSafetyCond(t) \wedge PHolderMonitored(t) \implies PNestedMonitoring(t)$.

Prova. Seja S_i e S_j dois nós cujos respectivos conjuntos de detectores são não vazio. Como por hipótese $PHolderMonitored(t)$ é verdadeiro e estes conjuntos de detectores são compostos de nós sucessivos (linha 25), então $\{Holder(t), .., S_i\} \in \mathcal{D}(S_i, t)$ e $\{Holder(t), .., S_j\} \in \mathcal{D}(S_j, t)$. Consequentemente, se $S_i \prec_t S_j$ (resp. $S_j \prec_t S_i$) em $\mathcal{T}(t)$, então $\{Holder(t), .., S_i\} \in \mathcal{D}(S_j, t)$ (resp. $\{Holder(t), .., S_j\} \in \mathcal{D}(S_i, t)$).

Lema 4 $\forall t \in \mathcal{C}, PSafetyCond(t) \wedge PHolderCount(t) \wedge PHolderMonitored(t) \wedge PNestedMonitoring(t) \implies PHolderMonitored(t+1)$.

Prova.

- Se $t+1 \notin \mathcal{C}_d$: Prova por contradição. Suponhamos que $PSafetyCond(t)$ e $PHolderCount(t)$ são verdadeiros, mas não $PHolderMonitored(t+1)$, ou seja, existe um nó S_j com $\mathcal{D}$ não vazio que não monitora $Holder(t+1)$. $PSafetyCond(t)$ e Hyp.1 asseguram que apenas S_i, o detentor do token a t, pode chamar a função $SafeSendToken$ a $t+1$ para enviar uma nova mensagem $\langle$*TOKEN*$\rangle$ a seus $k+1$ sucessores. Além disso, $PHolderMonitored(t)$ garante que S_j monitora o detentor do token a t. Assim, por construção (linha 25), se $S_i \in \mathcal{D}(S_j, t)$, S_j monitora todos os nós entre S_i e S_j no anel. Chegamos então a uma contradição pois o $Holder(t+1) = S_{i+1}$ é monitorado por S_j.
- Se $t+1 \in \mathcal{C}_d$: S_d não monitora nenhum nó correto pois todos os nós que pertencem a $\mathcal{D}(S_d, t)$ são falhos. Além disso, $PNestedMonitoring(t+1)$ garante que dois nós não falhos com $\mathcal{D} \neq \emptyset$, ao menos um deles monitora o outro. Consequentemente, S_d pertence a todo conjunto de detecção não vazio. Como $Holder(t+1) = S_d$, $PHolderMonitored(t+1)$ é verdadeiro.

Lema 5 $\forall t \in \mathcal{C}, PSafetyCond(t) \wedge PHolderCount(t) \wedge PNestedMonitoring(t) \wedge PHolderMonitored(t) \implies PSafetyCond(t+1)$.

Prova.

1. Se $t+1 \notin \mathcal{C}_d$:
   $PSafetyCond$ é satisfeita em t. Nós distinguimos então os dois casos seguintes:
   (a) existe um nó, S_i com um ***REAL*** token ($Token(S_i, t) = REAL$) e não há nenhuma mensagem $\langle$***TOKEN***$\rangle$ pendente. Hyp. 1 assegura que S_i é o único nó que pode chamar a função ***SafeSendToken*** e, consequentemente, enviar uma mensagem $\langle$***TOKEN***$\rangle$ a $t+1$. Como ***SafeSendToken*** garante que $Token(S_i, t_i + 1) = NONE$ (linha 19), $PSafetyCond$ é verdadeira a $t+1$.
   (b) não existe um nó com um ***REAL*** token mas há uma mensagem $\langle Pend_{REAL}\rangle$ pendente, endereçada a S_i, no sistema. Hyp. 1 garante que nenhum site pode mandar uma nova mensagem $\langle$***TOKEN***$\rangle$ a $t+1$, ou seja, a mensagem em questão é a única $\langle Pend_{REAL}\rangle$ do sistema. Consequentemente, $PSafetyCond$ é verdadeira a $t+1$: se S_i receber $\langle Pend_{REAL}\rangle$ a $t+1$, ele será o novo detentor do token; senão ele é o único receptor desta mensagem.
2. Se $t+1 \in \mathcal{C}_d$:
   Como $PHolderMonitored$ é verdadeira a t, $Holder(t)$ pertence a $\mathcal{D}(S_d, t)$. Distinguimos então os seguintes dois casos:
   - $Holder(t)$ é um nó de $\mathcal{D}(S_d, t)$ diferente de S_d. Como S_d regera um novo token apenas quando todos os nós que ele monitora, com exceção dele mesmo, falharem (linhas 29 e 36), não há nenhum outro nó correto com um $REAL$ token a $t+1$, ou seja, Holder(t) se encontra falho a $t+1$.
   - S_d é o $Holder(t)$. Como S_d não detém o $REAL$ token, existe uma $\langle Pend_{REAL}\rangle$ que lhe é endereçada. Neste caso, $PHolderCount(t+1)$ garante que o valor de *count* contido nesta mensagem não pode ser maior que o valor corrente *count* de S_d. O teste da linha 23 irá então ignorar esta mensagem.

