# Um Algoritmo Eficiente para Exclusão Mútua para Recursos Redundantes em um Sistema Operacional Distribuído

Marco Antonio Dantas Ramos *†, Raimundo José de Araújo Macêdo † and Anne Blagojevic †
*Department of Exact Sciences, State University of Bahia Southwest
Vitória da Conquista, BA, Brazil
†Distributed Systems Laboratory, Computer Science Departament, Federal University of Bahia
Salvador, BA, Brazil

***Resumo*—Increasing demand for distributed applications raises the need for more reliable control of shared resources among distributed processes. Because of the inherent redundancy of distributed systems, algorithms for critical section access must consider not only the shared resource but also the possibility of redundancy of such a resource. These algorithms are thus known as *distributed k-mutex* since they control access to k versions of the same shared resource. The inherent uncertainties of distributed systems, like unbounded message transfer delay and the difficulty to detect failures, make the distributed k-mutex problem hard to solve. In this paper, we present a k-mutex algorithm that runs under the partitioned synchronous model, which have recently been formalized, which guarantee *liveness* (*termination*) without being restrictive as those conventional solutions of synchronous systems. We complement the presentation of our algorithm with related correctness proofs and simulations that show the efficacy of the proposed solution.**

***Resumo*—A crescente demanda por aplicações distribuídas leva à necessidade do controle mais robusto a recursos compartilhados entre processos distribuídos. Dada a característica de redundância inerente aos sistemas distribuídos, os algoritmos propostos para controle de acesso à região crítica devem levar em consideração não apenas o acesso a um determinado recurso compartilhado, mas também a possibilidade de redundância deste recurso. Tais algoritmos são conhecidos como *"distributed k-mutex"*, os quais tratam do controle de acesso a uma quantidade k de um mesmo recurso compartilhado. As incertezas inerentes aos sistemas distribuídos tais como atrasos aleatórios na rede e dificuldades de detecção de falhas, tornam o problema de exclusão mútua distribuída de difícil solução. Neste artigo apresentamos um algoritmo k-mutex que roda sobre o modelo partitioned synchronous, recentemente formalizado, e que garante liveness sem ser demasiadamente restritivo como soluções convencionais para sistemas síncronos. A apresentação do algoritmo é complementada com provas formais de correção e simulações onde se pode aferir a eficácia da solução.**

## I. Introdução

A exclusão mútua se configura pelo controle de acesso a recursos compartilhados do sistema por diversos processos. Mais especificamente, o controle de acesso a uma parte do código chamada *Critical Section - CS*. Mesmo em sistema de tempo compartilhado e não distribuído, a implementação de exclusão mútua não tem soluções triviais. Em sistemas distribuídos algumas abordagens foram propostas. De modo geral, a forma mais simples de garantir acesso exclusivo a CS num sistema distribuído é através do uso de um processo coordenador/servidor. Tal processo enfileira os pedidos de acesso a CS e libera um *token* para que o processo possa acessar a CS. Obviamente, essa solução apesar de simples cria um gargalo que é o processo servidor. Neste mesmo sentido, outra solução utilizada é a disposição dos N processos em um anel lógico, onde cada processo $p_i$ deve manter uma comunicação com o processo seguinte $p_{(i+1)}$ mod N. Assim, cada processo quando recebe uma mensagem com o *token* do processo antecessor pode então acessar a *CS*; caso o processo com o *token* não deseje acessar a *CS*, este reenvia a mensagem com o *token* para o processo seguinte e assim por diante. Outras soluções são baseadas em um conjunto de permissões, neste caso o processo que deseja acessar a *CS* aguarda a formação de um *Quorum* de permissões. Nesta categoria de soluções, além da possibilidade de gargalos ser reduzida, a formação de *Quoruns* facilita a possibilidade de implementação de tolerância a falhas. Em suma, as soluções propostas podem ser classificadas basicamente nesses dois grupos [1]: os baseados em permissão [2] [3] e os baseados em *token* [4] [5] [6]. Algumas abordagens também levam em consideração a tolerância a falhas, como em [7].

