O Confeiteiro Bizantino: Exclusão Mútua em Sistemas Abertos Sujeitos a Faltas Bizantinas.

Alysson Neves Bessani[1]*, **Joni da Silva Fraga**[1] , **Lau Cheuk Lung**[2]

[1]DAS - Departamento de Automação e Sistemas
UFSC - Universidade Federal de Santa Catarina

[2]PPGIA - Programa de Pós-Graduação em Informática Aplicada
PUC-PR - Pontifícia Universidade Católica do Paraná

{neves,fraga}@das.ufsc.br, lau@ppgia.pucpr.br

Abstract. *The Bakery algorithm [Lamport, 1974] is a classical solution for the mutual exclusion problem in shared memory. This paper presents this algorithm adaptation for infinitely many processes that are subject to byzantine failures in a message passing communication system. This adaptation results in the byzantine bakery algorithm which is the first one for mutual exclusion that considers byzantine processes and guarantees that no process suffers starvation.*

Resumo. *O algoritmo do confeiteiro [Lamport, 1974] é uma solução clássica para o problema de exclusão mútua em memória compartilhada. Este artigo apresenta uma adaptação desse algoritmo para um cenário com infinitos processos sujeitos a falhas bizantinas em um modelo de comunicação por passagem de mensagens. Esta adaptação da origem ao algoritmo do confeiteiro bizantino, o primeiro algoritmo de exclusão mútua a garantir progresso justo em um cenário com processos maliciosos.*

1. Introdução

O **algoritmo do confeiteiro (*bakery*)** [Lamport, 1974] é um dos algoritmos mais notáveis já produzidos para a resolução do problema de exclusão mútua. Este algoritmo apresenta algumas características extremamente interessantes como o uso de registradores seguros (*safe*) para um escritor e múltiplos leitores [Lamport, 1986] e a garantia de progresso justo (*starvation-freedom*) [Anderson et al., 2003]. A primeira característica permite que o tipo de registrador compartilhado mais simples seja usado em sua implementação enquanto a segunda garante que todo processo que deseja um recurso compartilhado tenha sua chance de obtê-lo.

Este algoritmo trabalha de forma análoga a uma confeitaria, cada um dos processos que deseja o recurso obtém um *ticket* numerado e o atendimento é feito de acordo com a ordem numérica crescente dos *tickets* [Lamport, 1974, Lynch, 1996]. O algoritmo do confeiteiro considera um conjunto fixo de n processos livres de falha que se comunicam de forma assíncrona através de memória compartilhada.

O presente trabalho estende o algoritmo do confeiteiro e propõem um modelo para implementação do mesmo em sistemas distribuídos assíncronos onde infinitos processos

*Doutorando Bolsista PGI/CNPq.

[Aguilera, 2004] sujeitos a faltas bizantinas [Lamport et al., 1982] se comunicam através de passagem de mensagens. Este cenário de execução se caracteriza por ser o pior possível para a resolução de exclusão mútua e, portanto, uma série de mecanismos desenvolvidos recentemente são integrados ao algoritmo para adequá-lo a este modelo. Em particular, são utilizados sistemas de quóruns bizantinos [Malkhi and Reiter, 1998a], detectores de mudez [Doudou et al., 1999] e detectores de comportamento malicioso [Baldoni et al., 2003]. A aplicação destas técnicas dá origem ao algoritmo do confeiteiro bizantino.

O algoritmo do confeiteiro bizantino usa objetos de memória compartilhada implementados sobre um sistema de quóruns bizantinos [Malkhi and Reiter, 1998a]. Neste sistema temos um conjunto de servidores sujeitos a falhas bizantinas que formam um mecanismo de armazenamento confiável compartilhado usado por processos clientes. Além das técnicas para tolerância a faltas bizantinas, este novo algoritmo também utiliza o conceito de conjunto ativo [Afek et al., 1999] para suportar infinitos processos ao invés dos "n" usualmente previstos em algoritmos distribuídos. O protocolo apresentado neste trabalho é o único a implementar conjunto ativo sobre um sistema de quóruns e também o primeiro a tolerar faltas bizantinas.

O artigo está organizado da seguinte forma: a seção 2 apresenta os sistemas de quóruns bizantinos e o modelo de sistema considerado neste trabalho. Na seção 3 são descritos os objetos de memória compartilhada usados no algoritmo e sua implementação sobre um sistema de quóruns bizantinos. O algoritmo do confeiteiro bizantino é apresentado na seção 4 e na seção 5 temos um modelo de implementação para este algoritmo. Alguns trabalhos relacionados e considerações finais são apresentados nas seções 6 e 7, respectivamente.

2. Sistema de Quóruns Bizantinos e Modelo de Sistema

Os **sistemas de quóruns bizantinos** [Malkhi and Reiter, 1998a] são uma abordagem para a emulação de objetos de memória compartilhada em sistemas distribuídos onde os processos se comunicam por passagem de mensagens. O grande atrativo desta abordagem é o fato de que seus algoritmos não requerem a resolução do problema de consenso, não estando portanto sujeitos a impossibilidade FLP [Fischer et al., 1985].