   Em ambos os casos $PSafetyCond(t+1)$ é verdadeira.

Teorema 1 *Para o máximo de k falhas consecutivas, nosso algoritmo assegura a propriedade de segurança.*

Prova. A demonstração é consequência direta dos lemas anteriores

Teorema 2 *Para o máximo de k falhas consecutivas, nosso algoritmo assegura a propriedade de vivacidade.*

Prova. A fim de demonstrar a propriedade de vivacidade, basta provar que (1) se o token nunca se perder, todo site que detém o ***REAL*** token o enviará ao seu sucessor e (2) se o token é perdido devido à falha do detentor do ***REAL*** token, este será regerado pour um dos k nós sucessores do detentor que falhou.

A prova de (1) é trivial pois quando S_i envia $k+1$ cópias da mensagem $\langle$***TOKEN***$\rangle$ (linha 18) a t, ele informa nesta mensagem que o próximo detentor do token é o nó S_{i+1} ($Holder(t+1) = S_{i+1}$). Como os canais são confiáveis e S_i possui o maior valor de *count* a t ($PSafetyCond(t)$ e $PHolderCount(t)$), S_{i+1} terá o $REAL$ a $t+1$.

Para provar (2), consideremos que S_i é o último nó a possuir o token que enviou as $k+1$ cópias da mensagem $\langle$***TOKEN***$\rangle$ a t e que f é o número de sucessores falhos de

Si a $t+1$. Por hipótese, entre estes $k+1$ nós, há pelo menos um correto. Como $f \leq k$, o nó S_i enviou uma cópia da mensagem a $S_{i+f+1} \in \{S_{i+1}, .., S_{i+k+1}\}$. Além disso, como $PSafetyCond(t)$ e $PHolderCount(t)$ são verdadeiras pelas mesmas razões descritas em (1), S_{i+f+1} receberá a mensagem $\langle$*TOKEN*$\rangle$ enviada por S_i. A linha 25 atribuirá então $\{S_{i+1}, .., S_{i+f+1}\}$ a $\mathcal{D}$ de S_{i+f+1}. As falhas dos nós $\{S_{i+1}, .., S_{i+f}\}$ serão a termo detectadas, o que resultará na chamada da função *UseBackup* (linhas 30 e 37) que criará um novo *REAL* token.

3.4. Exemplos do Uso de Nosso Algoritmo

Discutimos nesta sub-seção sobre como alguns algoritmos existentes na literatura baseados na circulação do token e sua unicidade podem chamar as funções oferecidas por nosso algoritmo para que continuem corretos em presença de falhas dos nós. Consideramos que o sistema sobre o qual esses algoritmos executam são síncronos.

Nosso algoritmo se adapta naturalmente ao algoritmo proposto por Le Lann [Lann 1977] em que o acesso exclusivo ao recurso compartilhado é condicionado à possessão de um token, único, que circula entre todos os nós do sistema organizados em anel. Quando um processo recebe o token, se ele precisar acessar um recurso compartilhado, ele o faz. Senão, simplesmente repassa o token ao seu sucessor. Ao terminar o acesso ao recurso compartilhado, ele também envia o token ao seu sucessor. Para garantir a correta execução do algoritmo em presença de falhas, cada processo precisa simplesmente chamar as funções $SafeSendToken$ e $SafeReceiveToken$ oferecidas pelos nosso API de comunicação para respectivamente enviar e receber o token.

Alguns algoritmos de detecção do término de uma aplicação distribuída [Dijkstra et al. 1986] [Misra 1983] consideram os nós organizados logicamente em anel. O princípio destes algoritmos é verificar que todos os N nós se encontram passivos após uma volta completa do token no anel. No algoritmo de [Misra 1983], os nós possuem a cor branca (inicial) ou preta e um token circula entre os nós. A cor preta indica que o nó ficou ativo após a passagem do token. Este contém uma variável, *passif_count*, que contabiliza o número de nós que o token encontrou passivo (cor branca). Um nó inicia o algoritmo de detecção. Ao receber o token, se o nó receptor é de cor branca, ele reinicializa *passif_count* a 1, senão ele o incrementa. O término é detectado quando todos os nós são de cor branca após uma volta total do token ($passif\_count = N$). Este algoritmo pode utilizar as funções $SafeSendToken$ e $SafeReceiveToken$ para assegurar a circulação do token. Um nó falho pode ser visto como passivo. Assim, se o novo detentor do $REAL$ token não o recebeu de seu predecessor (falha de um ou mais nós), o valor do contador do token precisa ser atualizado antes de ser entregue ao algoritmo de término. A função $UpdateToken()$ teria então o seguinte código:

```
43 UpdateToken (⟨TOKEN⟩)
44 ⌊ ⟨TOKEN⟩.passif_count ← ⟨TOKEN⟩.passif_count + ( #(D) - 1 )
```