Como vimos, dada a característica de redundância em sistemas distribuídos é plausível estender a visão de apenas um recurso compartilhado para uma quantidade k deste mesmo recurso no sistema. Assim, em um sistema com $\Pi$ processos, o algoritmo deve garantir que não mais que k processos estejam acessando a *CS* ao mesmo tempo. Algoritmos deste tipo são chamados de algoritmos k-exclusão mútua distribuídos ou *(distributed k-mutex algorithm)*. Neste contexto, Raymond [2] propôs um algoritmo k-mutex, seu algoritmo, no entanto, não é tolerante a falhas. Para contornar tal limitação, Bouillaguet, Arantes e Sens [8] estenderam a proposta de Raymond para um algoritmo k-mutex tolerante a falhas num modelo de sistema distribuído time-free. Tal algoritmo tem resiliência $f \leq k - 1$, onde f representa o número máximo de processos que podem falhar. Apesar de time-free ser um modelo bastante abrangente e interessante do ponto de vista de projeto de algoritmos, ele não permite a verificação da terminação de alguns algoritmos. Em particular, a solução proposta em [8]

depende da propriedade *Responsiveness Property - RP* descrita por Mostefaoui, Mourgaya and Raynal em [9]. Tal propriedade é equivalente às propriedades de um detector de falhas do tipo $\diamond$S [10], que, por sua vez, não são implementáveis em um sistema time-free. Portanto, em um sistema time-free o algoritmo de Bouillaguet, Arantes e Sens [8] não garante terminação.

O presente trabalho apresenta uma solução tolerante a falhas para o problema k-mutex distribuídos sem as restrições de terminação encontradas na solução proposta em [8]. Para tanto, propomos um algoritmo sobre o modelo *partitioned synchronous* [11], baseado em partições síncronas interconectadas por canais assíncronos onde, em cada partição, existe pelo menos um processo correto (que não falha). Exemplos de sistemas deste tipo são clusters, os quais são síncronos localmente, mas são conectados a outros clusters por canais assíncronos tipo TCP/IP. Como será mostrado ao longo deste trabalho, mostramos que com nosso algoritmo é possível garantir a terminação do k-mutex, além do aumento da resiliência quando comparado à solução proposta por Bouillaguet, Arantes e Sens [8].

A fim de mostrar a funcionalidade e desempenho de nossa proposta, utilizamos o simulador *HDDS (Hybrid Dynaminc Distributed System simulator)* [12]. Nesse sentido, não só o algoritmo proposto neste trabalho foi simulado, mais também o algoritmo k-mutex proposto em [8], por se aproximar de nossa abordagem e também, por já ter sido simulado em outro ambiente como descrito em [8]. Os resultados obtidos nas simulações mostraram um melhor desempenho de nosso algoritmo na presença de falhas.

Este trabalho está organizado da seguinte maneira. A seção 2 trata dos principais trabalhos relacionados. A seção 3 discute o modelo do sistema Spa e suas principais características. A seção 4 descreve o algoritmo k-mutex que foi usado para a proposta deste trabalho. A seção 5 discute os experimentos e resultados obtidos. Finalmente a seção 6 trata das conclusões e trabalhos futuros.

## II. Trabalhos Relacionados

Em [13] três algoritmos baseados em alternância estrita são apresentados. Basicamente os algoritmos fazem uso de uma área de memória compartilhada para acessar a CS. Os processos explicitam a vontade de acessar a *CS* escrevendo em uma posição da memória compartilhada o seu identificador único - UID, em seguida fazem uma leitura de outra posição para verificar se algum processo está acessando a *CS*. Caso não haja processos em *CS*, o processo, então, escreve seu UID na memória indicando que ele irá acessar *CS* e, então, entra na *CS*.

Outra abordagem é o uso de *token*. O algoritmo proposto em [14] faz uso de m *tokens* numerados de 1 a m. Cada processo pode ter no máximo 1 solicitação de acesso a *CS*. Caso um processo possua qualquer um dos *tokens*, ele acessa a *CS* sem atrasos, no entanto, caso contrário, ele envia uma solicitação e fica aguardando. Comparado com o algoritmo proposto em [2], a redução na quantidade de mensagens necessária para entrada em *CS* é de aproximadamente 2(N/K-1). No entanto, o mesmo não é tolerante a falhas e a quantidade reduzida de mensagens pode levar a uma demora maior para acessar *CS*.

Ricart e Agrawala [15] [16] propuseram um algoritmo para acesso a *CS* baseado em permissão (formação de quorum) e relógios lógicos. O valor do relógio lógico de cada processo é usado para enfileirar as solicitações, caso o processo esteja em *CS* ou aguardando para acessar e o valor do seu relógio lógico seja menor que o do processo solicitante.