Um sistema de quóruns para um universo de servidores de dados é um conjunto de vários conjuntos de servidores, chamados quóruns, que se intersectam. O princípio por trás de seu uso em serviços de armazenamento é que, se uma variável compartilhada é replicada entre todos esses servidores, as operações de leitura e escrita precisam ser feitas apenas em um dos quóruns destes servidores, e não em todo o sistema. A existência de intersecções entre os quóruns permite a construção de protocolos de leitura e de escrita que mantém a integridade da variável compartilhada mesmo que estas operações sejam realizadas em diferentes quóruns.

Formalmente, considera-se um universo U (finito) de servidores e um conjunto de clientes Π (infinito). O sistema de quóruns bizantinos é um conjunto de subconjuntos de servidores (quóruns) $\mathcal{Q} \subseteq 2^U$ onde todos os quóruns $Q \in \mathcal{Q}$ se intersectam em um número suficiente de servidores (propriedade de consistência) e sempre existe pelo menos um quórum onde não existem servidores faltosos (propriedade de disponibilidade) [Malkhi and Reiter, 1998a].

Tanto os processos clientes quanto os servidores estão sujeitos a faltas bizantinas [Lamport et al., 1982]: podem desviar arbitrariamente de sua especificação e podem, inclu-

sive trabalhar em conlúios maliciosos visando corromper os protocolos.

Os servidores implementam objetos de memória compartilhada (registrador e conjunto ativo) sendo organizados como um **sistema de quóruns de disseminação** que tolera até f faltas [Malkhi and Reiter, 1998a]. Devido as exigências deste tipo de sistema de quóruns, assumimos que $|U| \geq 3f + 1$ e que $\forall Q \in \mathcal{Q} : |Q| = 2f + 1$ (todo subconjunto de U com $2f + 1$ servidores é um quórum de $\mathcal{Q}$). A propriedade de consistência para este tipo de sistema de quóruns exige que $\forall Q_1, Q_2 \in \mathcal{Q} : |Q_1 \cap Q_2| \geq f + 1$. A propriedade de disponibilidade é garantida pelo fato de que os quóruns são sempre compostos por $2f + 1$ servidores dentro de um universo de $3f + 1$. Este tipo de sistema de quóruns é ótimo em termos do número de servidores necessários para prover emulação de memória em sistemas sujeitos a faltas bizantinas [Martin et al., 2002], entratanto, só pode ser usado no armazenamento de dados auto-verificáveis, i.e. qualquer tentativa de modificação dos dados armazenados no sistema por parte de servidores maliciosos pode ser detectada pelos clientes [Malkhi and Reiter, 1998a].

Todos os processos do sistema (clientes ou servidores) se comunicam através de canais ponto a ponto confiáveis FIFO. Assume-se adicionalmente que todos os processos têm um identificador único e que existe um esquema de assinaturas digitais não forjáveis [Rivest et al., 1978] que permite que todas as mensagens trocadas nos protocolos sejam autenticadas.

Os objetos de memória compartilhada (registrador e conjunto ativo) emulados pelos sistemas de quóruns bizantinos tornam disponíveis operações que não são executadas de forma atômica (indivisível) graças ao não determinismo das réplicas e as características do ambiente assíncrono. Desta forma, assumimos que uma operação tem início no momento em que o cliente envia sua requisição aos servidores e termina quando este mesmo cliente seleciona a sua resposta. Se uma operação o_2 começa durante a execução de uma outra o_1 dizemos que o_1 e o_2 são **concorrentes**. Para simplificar os algoritmos, assumimos que duas operações executadas por um mesmo processo **nunca** são concorrentes.

3. Objetos Básicos sobre Quóruns

O algoritmo do confeiteiro foi originalmente desenvolvido considerando memória compartilhada. Nesta seção apresentamos os protocolos necessários para implementar esta memória (no caso, objetos registradores) sobre sistemas de quóruns de disseminação. Além disso, esta seção também apresenta um objeto usado para manter o conjunto de processos executando o algoritmo de forma concorrente. Este objeto, conhecido como conjunto ativo, é a chave para o algoritmo suportar infinitos processos.

Visando melhorar sua legibilidade, os algoritmos apresentados nesta seção não levam em conta clientes faltosos. Na seção 3.3 são apresentadas modificações que permitem que os algoritmos para registradores e conjuntos ativos tolerem também faltas em clientes.

3.1. Registradores

Um **registrador** é um objeto de memória compartilhada que permite o armazenamento de dados. Este objeto suporta duas operações: $x.write(v)$, para escrita de um valor v no registrador x, e $x.read()$, que retorna o último valor escrito em x antes do início desta operação de leitura, desde que não haja a concorrência com escritas. Em caso de concorrência, o valor retornado depende do tipo do registrador [Lamport, 1986]. O algoritmo do confeiteiro

requer apenas registradores **seguros (*safe*)**, onde o valor retornado em caso de concorrência pode ser qualquer valor no domínio dos possíveis valores do registrador.