Note que consideramos que (1) o nó iniciador do algoritmo só pode falhar após ter enviado as $k+1$ cópias da mensagem $\langle$*TOKEN*$\rangle$ e que (2) no caso de falhas, o sistema de monitoramento só indica a ocorrência de uma ou mais falhas (função $ReceiveSuspected()$) após esperar um certo intervalo de tempo que assegure que não existe mensagens pendentes para o novo detentor do token quando este o entregar ao

algoritmo de término.

Vários autores [Chang and Roberts 1979], [Franklin 1982], [Peterson 1982], etc. propuseram algoritmos para o problema da eleição de um líder para nós interligados em uma estrutura lógica de anel. Os nós candidatos a se tornarem líder enviam uma mensagem de candidatura (token) ao seu sucessor que circula no anel segundo as regras de comparação e transmissão do algoritmo em questão. Por exemplo, no algoritmo de Chang e Roberts, a mensagem é retransmitida enquanto um melhor candidato não é encontrado. Um nó é eleito líder quando receber a mensagem que contém a sua própria candidatura.

Para assegurar a correta execução do algoritmo de Chang e Roberts na presença de falhas, a circulação e a unicidade das mensagens de candidaturas precisam ser garantidas. Entretanto, uma mensagem de candidatura pode ser vista, analogamente ao token, como um objeto único que circula no anel. Em outras palavras, podemos aplicar nosso algoritmo para garantir a tolerância a falha dos pedidos de candidatura: a mensagem $< TOKEN >$ representa então a candidatura de um nó cuja identificação S_l está contida na mensagem. Como para um anel com N nós, o número máximo de pedidos pendentes é N, nosso algoritmo precisa controlar a circulação e unicidade de no máximo N objetos. Observe que consideramos a mesma hipótese (2) de que não há mensagens pendentes para o novo detentor do objeto no caso de falhas de seus predecessores.

O nosso algoritmo assegura que em presença de no máximo $k + 1$ falhas consecutivas, o algoritmo de eleição termina (se e somente se ao menos um dos candidatos emitiu $k + 1$ cópias da mensagem de candidatura). Vale ressaltar que não podemos assegurar que o líder seja um processo correto. Entretanto, poderíamos oferecer uma função $UpdateToken$ para tanto. O nó S_j se torna líder ao receber sua própria mensagem de candidatura. Esta foi enviada com a utilização da função $SafeSendToken$ e, consequentemente, também recebida pelos k sucessores de S_j no anel, que monitoram então S_j. No caso de falha deste, a função $UpdateToken()$ é executada por S_i, o sucessor correto de S_j. Este sabe que se trata da falha de um líder pois o havia registrado como tal (*currentLeader*). Assim, para assegura a eleição de um nó correto basta alterar a candidatura de S_j pela de S_i. Ao receber a sua propria candidatura S_i se tornará então o novo líder. O código da função $UpdateToken()$ seria:

```
45 UpdateToken (⟨TOKEN⟩)
46     if currentLeader = S_l then
47         set S_i as S_l in ⟨TOKEN⟩
```

4. Trabalhos Relacionados

Vários algoritmos de exclusão mútua tolerante a falhas [Nishio et al. 1990] [Manivannan and Singhal 1994] [Chang et al. 1990], etc. existem na literatura. Porém, estes geralmente adotam soluções que não são escaláveis como por exemplo uma eleição global ou necessitam que o nó que detectou a perda do token receba um confirmação (*acknowledge*) de todos os outros nó antes de recriar o token como em [Nishio et al. 1990] [Manivannan and Singhal 1994].

Misra [Misra 1983] propõe em seu artigo sobre detecção do término de uma aplicação distribuída para topologias em anel adaptar o algoritmo para que a perda do

token possa ser detectada e un novo token então regerado. O autor argumenta que a perda do token é similar à detecção de término da aplicação se esta considerar apenas as mensagens ligadas ao token. Como nossa solução, o autor usa o conceito de ter mais do que um token: há dois token simétricos no sistema mas um deles é visto como o token backup. Comparando o número de sequência que um token possui com o armazenado pelos nós, um token pode detectar a perda do outro. Entretanto, a detecção é possível somente se aquele também não se perder na mesma volta que este (*round*).