Estas abordagens não levam em consideração recursos redundantes. No entanto, como mencionado anteriormente, outras abordagens levam em consideração o controle de acesso a recursos redundantes e compartilhados em sistemas distribuídos, mais especificamente, a uma quantidade k deste mesmo recurso no sistema. Assim, em um sistema com $\Pi$ processos, o algoritmo deve garantir que não mais que k processos estejam acessando a *CS* ao mesmo tempo *(safety)*, desta forma, caso um processo receba permissão para acessar *CS* não mais que k-1 processos estão em *CS*. Neste contexto, em [8], é proposto um algoritmo do tipo k-mutex e tolerante a falhas, com resiliência $f \leq k - 1$. Este algoritmo foi baseado no algoritmo proposto em [2] e, além de introduzir características de tolerância a falhas, também conseguiu a diminuição no número de mensagens trocadas para a acesso a *CS*. Neste algoritmo uma condição necessária para a sua corretude é que o subsistema de comunicação satisfaça a propriedade *Respossiveness Property – RP* descrita em [9]. Basicamente, se um sistema verifica *RP* então, implica dizer que, a partir de um tempo t um processo correto $p_i$ não será suspeito por pelo menos um outro processo correto $P_j$ , tal que $p_i \neq p_j$ . Essa propriedade equivale à propriedade *eventually weak accuracy* dos detectores de defeitos [10]. Como se sabe, tal propriedade não é implementável em sistemas time-free o que compromete a utilidade prática da solução proposta em [8].

Para a construção do conjunto contendo os processos falhos (conjunto $faulty_i$), o algoritmo proposto em [8] começa "desconfiado"de todos os demais processos exceto ele próprio e, a medida que os processos vão respondendo a sua solicitação vão deixando de ser suspeitos de falha. Assim, quando a quantidade de suspeitos é menor ou igual a f o algoritmo faz a intersecção dos conjuntos de faltosos dos processos que responderam sua solicitação e adiciona ao seu próprio conjunto de faltosos. Portanto, para que um processo seja inserido em $faulty_i$ ou ele não respondeu uma solicitação de $p_i$ ou ele pertence a todos os conjuntos *faulty* dos processos que responderam a $p_i$. Para acesso a *CS* $p_i$ envia para todos os demais processos sua solicitação, juntamente com seu conjunto *faulty*, então, ele aguarda a formação do quorum $(Q \geq |\Pi - faulty_i| - k)$; quando isso acontece, ele acessa a *CS*. Devido a característica do modelo Spa, assumido em nosso trabalho, como será mostrado adiante, o algoritmo k-mutex proposto neste trabalho constrói seu conjunto faulty sem a necessidade de suspeitar todos os processos envolvidos, o que o torna mais eficiente e com uma resiliência maior que o a proposta descrita em [8].

## III. MODELO DE SISTEMA

Os processos falham por *crash*, deixando de gerar informação. Para simplificar a apresentação, é também assumido que os processos, mesmo que assincronamente, sempre solicitam acesso a CS. O algoritmo k-mutex será executado sobre o modelo Spa, descrito resumidamente a seguir.

### A. O modelo síncrono particionado - Spa [11]

O sistema é composto por um conjunto de processos $\Pi = p_1, p_2, \ldots, p_n$, possivelmente pertencentes a diferentes redes e um conjunto $\chi = c_1, c_2, \ldots, c_m$ de canais de comunicação confiáveis, ou seja, não perdem nem modificam as mensagens. A topologia da rede é arbitrária e os processos se comunicam por meio de protocolos de transporte que implementam uma comunicação processo-a-processo.

Tanto os processos quanto os canais de comunicação podem ser síncrono *(timely)* ou assíncrono *(untimely)*. Um processo é dito síncrono quando existe um limite superior $\phi$, conhecido, para o processamento interno de suas informações, caso contrário ele é dito assíncrono. Já um canal de comunicação é dito síncrono caso exista um limite superior $\delta$, conhecido, para comunicação processo a processo, da mesma forma, caso não exista este limite o canal é dito assíncrono. Caso o canal de comunicação entre dois processos síncronos seja também síncrono, então, estes pertencem à mesma partição, caso contrário estão em partições distintas. Da mesma forma, se o canal de comunicação entre dois processos é assíncrono, implica dizer que eles pertencem a partições distintas. Assim, o sistema é composto por partições síncronas interconectadas por canais assíncronos. No modelo Spa, é assumida a existência de um oráculo "temporal"definido pela função QoS o qual mapeia canais e processos com os labels T ou U (*timely* e *untimely*, respectivamente). Portanto, o oráculo executando nos processos os provê de informações temporais acerca dos demais processos, e dos canais de comunicação. A implementação desse oráculo é trivial e descrita em [11].

## IV. ALGORITMO DE EXCLUSÃO MÚTUA EM SPA COM DETECTOR DE FALHAS INCORPORADO

O algoritmo apresentado nesta seção além de controlar o acesso a *CS* também possui linhas dedicadas a detecção de falhas. Esta abordagem permite a detecção de falhas sem a necessidade de outro módulo sendo executado no processo. Assim como em [15], nossa abordagem também faz uso de relógios lógicos para garantir a sequencialização das solicitações. Além disso, o quorum utilizado para acesso a *CS* é o mesmo proposto em [8], pois garante a propriedade de *safety*, como será provado a frente.