O conceito de registradores seguros [Lamport, 1986] permite a utilização de protocolos simples para sua implementação em sistemas de quóruns de disseminação. A seguir é apresentado um protocolo de leitura e escrita em um registrador para **um escritor** e **vários leitores** [Malkhi and Reiter, 1998a]. Como consideramos dados auto-verificáveis, o cliente (único) que vai escrever assina o valor a ser armazenado no registrador, e cada leitor, ao acessar o registrador para leitura, verifica os valores retornados pelos servidores e desconsidera os que apresentarem assinaturas inválidas.

- **Escrita:** Para um cliente escrever um valor v em x (operação $x.write(v)$), primeiro ele deve **assinar** v e depois escolher um *timestamp* t que seja maior que todos os *timestamps* usados por ele no passado. Em seguida ele difunde a operação de escrita $\langle v, t\rangle$ em um quórum $Q \in \mathcal{Q}$;
- **Leitura:** Para que um cliente leia o valor da variável x (operação $x.read()$) ele deve primeiro buscar um conjunto $A = \{\langle v_u, t_u\rangle\}_{u \in Q}$ em um quórum $Q \in \mathcal{Q}$. O par escolhido $\langle v, t\rangle \in A$ como o valor da leitura é o que apresentar o **maior** t com v **válido**.

Os protocolos descritos acima garantem a construção de um registrador **regular** [Malkhi and Reiter, 1998a] (mais forte que o **seguro**) onde o valor lido do registrador em caso de escrita concorrente é o valor antigo do registrador ou o valor sendo escrito [Lamport, 1986].

3.2. Conjunto Ativo

O **conjunto ativo** [Afek et al., 1999], proposto inicialmente no contexto dos algoritmos adaptativos[1], é um objeto de memória compartilhada que mantém o conjunto de processos participantes de uma computação distribuída. A noção de conjunto ativo é de grande valia na construção de algoritmos para infinitos processos.

Um conjunto ativo A suporta as seguintes operações: $A.join()$, que acrescenta o processo executor ao conjunto de processos ativos; $A.leave()$ que remove o processo executor do conjunto de processos ativos; e $A.getSet()$, que obtém a lista de processos (possivelmente) ativos. Formalmente, a execução de $A.getSet()$ retorna o conjunto dos processos que executaram $A.join()$ e não executaram $A.leave()$ antes do início da mesma se não ocorrer nenhuma operação de escrita ($A.join()$ ou $A.leave()$) concorrente a ela. Em caso de concorrência, o conjunto resultante pode ou não ter sido afetado pelas operações concorrentes.

Nossa implementação de conjunto ativo sobre um sistema de quóruns de disseminação é apresentada no algoritmo 1. As operações $join()$ e $leave()$ são triviais: uma requisição é enviada aos servidores que executam a operação em suas cópias locais do conjunto ativo e retornam uma confirmação ao cliente.

A implementação da operação $getSet()$ é bem mais complexa. O padrão de projeto *Listener* [Gamma et al., 1995] é usado nesta operação da mesma forma que no protocolo SBQ-L para registradores atômicos sobre quóruns bizantinos [Martin et al., 2002]. O

[1]Consideramos **algoritmos adaptativos** os algoritmos distribuídos que têm sua complexidade computacional em função da concorrência da execução [Afek et al., 1999, Aguilera, 2004].

Algoritmo 1 Conjunto Ativo (cliente c e servidor s).

```
1: {Parte Cliente}
   {entrada no conjunto}
2: procedure join()
3: ∀s ∈ U, send(s, ⟨ADD⟩)
4: wait until receive(⟨ACK-ADD⟩) de Q
   {saída do conjunto}
5: procedure leave()
6: ∀s ∈ U, send(s, ⟨REMOVE⟩)
7: wait until receive(⟨ACK-REMOVE⟩) de Q
   {leitura do conjunto}
8: procedure getSet()
9: ∀s ∈ U, send(s, ⟨READ⟩)
10: repeat
11:    wait until receive(s, ⟨DATA, S_s⟩)
12:    R_s ← R_s ∪ {S_s}
13: until ∃S : S aparece em pelo menos 2f + 1 Rs
14: ∀s ∈ U, send(s, ⟨READ-COMPLETE⟩)
15: return S
```

```
1: {Parte Servidor}
   {requisição de entrada no conjunto}
2: Require: receive(c, ⟨ADD⟩)
3: S ← S ∪ {c} {Alteração no conjunto.}
4: send(c, ⟨ACK-ADD⟩)
5: ∀c ∈ L, send(c, ⟨DATA, S⟩)
   {requisição de saída do conjunto}
6: Require: receive(c, ⟨REMOVE⟩)
7: S ← S \ {c} {Alteração no conjunto.}
8: send(c, ⟨ACK-REMOVE⟩)
9: ∀c ∈ L, send(c, ⟨DATA, S⟩)
   {requisição de leitura do conjunto}
10: Require: receive(c, ⟨READ⟩)
11: send(c, ⟨DATA, S⟩)
12: L ← L ∪ {c}
   {notificação de leitura completa}
13: Require:
   receive(c, ⟨READ-COMPLETE⟩)
14: L ← L \ {c}
```