Em [Mueller 2001], Mueller apresenta um mecanismo que oferece tolerância a falhas para protocolos de sincronização baseados em token. Um anel lógico é utilizado para detectar a falha de um nó e, se necessário, eleger um novo detentor do token. Porém, contrariamente à nossa solução, a detecção e tratamento das falhas não é transparente à aplicação. Esta precisa ser modificada para incluir um sistema de monitoramento de nós e este então, ao suspeitar uma falha, chama o mecanismo tolerante a falhas proposto pelo autor. Além disso, este mecanismo organiza os nós em um anel lógico que permite detectar a falha de apenas um único nó e cuja manutenção é extremamente cara, o que limita a escalabilidade da solução. Um terceiro ponto é que ao criar um novo token, o estado do token não é preservado ou restaurado, como na nossa solução.

Inúmeros algoritmos de difusão de mensagens utilizam um mecanismo baseado em token com nós organizados em anel para ordenar mensagens emitidas por difusão (*total order broadcast*). O token circula entre os nós ou um subconjunto de nós. Alguns algoritmos como [Chang and Maxemchuck 1984] e [Amir et al. 1995] toleram falhas de nós mas envolvem mecanismos caros e globais como uma fase de reconstrução em que a difusão não é permitida ou um protocolo de gestão de filiação de grupo para reconstruir o anel e eleger um novo detentor to token.

O trabalho de Ekwall et al. [Ekwall et al. 2004] é proximo ao nosso no sentido em que os nós do sistema são organizados em um anel lógico e o token é enviado a $f + 1$ nós sucessores, sendo f o número máximo de falhas. Porém, o objetivo dos autores é diferente do nosso. Eles consideram um sistema assíncrono sobre o qual querem construir um algoritmo de consenso baseado em token. Um nó espera receber o $token$ de seu predecessor. Porém, se aquele suspeitar que este falhou ele espera o token de qualquer um de seus f predecessores. Contrariamente à nossa solução, a detecção não é perfeita e a unicidade do token não é garantida. Além disso, a proposta de nosso algoritmo é que ele possa ser utilizado por outros algoritmos baseados em circulação do token em anel que necessitem garantir a unicidade do token em caso de falhas. Esta portabilidade não é oferecida na solução dos autores que se concentram no problema do consensus et difusão atômica (*atomic broadcast*). Uma última observação é que nosso algoritmo tolera k falhas consecutivas e não apenas k, ou seja, o número de falhas pode ser maior que k.

5. Conclusão

Apresentamos neste artigo um algoritmo que evita a perda do token enviando cópias deste a $k + 1$ nós. Nossa solução tolera até k falhas consecutivas. As funções oferecidas podem ser facilmente utilizadas por algoritmos existentes organizados logicamente em anel que precisem assegurar a unicidade do token, mas que não tratam o problema da perda do token em caso de falhas. Tanto a detecção de falhas como a criação de um novo token são transparentes a este algoritmo e implementadas de forma eficaz e escalável: um nó monitora apenas seus k nós predecessores e a criação de um token é praticamente

instantânea. Além disso, o anel não precisa ser reconstruído e, se o token contiver dados da aplicação, estes podem ser mais facilmente recuperados.

Referências

Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A., and Ciarfella, P. (1995). The totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4):311–342.

Chang, E. and Roberts, R. (1979). An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283.

Chang, I., Singhal, M., and Liu, M. T. (1990). A fault tolerant algorithm for distributed mutual exclusion. In *Proceedings of the IEEE 9th Symposium on Reliable Distributed Systems*, pages 146–154.

Chang, J.-M. and Maxemchuck, N. F. (1984). Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):351–273.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421.

Dijkstra, E. W., Feijen, W. H. J., and van Gasteren, A. J. M. (1986). Derivation of a termination detection algorithm for distributed computations. In *Proc. of the NATO Advanced Study Institute on Control flow and data flow: concepts of distributed programming*, pages 507–512.

Ekwall, R., A.Schiper, and Urbán, P. (2004). Token-based atomic broadcast using unreliable failure detectors. In *SRDS*, pages 52–65.

Franklin, R. W. (1982). On an improved algorithm for decentralized extrema finding in circular configurations of processors. *Communications of the ACM*, 25(5):336–337.

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

Manivannan, D. and Singhal, M. (1994). An efficient fault-tolerant mutual exclusion algorithm for distributed systems. In *ISCA International Conference on Parallel and Distributed Computing Systems*, pages 525–530.

Misra, J. (1983). Detecting termination of distributed computations using markers. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 290–294.

Mueller, F. (2001). Fault tolerance for token-based synchronization protocols. *Workshop on Fault-Tolerant Parallel and Distributed Systems, IEEE*.

Nishio, S., Li, K. F., and Manning, E. G. (1990). A resilient mutual exclusion algorithm for computer networks. *IEEE Trans. on Parallel and Distributed Systems*, 1(3):344–355.

Peterson, G. L. (1982). An o(nlog n) unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4):758–762.