O algoritmo 1 leva em consideração sua execução em $p_i$, e é composto por SEIS PARTES. A primeira é dedicada a solicitação de entrada em CS, nesta instancia é feita a atualização do estado ($state_i$)do processo para *requesting*, a variável temporária que armazena o valor do relógio lógico de $p_i$ ($last_i$) também é incrementada em 1 e o vetor de permissão (*haveperm$_i$[ ]*) é zerado (linas 4 a 6). O vetor binário *haveperm$_i$[ ]* armazena true (PERM) ou false (NOPERM) a depender da resposta dada por cada processo a solicitação de $p_i$. É realizada, então, a atualização do *Timeout* ($Timeout[p_j] \longleftarrow CT_i + 2\delta + \alpha$), aqui, é considerado o valor atual do relógio físico do processo ($CT_i$), o *upperbound* do RTT($2\delta$) e uma margem de segurança ($\alpha$). Em seguida é feito o envio da solicitação de $p_i$ para todos os processos $p_j$ ($p_j \neq p_i$ e $p_j \notin faulty_i$). É importante notar que, juntamente com sua solicitação $p_i$ envia também seu conjunto *faulty* (i; $last_i, faulty_i$), permitindo que cada processo $p_j$ possa atualizar seu próprio conjunto com algum processo que por ventura ainda não tenha sido detectado pelo seu módulo de detecção. Finalmente, $p_i$ aguarda o quorum ($|\Pi - faulty_i| - k$) ser alcançado, ele faz isso contando a quantidade de valores true no vetor de permissões, só aí ele pode acessar CS mudando, assim, seu estado.

A segunda parte do algoritmo trata resolicitação de entrada em *CS*, necessária, pois, caso um processo $p_j$ responda com NOPERM seu TimeOut deve ser atualizado(linha 17), fazendo com que $p_i$ não o invalide antes de receber PERM ou $p_j$ falhar. Desta maneira, quando $p_i$ recebe NOPERM de $p_j$, este será inserido no conjunto *using$_i$* (linha 56). Então, para todos os processos em *using$_i$* que não expiraram seus TimeOuts, $p_i$ enviará uma nova solicitação, a fim de manter TimeOut$_j$ atualizado.

A terceira etapa é dedicada a saída de CS (RELEASE). O algoritmo envia apenas para os processos $p_j$ que estão aguardando uma notificação ($p_j \in (pending_i / faulty_i)$) (linhas 25 e 26). Finalmente $p_i$ atribui vazio ao conjunto de pendentes e muda seu estado para not_requesting.

Já a quarta parte é dedicada exclusivamente a atualização do conjunto $faulty_i$. No entanto, neste momento, só são considerados os processos falhos da mesma partição de $p_i$ (canal de comunicação $(c_{i/j})$ é síncrono), pois, para a detecção de falhas, é considerado apenas a extrapolação do timeout do processo $p_j$ ($C_i > Timeout_i[p_j]$), desta forma, quando estas condições são satisfeitas por um processo $p_j \neq p_i$ e $p_j \notin faulty_i$, então o processo $p_j$ é inserido no conjunto $faulty_i$ (linhas 31 e 32), e uma notificação é enviada aos demais processos. Caso contrário, então o algoritmo não executa nenhuma ação, esperando por uma notificação remota de outra partição.

Já a quinta etapa é dedicada ao tratamento de mensagens de solicitação REQUEST($j; C_j; faulty_j$). Inicialmente o relógio lógico de $p_i$ ($C_i$) é atualizado com o máximo entre seu próprio relógio e o do processo solicitante incrementado de 1 (linha 38). Após atualizar seu relógio lógico, $p_i$ desconsidera as permissões $haveperm_i[k]$ dadas por processos que vieram a falhar, a fim de evitar que sejam considerados na formação do quorum. Se $p_i$ estiver em CS ou estiver em *pending* e seu relógio lógico era menor que o de $p_j$ no momento da solicitação, então $p_i$ reponderá com NOPERM < i; NOPERM > para $p_j$ e atualiza seu conjunto *pending$_i$* com o ID de $p_j$. Caso contrário, $p_i$ envia a tupla < i; PERM > para $p_j$, autorizando sua entrada em CS.