uso do padrão *Listener* permite que um cliente executando $getSet()$ perceba as alterações concorrentes à sua leitura e possa inferir um conjunto ativo consistente a ser retornado, independente da ordem de recebimento das escritas (operações $join()$ e $leave()$) nos servidores. Um ponto negativo desta técnica é a necessidade do acesso a mais servidores e, de pelo menos uma difusão a mais no sistema durante a leitura (o envio da mensagem $\langle$READ-COMPLETE$\rangle$) [Martin et al., 2002]. A operação $getSet()$ começa com o cliente enviando uma requisição de leitura a todos os servidores (linha 9 do cliente). Cada servidor que recebe esta requisição retorna seu valor atual do conjunto S ao cliente e adiciona o cliente ao conjunto L (linhas 10-12 do servidor). A partir daí, durante cada alteração no conjunto ativo deste servidor, uma notificação é enviada aos clientes em L com o novo valor de S (linhas 5 e 9 do servidor). O cliente permanece coletando respostas de sua requisição de leitura até que exista uma resposta enviada por $2f + 1$ servidores (laço das linhas 10-13 no cliente). Após obter esta resposta, o cliente notifica aos servidores que sua leitura está completa para evitar que eles permaneçam enviando notificações ao mesmo (linhas 14 do cliente e 13-14 do servidor).

A correção e vivacidade do algoritmo 1, considerando um sistema de quóruns de disseminação, advém diretamente do fato de que, diferente dos protocolos para registradores (seção anterior), as requisições de escrita e leitura são difundidas em **todos** os servidores de U (linhas 3, 6, 9 e 14 no cliente). Este fato garante que $2f + 1$ servidores corretos vão acabar por receber estas requisições e respondê-las. A prova completa das propriedades do algoritmo se encontra na versão estendida deste artigo [Bessani et al., 2005].

3.3. Lidando com Clientes Faltosos

Os protocolos apresentados nas seções anteriores para registrador seguro e conjunto ativo não toleram atividades de clientes faltosos. Em particular, o grande problema com estes clientes advém da possibilidade deles difundirem escritas em conjuntos com um número

insuficiente de servidores: menos de $2f + 1$ para os registradores e $3f + 1$ para o conjunto ativo. Este tipo de operação pode levar o sistema a um estado inconsistente pois não existirá um quórum de leitura consistente de tal forma que o último valor escrito possa ser lido.

Para que este comportamento seja tolerado sem danos ao sistema, os servidores devem executar uma difusão confiável tolerante a faltas bizantinas [Bracha, 1984] a fim de disseminar cada requisição no sistema de quóruns. Desta forma, mesmo com o comportamento bizantino de infinitos clientes e com uma parcela dos servidores comprometidos (no máximo f) pode-se garantir que a operação recebida será consistente.

Um comportamento malicioso que poderia ser executado por um cliente contra o conjunto ativo seria a inclusão e remoção de processos arbitrariamente. Pelo próprio algoritmo e pela premissa assumida de que as mensagens são autenticadas, é facilmente verificável que esta forma de ataque não é passível de concretização em nossa proposta de conjunto ativo. A recepção das mensagens $\langle$ADD$\rangle$ e $\langle$REMOVE$\rangle$ pelos servidores causam a inclusão ou remoção apenas dos emissores destas do conjunto ativo local. Assim, um cliente faltoso só pode incluir-se ou retirar-se do conjunto ativo (linhas 2-3 e 6-7 do servidor).

Um outro comportamento bizantino do cliente a ser considerado na implementação do conjunto ativo é o não envio da notificação $\langle$READ-COMPLETE$\rangle$ (linha 14 da parte cliente do algoritmo 1) que visa fazer no fim da operação $getSet()$ que o servidor pare de desperdiçar recursos com este cliente. Apesar deste problema não ferir nem a correção (*safety*) e nem a vivacidade (*liveness*) dos protocolos, ele pode tornar os servidores extremamente carregados a medida em que o conjunto L se torna grande, provocando uma violação conhecida como negação de serviço (*Denial of Service* [Landwehr, 1981]). Os efeitos de ações como esta podem ser eliminados com a definição de um número máximo de notificações de alteração do conjunto ativo que um servidor pode enviar a cada leitor. Portanto, é fixado então que os servidores notificam a cada cliente em L no máximo T vezes; atingido este limite o cliente é então removido deste conjunto.

4. Exclusão Mútua

O problema da **exclusão mútua** [Dijkstra, 1965] consiste em gerenciar um conjunto de processos que desejam acessar um recurso indivisível que não pode sofrer acessos simultaneamente. Um algoritmo que resolve este problema deve garantir que processos nestas condições tenham acesso ao recurso um por vez. Os processos obtém acesso ao recurso exclusivo executando uma seção **crítica** de seu código. Os processos que estão tentando executar suas seções críticas são ditos na seção de **entrada** e os processos que já executaram suas seções críticas são ditos na seção de **saída**. Os algoritmos para exclusão mútua usualmente consideram apenas execuções **bem formadas** [Lynch, 1996]: todos os processos (inclusive os faltosos) executam as seções de entrada, crítica e de saída nesta ordem e podem eventualmente repetir esta sequência infinitas vezes.