Finalmente, a sexta etapa é dedicada ao tratamento de

**Algorithm 1** Algoritmo para exclusão mútua com detector de falhas incorporado

1: $C_i \leftarrow 0$
2: $faulty_i \leftarrow \emptyset$
3: $pending_i \leftarrow \emptyset$
4: $state_i \leftarrow not\_request$
5: **Request resource()**:
6: $using_i \leftarrow \emptyset$
7: $state_i \leftarrow requesting$
8: $last_i \leftarrow C_i + 1$
9: $haveperm_i[] \leftarrow false$
10: **for all** $p_j \neq p_i : p_j \notin falty_i$ **do**
11: Timeout[$P_j$] $\leftarrow$ $CT_i$ + 2$\delta$ + $\alpha$
12: send REQUEST($i, last_i, faulty_i$) to $p_j$
13: **end for**
14: **Re-request resource()**:
15: **if** using$_i \neq \emptyset$ **then**
16: **for all**$p_j \neq p_i : (p_j \notin falty_i \wedge p_j \in using_i)$ **do**
17: Timeout[$p_j$] $\leftarrow$ $CT_i$ + 2$\delta$ + $\alpha$
18: send REQUEST($p_i, last_i, faulty_i$) to $p_j$
19: using$_i \leftarrow using_i - p_j$
20: **end for**
21: **end if**
22: **wait until** (Count perm(have perm$_i$)$\geq (|\pi - faulty_i| - k)$)
23: state$_i \leftarrow$ CS
24: **Release resource():**
25: **for** all($p_j \neq p_i : p_j \notin$ (pending/faulty$_i$)) **do**
26: send REPLY($p_i$;PERM) to $p_j$
27: **end for**
28: pending$_i \leftarrow \emptyset$
29: state$_i \leftarrow$ not_request;
30: **Updating faulty set**
31: **when** $\exists$(p$_i$ $\notin$ faulty$_i$)$\wedge$(CT$_i$ > Timeout[p$_j$])$\wedge$ (QoS$_{(c_i/j)}$)=T **do**
32: faulty$_i \leftarrow faulty_i \cup \{p_j\}$
33: **for all**$p_j \neq p_i : (p_j \notin falty_i)$ **do**
34: send NOTIFICATION($p_i, faulty_i$) to $p_j$
35: **end for**
36: upon receive REQUEST($p_j$;last$_i$;faulty$_i$) **do**
37: Timeout[p$_j$] $\leftarrow \infty$
38: C$_i \leftarrow$ max(C$_i$;C$_j$) + 1
39: **for all** $k \in faulty_j / faulty_i$ **do**
40: **if** have perm$_i$[k] **then**
41: have perm$_i$[k] = false
42: faulty$_i \leftarrow faulty_i \cup faulty_j$
43: **end if**
44: **end for**
45: **if** (state$_i$ = CS) $\vee$ (state$_i$=requesting $\wedge$ (last$_i$;i)<(C$_j$;j)) **then**
46: send REPLY ($p_i$;NOPERM;faulty$_i$) to $p_j$
47: pending$_i \leftarrow$ pending$_i \cup \{p_j\}$
48: **end if**
49: send REPLY ($p_i$;PERM;faulty$_i$) to $p_j$
50: **upon** receive REPLY ($p_j$;ack;faulty$_j$) **do**
51: faulty$_i \leftarrow faulty_i \cup faulty_j$
52: **if** (state$_i$=requesting)$\wedge$ (ack=PERM)$\wedge(p_j \notin faulty_i)$ **then**
53: Timeout[p$_j$] $\leftarrow \infty$
54: have perm$_i$[j] = true
55: **else if** (state$_i$=requesting)$\wedge$ (ack=NOPERM)$\wedge(p_j \notin faulty_i)$ **then**
56: using$_i \leftarrow$ using$_i \cup p_j$
57: **else**
58: using$_i \leftarrow$ using$_i$ - {p$_j$}
59: Timeout[p$_j$] $\leftarrow \infty$
60: **end if**
61: **upon** NOTIFICATION($p_j$;faulty$_i$) ARRIVES**do**
62: faulty$_i \leftarrow$ faulty$_i \cup$ faulty$_j$

mensagens de resposta a solicitação de $p_i$ (REPLAY). Como já foi mostrado anteriormente, caso p$_i$ receba uma mensagem com Ack$_j$=PERM, TimeOut$_j$ é cancelado, a posição de P$_j$ no vetor de permissão é atualizado, e o conjunto $faulty_i$ é atualizado com base no conjunto $faulty_j$ (linhas 52 a 55), ou seja, caso haja algum processo que $p_j$ tenha detectado com falho e $p_i$ não, então $p_i$ o adicionará ao seu conjunto de faltosos. Note que, esta abordagem contempla a recepção remota de notificação por processos que não estejam na mesma partição de $p_i$. Já, quando p$_i$ recebe Ack$_j$=NOPERM e seu estado ainda é *pending*, p$_j$ é inserido em $using_i$. Caso contrário, p$_j$ é retirado de $using_i$ e seu TimeOut é cancelado.

*Teorema 1:* o algoritmo de exclusão mútua satisfaz a propriedade safety. Isto significa que, caso um processo $p_i$ obtenha acesso a CS, não mais que k-1 processos estarão acessando CS neste instante. Formalmente temos, Quorum $\geq |(\Pi - faulty_i)| - k \Longrightarrow p_{cs} \leq k - 1$; (onde $p_{cs}$ é o número de processos em CS).

**Prova:** A prova será feita por contradição. Suponhamos para tanto que no tempo t, k processos estejam acessando CS ($p_{cs} = k$) e $p_i$ obtenha acesso a CS (Quorum $\geq |(\Pi - faulty_i)| - k$). Como, $p_i$ só pode receber PERM de um processo p$_j$ se ele não está acessando CS (linha21) ou não está no estado de *pending* com o valor de relógio lógico menor (linhas 45 e 46). Nestas circunstâncias, como $p_i$ não recebe mensagem dele mesmo e, os canais de comunicação são confiáveis, não trocando os valores das mensagens enviadas. Então, o máximo de mensagens contendo PERM que p$_i$ pode receber é $|(\Pi - faulty)| - k - 1$. O que contradiz a premissa Quorum $\geq |(\Pi - faulty_i)| - k$. ■ Teorema 1.

*Lema 1:* O detector de falhas incorporado ao algoritmo k-mutex verifica eventual completude forte. Ou seja, a partir de um tempo t, todos os processos falhos serão inseridos em *faulty*.

**Prova:** Assim, seja t o tempo em que p$_i$ solicita entrada em *CS*, e seja p$_j$, tal que, p$_j \neq$ p$_i$. Assim seja o tempo t', tal que, t' > t, o tempo em que p$_j$ falhe, então, pela premissa de Spa, onde há pelo menos um processo correto por partição, e o fato de que os processos sempre solicitam acesso a *CS*, em um tempo t", tal que, t" $\geq$ TimeOut$_j$, p$_j$ será inserido permanentemente

no conjunto $faulty_i$ e uma notificação é enviada a todos os demais processos para que atualizem seus conjuntos *faulty* (linhas 31 a 34).■ lema1.

*Lema 2:* O detector de falhas incorporado ao algoritmo k-mutex verifica acurácia forte. Ou seja, em algum instante t, nenhum processo correto será inserido no conjunto *faulty*.

**Prova:** A prova será feita por contradição. Suponha que em um tempo t $p_j$, tal que, $p_j \neq p_i$, não tenha falhado, e $p_j$ seja inserido em $faulty_i$. Assim, existem dois casos para que $p_j$ seja inserido em $faulty_i$. No primeiro caso, $p_j$ pertence a mesma partição de $p_i$, então para que $p_j$ seja inserido a $faulty_i$, é preciso que seu TimeOut expire, no entanto, como $p_j$ não falhou até o tempo t, então ele envia para $p_i$ uma notificação ou com PERM, e neste caso seu TimeOut é cancelado (linha37), ou NOPERM e neste caso seu TimeOut é renovado na resolicitação (linha 17). O que leva a uma contradição. O segundo caso é $p_j$ não pertencer a mesma partição de $p_i$, neste caso o algoritmo desconsidera o TimeOut e não faz nada. O que também leva a uma contradição. ■ lema2.

*Teorema 2:* o algoritmo de exclusão mútua verifica a propriedade de terminação. Ou seja, um processo $p_i$ que solicite acesso a CS, obterá acesso a CS em um tempo finito.

Prova: No primeiro caso, no máximo k – 1 processos estão acessando CS no tempo t, no qual $p_i$ solicita acesso e, nenhum outro processo $p_j$, tal que $p_j \neq p_i$, esteja em *request*. Então, pelo Teorema 1 (formação do quorum)e pela execução das linha 49 do algoritmo, $p_i$ receberá o direito de acessar CS.

O segundo caso leva em consideração o fato de k processos estarem acessando CS no tempo t. Neste caso, quando uma solicitação de $p_i$ chegar a qualquer $p_j$, tal que $pj \notin faulty_i$, e $p_j$ esteja acessando CS (Lina 45), temos que, $p_j$ responderá a $p_i$ com NOPERM e adicionará $p_i$ em $pending_j$ (linha 46). E,em algum instante t' > t, quando $p_j$ deixar CS, ele enviará uma mensagem a $p_i$ com o valor PERM (linhas 25 e 26), possibilitando que $p_i$ possa considerar $p_j$ em seu vetor de permissão. Finalmente, pelos lema1 e 2 os processos faltosos não serão considerados na formação do quorum o que poderia comprometer a terminação ■ Teorema 2.

Portanto, baseado no Teorema 1, nossa abordagem garante a a terminação mesmo que (|Π| - 1) falhem. No entanto não garantimos deadlines para a terminação do algoritmo.