Um algoritmo de exclusão mútua é muitas vezes modelado como um objeto de memória compartilhada onde os processos executam operações $enter()$ (seção de entrada) para tentar obter o acesso ao recurso, e $exit()$ para liberar o recurso (seção de saída). As operações $enter()$ e $exit()$ só retornam quando o processo é bem sucedido na obtenção e liberação do recurso, respectivamente. Neste trabalho seguimos esta modelagem, uma vez que a mesma é natural para uma implementação baseada em sistemas de quóruns e similar aos objetos de memória compartilhada introduzidos na seção 3 (registrador e conjunto

ativo).

Um algoritmo de exclusão mútua precisa satisfazer algumas propriedades [Lynch, 1996, Anderson et al., 2003]. Uma destas propriedades fundamenta a correção (*safety*) do algorithmo:

Exclusão Mútua: Não existe um estado alcançável do sistema onde dois processos corretos estão em suas seções críticas (executaram $enter()$ e não começaram a executar $exit()$);

Em termos de vivacidade (*liveness*), são duas as principais garantias que devem ser consideradas em algorítmos de exclusão mútua:

Progresso (*Deadlock-freedom*): Se um processo correto está na região de entrada (executando $enter()$), então **algum** processo correto acabará por executar sua região crítica (terminando a sua execução de $enter()$).

Progresso justo (*Starvation-freedom*): Se um processo correto está na região de entrada (executando $enter()$), então **este** processo acabará por executar sua região crítica (terminando a sua execução de $enter()$).

Note que a diferença entre a propriedade de progresso e a de progresso justo está no fato de que na primeira um processo pode ficar esperando por um recurso indefinidamente ("**algum** processo correto acaba por executar sua região crítica") enquanto que na segunda não ("**este** processo acaba por executar sua região crítica").

Quando consideramos o problema da exclusão mútua em sistemas com processos sujeitos a falhas bizantinas, algumas premissas devem ser assumidas. Conforme observado por Lynch [Lynch, 1996], o progresso de um algoritmo de exclusão mútua depende dos processos concorrentes saírem de sua região crítica (liberarem o recurso compartilhado). Como processos faltosos podem não liberar os recursos obtidos, faz-se necessária uma premissa extra no sistema para que processos faltosos possam ser tolerados [Rivest and Pratt, 1976, Katseff, 1978]. Esta premissa é justificada pelo uso de detectores de falhas byzantinas (seções 5.2 e 5.3).

Premissa 1 (Efeito transiente de falhas bizantinas) *As variáveis compartilhadas cujos processos faltosos podem escrever sempre voltam ao seu estado inicial.*

Uma outra premissa implícita em nosso algoritmo é o controle de acesso nos registradores compartilhados. Conforme será visto a seguir, cada registrador usado no algoritmo do confeiteiro bizantino só pode ser escrito por um único processo cliente (seu proprietário). A implementação desta premissa é feita facilmente a partir da implantação de listas de controle de acesso [Landwehr, 1981] localmente nos servidores com as réplicas dos registradores.

O algoritmo do confeiteiro [Lamport, 1974] resolve o problema da exclusão mútua satisfazendo progresso justo e necessitando apenas de registradores seguros para um escritor e vários leitores [Lamport, 1986]. Estas características fazem deste algoritmo uma boa escolha como ponto de partida para resolução de exclusão mútua em sistemas abertos[2] sujeitos a faltas bizantinas. O progresso justo permite que os processos corretos tenham garantias

[2]Consideramos como sistemas abertos, os sistemas em que os participantes não são previamente conhecidos. Os algoritmos distribuídos para estes sistemas geralmente suportam infinitos processos.

que suas tentativas de obter o recurso serão bem sucedidas enquanto o uso deste tipo simples de registrador permite a utilização direta de protocolos simples e eficientes baseados em sistemas de quóruns bizantinos [Malkhi and Reiter, 1998a] para prover uma memória compartilhada confiável.

Algoritmo 2 Algoritmo do Confeiteiro Bizantino (processo p).

```
 1: {Objetos compartilhados definidos com seus estados iniciais.}
 2: ActiveSet A ← ∅
 3: boolean chosing_p ← false, ∀p ∈ Π
 4: ℕ number_p ← 0, ∀p ∈ Π
    {Sincronização para entrar em seção crítica}
 5: procedure enter()
 6: A.join()
 7: chosing_p.write(true)
 8: S_p ← A.getSet()
 9: number_p.write(1 + max_{q∈S_p}(number_q.read()))
10: chosing_p.write(false)
11: S_p ← A.getSet()
12: for all q ∈ S_p \ {p} do
13:    wait until ¬chosing_q.read()
14:    wait until number_q.read() = 0 ∨ (number_p.read(), p) < (number_q.read(), q)
15: end for
    {Saída da seção crítica}
16: procedure exit()
17: number_p.write(0)
18: A.leave()
```

O algoritmo 2 apresenta o **confeiteiro bizantino** que faz uso de conjunto ativo e registradores implementados a partir de um sistema de quóruns bizantinos. Nas linhas 2-4, os seguintes objetos de memória compartilhada são declarados: um conjunto ativo A, onde os processos executando o algoritmo ficam registrados, e um par de registradores ($chosing$ e $number$) para cada processo que participa do algoritmo. Como citado anteriormente os processos donos destes registradores são os únicos que podem escrever nos mesmos.

No algoritmo, as operações $enter()$ e $exit()$ determinam as seções de entrada e saída, respectivamente. A operação $enter()$ começa com o processo se juntando ao conjunto ativo (linha 6) para então definir o valor de seu registrador $number$ com um valor maior que todos os outros já definidos pelos processos pertencentes ao conjunto ativo (linhas 8-9). Enquanto esta definição é feita, o registrador $chosing$ do processo é marcado para avisar aos demais que este processo está escolhendo seu *ticket* (linhas 7 e 10). Tendo definido seu *ticket* ($number$), o processo obtém uma nova estimativa dos processos ativos no momento (linha 11) e então espera até que todos estes processos *(i)* não estejam em escolha (linha 13) e *(ii)* que tenham *ticket* com valor maior que o seu[3] (linha 14). Verificadas as condições *(i)* e *(ii)*, o processo passa a executar sua seção crítica. A operação $exit()$ é trivial: apenas zera o registrador $number$ do processo e remove o mesmo do conjunto ativo.

Note que o algoritmo é adaptativo: o número de acessos a memória compartilhada

[3]Como podem ocorrer execuções concorrentes da linha 9, o identificador do processo é usado para resolver os casos de empate (por isso a comparação literal de $(number_p, p)$).

(sistema de quóruns bizantinos) é proporcional ao número de processos tentando executar o algoritmo concorrentemente [Afek et al., 1999]. Assim, sua complexidade de passos é $O(k)$, sendo k a quantidade de processos o executando concorrentemente (ponto de contenção). A prova de correção e vivacidade deste algoritmo pode ser encontrada na versão estendida deste artigo [Bessani et al., 2005].

5. Implementando Exclusão Mútua em Quóruns

A figura 1 apresenta uma arquitetura de sistema distribuído (clientes e servidores) para implementação do algoritmo 2 em um sistema de quóruns bizantinos.

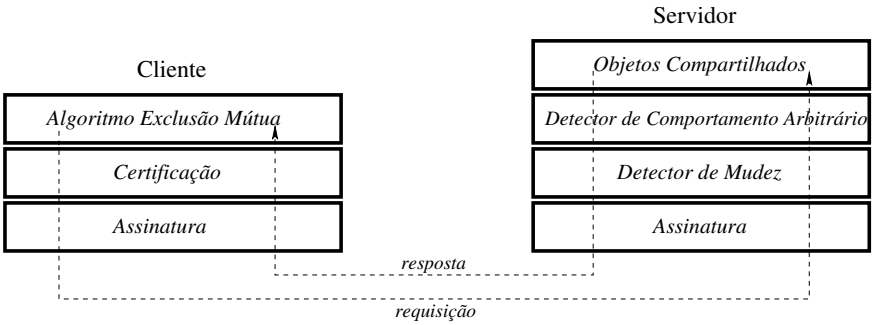

Figura 1: Arquitetura dos clientes e servidores.

Um módulo básico presente na figura, usado tanto por clientes quanto por servidores, é o modulo de assinatura. Este módulo permite a comunicação autenticada entre os processos do sistema. Em um nível mais alto das pilhas de protocolos temos os algoritmos propostos e as estruturas de dados usadas em cada servidor. Nesta figura é possível ainda observar três módulos usados para detecção de falhas: detector de mudez e detector de comportamento bizantino no servidor e certificação no cliente. Estes módulos justificam a premissa 1 e são descritos detalhadamente nesta seção. Antes porém, é apresentada uma otimização do algoritmo 2.

5.1. Um Protocolo Eficiente

A implementação direta do algoritmo 2 usando os objetos descritos na seção 3 dá origem a um protocolo que requer um número grande de mensagens trocadas entre o cliente e os servidores. Para um cliente tentando obter um recurso sem concorrência (conjunto ativo inicialmente vazio) são necessários 6 acessos ao sistema de quóruns, o que acarreta em um número de mensagens impraticável.

Felizmente este número pode ser drasticamente diminuído se agruparmos mensagens não dependentes em um único acesso ao quórum. Desta forma, podemos otimizar a seção de entrada para apenas 2 acessos em casos sem concorrência e 3 acessos no caso geral. Este agrupamento torna a implementação do procedimento $enter()$ (pelo cliente c) realizável através de apenas três acessos ao sistema, a saber:

1. **Obtenção de** $number_q, \forall q \in S$ **(GET-TICKET):** Este acesso corresponde à requisição das operações das linhas 6-8 mais a leitura dos registradores $number$ usados para definição de $number_c$ na linha 9;
2. **Definição de** $number_c$ **(SET-TICKET):** Este acesso corresponde a execução das linhas 9-11. O cliente define seu $number$ no quórum e obtém o conjunto ativo;

3. **Espera sua vez (LISTEN):** Neste acesso, executado apenas quando o conjunto ativo contém outros processos que não o executor do protocolo, o cliente avisa aos servidores que condições ele espera para obter o recurso. Na prática isto implica na espera que as condições das linhas 13-14 do algoritmo 2 se verifiquem, considerando as variáveis compartilhadas.

O procedimento $exit()$ é executado em apenas um acesso ao sistema de quóruns através da mensagem de liberação (EXIT) que equivale à execução das linhas 17 e 18 do algoritmo 2. A figura 2 ilustra uma execução do protocolo apresentado.