## V. Experimentos e avaliação de performance

Para verificar a robustez do algoritmo proposto anteriormente foi feita sua implementação no simulador HDDS (Hybrid and Dynamic System simulator) [12]. O algoritmo proposto em [8] também foi implementado, por já ter sido implementado em outro ambiente e também por ter abordagem semelhante a deste trabalho.

Desta forma o simulador foi configurado com 10 partições e 10 processos por partição. O número de slots para CS (k) também foi definido como 10 e o número de falhas inseridos foi igual a resiliência, ou seja f = 9 e foram inseridas aleatoriamente seguindo uma distribuição de probabilidade.

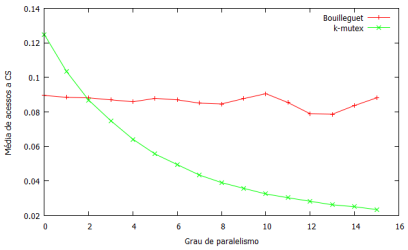


Figura 1. largura de banda

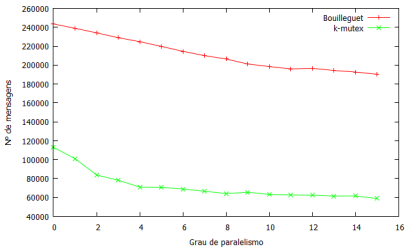


Figura 2. mensagens

Duas outras métricas também são utilizadas na simulação. A primeira é representada por $\alpha$, e trata do tempo que o processo leva para concluir seu acesso a *CS*. A segunda é representada por $\beta$, e trata do tempo decorrente entre o instante que o processo deixa *CS* e uma nova solicitação sua. A razão entre $\beta$ e $\alpha$ é o grau de paralelismo do sistema e é representado por $\rho$. Neste trabalho o valor de $\alpha$ é fixo e igual a 20 unidades de tempo, já o valor de $\rho$ varia entre 1 e 16. Neste contexto, quanto maior o valor de $\rho$ menor é a frequência que um processo solicita acesso a *CS*.

Para a construção do modelo de simulação foram consideradas as seguintes métricas:

- largura de banda: quantidade média de acessos a *CS* concluídos por unidade de tempo;
- tempo de espera: tempo necessário para os processos acessarem *CS*, para um determinado grau de paralelismo;
- overhead da troca de mensagens: quantidade de mensagens trocadas para um determinado grau de paralelismo;
- Ocupação de *CS* (eficiência): quantidade média de processos em *CS* ao mesmo tempo;

### A. Avaliação dos resultados

Com o aumento de $\rho$ é de se esperar que a quantidade de acessos concluídos por unidade de tempo também diminua. Dada a característica do detector da abordagem feita por Bouilleguet [8], na presença de falhas, o algoritmo demora mais tempo para formar seu quorum, já que começa "desconfiando"de todos os processos e também porque não faz uso das características temporais das partições. Assim, como pode ser observado na Figura 1, enquanto nossa abordagem diminui proporcionalmente a média de acesso a *CS* com o aumento de $\rho$, o outro algoritmo tem comportamento inverso.

Em relação ao overhead na troca de mensagens para acesso a *CS*, conseguimos também uma diminuição. Nossa aborda-

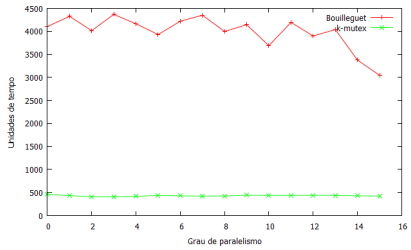


Figura 3. tempo de espera

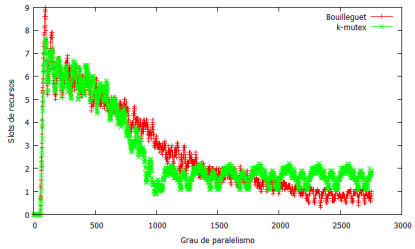


Figura 4. Ocupação de *CS*

gem fez um uso médio de 84% a menos na quantidade de mensagens trocadas, comparado com o algoritmo de Bouilleguet. Como pode ser observado na Figura 2.

A Figura 3 mostra os tempos que os processos levam aguardando para acessar *CS*. Devido a estratégia de detecção de falhas adotada por nossa abordagem, os tempos para a nossa abordagem ficaram sempre abaixo de 500 unidades de tempo. Contrastando com o outro algoritmo o qual variou entre 3000 a 4300 unidades de tempo.

Finalmente, em relação a ocupação dos slots de recursos compartilhados ao longo do tempo, os gráficos mostram curvas semelhantes. No entanto, esta métrica é diretamente influenciada pelo tempo que um processo demora para acessar a *CS*. Neste caso, como visto na Figura 3, o tempo para que um processo seja atendido em nossa abordagem chega a ser 88% menor que o da abordagem de Bouilleguet. Isso faz com que os processos deixem a *CS* mais rapidamente também. Portanto, nunca atingimos a ocupação máxima de slots do recurso compartilhado. Como pode ser observado na Figura 4.