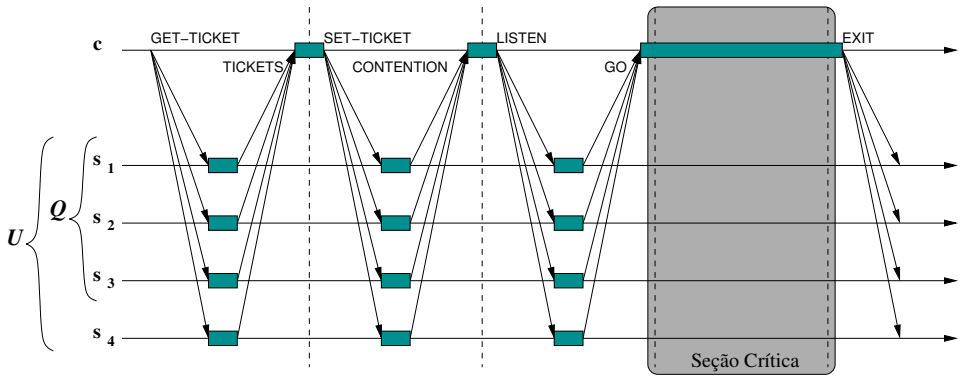

Figura 2: Execução do confeiteiro bizantino.

Todos os acessos ao sistema de quóruns feitos durante a execução dos procedimentos $enter()$ e $exit()$ requerem a difusão das mensagens em todos os servidores do sistema, logo a complexidade de mensagens do protocolo é $O(k|U|)$ (k é a concorrência da execução).

Note que as seqüências de operações requisitadas nestas mensagens só são equivalentes as operações definidas no algoritmo 2 se forem computadas na mesma ordem deste algoritmo tanto no servidor quanto no cliente. Esta equivalência pode ser facilmente verificada a partir da equivalência dos estados resultantes do sistema se executarmos cada operação individualmente e agrupadas conforme proposto.

5.2. Detector de Mudez

Os **detectores de mudez** [Doudou et al., 1999] são uma abordagem intermediária entre a detecção de falhas bizantinas completa e a detecção apenas de falhas de parada. O objetivo destes oráculos locais é detectar processos que não enviam mensagens regularmente ou não cumprem os requisitos de sintaxe (formatos das mensagens) impostos pelo algoritmo. Este comportamento é caracterizado como **falha por mudez**. Ao contrário das paradas convencionais (falhas por parada), a semântica de falha por mudez não indica propriamente a parada na execução de um processo, mas sim que este parou de enviar mensagens de acordo com o algoritmo ou protocolo que deveria executar, ou seja, a falha por mudez captura “paradas algorítmicas”.

Dado um algoritmo distribuído $\mathcal{A}$, este algoritmo gera um conjunto de mensagens que seguem uma sintaxe específica, sendo chamadas de mensagens-$\mathcal{A}$. A partir deste conceito podemos definir um **processo mudo**: um processo q é mudo, com respeito a um algoritmo $\mathcal{A}$ e um processo p, se q para prematuramente de mandar mensagens-$\mathcal{A}$ para p.

A classe de detectores de falhas por mudez, ou detectores de mudez, é definida através das propriedades de **completude** e **precisão** de forma análoga aos detectores de

falhas convencionais (para faltas de parada) [Chandra and Toueg, 1996]. Neste trabalho utilizamos um detector de mudez não confiável da classe $\Diamond\mathcal{PM}_{\mathcal{ME}}$, que é o equivalente à classe $\Diamond\mathcal{P}$ para falhas de parada [Chandra and Toueg, 1996]. A característica fundamental deste detector é o fato de que em algum momento da execução do algoritmo ele para de cometer erros.

Na arquitetura proposta, os detectores desta classe são usados pelos servidores para identificar clientes faltosos da seguinte forma: toda requisição que chega a um servidor passa pelo módulo detector de mudez (figura 1), este módulo atualiza sua lista de processos suspeitos e repassa a requisição para as camadas superiores. Periodicamente, cada servidor verifica sua lista de suspeitos e, para cada cliente c presente nesta lista, atribui valores padrões para as variáveis de controle correspondentes ($chosing_c \leftarrow false$ e $number_c \leftarrow 0$), remove c de sua cópia local do conjunto ativo ($S \leftarrow S \setminus \{c\}$) e envia uma mensagem avisando a c de que o mesmo não está mais disputando o recurso. O cliente não consegue mais executar o algoritmo se é removido da execução por mais de f servidores.

Note que pelo fato do detector de mudez ser não confiável, clientes corretos podem acabar por ser removidos da execução algoritmo. Entretanto, pela propriedade de precisão do detector, estes clientes acabarão por conseguir executar o algoritmo completamente.

5.3. Detecção de Clientes Maliciosos e Certificação

Mesmo com os mecanismos que protegem os registradores e o conjunto ativo de comportamentos maliciosos (seção 3.3) e com os detectores de mudez nos servidores (seção anterior), ainda existe a possibilidade de clientes maliciosos atrapalharem o progresso do protocolo de exclusão mútua. Bastaria para tanto que estes clientes maliciosos não executassem alguns passos como especificado no algoritmo 2.