## VI. Conclusão

Neste trabalho propomos um algoritmo do tipo k-mutex tolerante a falhas e executando sobre um sistema síncrono particionado, no caso o Spa [11]. Esta categoria de solução para exclusão mútua distribuída, baseia-se na formação de quorum ($Q \geq |\pi$ - $faulty_i|$-k) para liberação de acesso a *CS*, além do mais, dadas as características síncronas das partições de Spa, o mecanismo de detecção de falhas foi baseado em *TimeOuts*. Além disso, foram utilizados os relógios lógicos dos processos a fim de garantir que os processos com solicitações mais antigas ganhem prioridade sobre os mais novos. Esta abordagem, apesar de não favorecer a modularização do sistema, diminui consideravelmente o *overhead* na troca de mensagens, pois, as mesmas mensagens trocadas para a formação do quorum são também utilizadas para a detecção de processos falhos.

Para mostrar funcionalidade e a performance do algoritmo, o mesmo foi implementado e simulado no ambiente HDDS. Além disso a abordagem proposta em [8] também foi implementada para propiciar a analogia dos resultados de ambos. Os resultados mostraram que, na presença de falhas, nossa abordagem conseguiu um desempenho melhor que o algoritmo proposto em [8], e, na ausência de falhas, o comportamento dos dois algoritmos foi similar.

Como trabalho futuro, pretendemos utilizar um processo líder em cada partição. Tal processo será responsável pela comunicação com os líderes das demais partições. Assim, evitando o envio de mensagens de todos para todos entre partições, diminuímos o overhead de mensagens. Para a eleição do processo líder será utilizado um detector do tipo $\Omega$ operando com o auxilio de registradores distribuídos, os quais armazenarão os UID dos processos líderes de todas as partições.

## Referências

[1] M. Raynal, "A simple taxonomy for distributed mutual exclusion algorithms," *ACM SIGOPS Operating Systems Review Homepage archive*, vol. 25 Issue 2, 1991.

[2] K. Raymond, "A distributed algorithm for multiple entries to a critical section," *Information Processing Letters*, vol. 30 Issue 4, 1989.

[3] P. N., M. K. Park, H. E. K. Z., and W. W., "An efficient distributed mutual exclusion algorithm." *ICPP*, 1996.

[4] A. Lodha, S. Kshemkalyani, "A fair distributed mutual exclusion algorithm," *IEEE Transactions on Parallel and Distributed Systems archive*, 2000.

[5] C. W., L. S., L. Q., and Z. Z., "Sigma: a fault-tolerant mutual exclusion algorithm in dynamic distributed systems subject to process crashes and memory losses," *Dependable Computing, Proceedings. 11th Pacific Rim International Symposium*, 2005.

[6] G. Walter E. Jennifer Cao and M. Mohanty, "A k-mutual exclusion algorithm for wireless ad hoc networks," *POMC*, 2001.

[7] D.-G. C., F. H., G. R., and K. P., "Mutual exclusion in asynchronous systems with failure detectors," *JPDC*, 2005.

[8] L. Bouillaguet, M. Arantes and P. Sens, "Fault tolerant k-mutual exclusion algorithm using failure detector,," *ISPDC*, 2008.

[9] M. E. Mourgaya and M. Raynal, "Asynchronous implementation of failure detectors," *DSN*, 1989.

[10] C. T. D. and T. S., "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, 1996.

[11] R. J. A. Macêdo and S. Gorender, "Exploiting partitioned synchrony to implement accurate failure detectors," *Int. J. Critical Computer-Based Systems.*, 2010.

[12] R. J. d. A. Freitas, A. E. S. Macêdo, "Um ambiente para testes e simulações de protocolos confiáveis em sistemas distribuídos híbridos e dinâmicos," *In Proceedings of 27th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC2009)*, 2009.

[13] L. Lamport, "A fast mutual exclusion algorithm," *ACM Transactions on Computer Systems*, vol. 5, no. 1, 1985.

[14] S. Bulgannawar and N. H. Vaidya, "A distributed k-mutual exclusion algorithm," *ICDCS '95 Proceedings of the 15th International Conference on Distributed Computing Systems*, 1995.

[15] T. Ricart and A. K. Agrawala, "A optimal algorithm for mutual exclusion in computer networks," *Communications of the ACM*, vol. 24 Issue 1, 1981.

[16] D. Agrawala and A. E. Abbadi, "An efficient and fault-tolerant solution for distributed mutual exclusion," *ACM Transactions on Computer Systems (TOCS)*, 1991.