Para lidar com este tipo de problema, cada servidor usa um **detector de comportamento malicioso** (DCM) [Baldoni et al., 2003] para estimar qual o estado de um cliente e verificar se suas mensagens são devidamente justificadas. A premissa de ordem FIFO nos canais de comunicação possibilita o uso de autômatos nos servidores para a estimativa do estado de um cliente já que a ordem de recepção das mensagens pelos servidores corresponde a ordem do envio pelo cliente. A implementação deste detector se baseia em uma máquina de estados para cada cliente presente no conjunto ativo. Denotamos por $\mathcal{A}^{s,c}_{\mathcal{ME}}$ o autômato executado pelo servidor s para estimar o estado de um cliente c com relação ao algoritmo da seção 5.1. Toda vez que c envia uma mensagem ao servidor s, o DCM de s atualiza o estado do autômato. Uma descrição deste autômato é apresentado na figura 3.

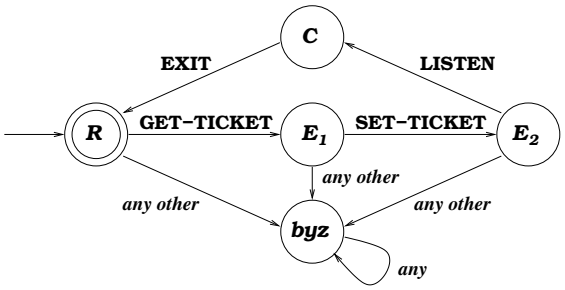

Figura 3: Autômato $\mathcal{A}^{s,c}_{\mathcal{ME}}$.

Nesta figura temos representada a função de transição de estados do autômato $\mathcal{A}^{s,c}_{\mathcal{ME}}$.

O alfabeto que dispara as transições é composto basicamente do conjunto de possíveis mensagens bem formadas[4] que um cliente pode enviar para um servidor. Uma descrição mais precisa dos estados e das transições é dada a seguir:

- **R:** Este é o estado inicial dos clientes. Um cliente c é considerado neste estado se não está executando o algoritmo de exclusão mútua e, portanto, não possui uma instância de $\mathcal{A}^{s,c}_{\mathcal{ME}}$ no DCM do servidor s do sistema de quórum. A única mensagem legítima que um cliente neste estado pode enviar aos servidores corretos é uma GET-TICKET requisitando os tickets dos demais clientes disputando o recurso;
- $\mathbf{E_1}$**:** O cliente obteve os *tickets* dos demais clientes e agora vai definir seu *ticket* através de uma mensagem SET-TICKET. Este é um passo crítico do algoritmo e o DCM deve garantir que o valor definido seja maior que o valor dos demais *tickets* dos clientes no conjunto ativo. Para verificar este valor o servidor não pode confiar em suas cópias locais dos dados compartilhados, pois ele pode não ter ainda recebido alguma atualização e portanto conter valores antigos (o que é aceitável pelas características de sistema assíncrono). Uma solução possível para este problema consiste em que cada cliente envie, juntamente com sua requisição SET-TICKET, o conjunto de respostas recebidas para a requisição GET-TICKET. Este conjunto de respostas autenticadas justifica o valor escolhido pelo cliente para seu registrador $number$ (linha 9 do algoritmo 2). A fim de evitar que clientes maliciosos utilizem justificativas antigas favoráveis repetidamente, cada servidor aceita cada justificativa apenas uma vez;
- $\mathbf{E_2}$**:** Tendo o cliente definido seu *ticket* só lhe resta esperar que os clientes com número de *ticket* menor sejam atendidos. Para dar início a este processo de espera o cliente envia uma mensagem LISTEN e espera por notificações de $2f+1$ servidores;
- **C:** Este estado corresponde a região crítica, onde o cliente tem o recurso e pode utilizá-lo livremente. Um cliente só sai desse estado se enviar a mensagem $EXIT$;
- **byz:** Qualquer outra ação que não as especificadas anteriormente leva o autômato permanentemente a este estado. Mensagens de clientes cujo autômato correspondente está no estado **byz** são ignoradas.

Note que, ao contrário do detector de mudez, o DCM é **confiável** e portanto não comete erros. Outro ponto a ser notado a respeito do DCM é que, apesar de sua aparente complexidade, ele pode ser implementado através de uma tabela nos servidores. Esta tabela conteria o estado atual do autômato correspondente a cada cliente pertencente à cópia local do conjunto ativo.

O módulo de certificação no cliente é usado para coletar respostas advindas dos servidores e construir justificativas (também chamadas **certificados**) [Bracha, 1984] a serem adicionadas às requisições seguintes. O tamanho destas justificativas é sempre linearmente proporcional à quantidade de processos no conjunto ativo e ao tamanho do quórum.

6. Trabalhos Relacionados

O algoritmo do confeiteiro bizantino é uma versão adaptada do algoritmo do confeiteiro para infinitos processos apresentado em [Aguilera, 2004]. Entretanto, duas características fundamentais distinguem o confeiteiro bizantino em relação a este. Em primeiro lugar o

[4]Uma mensagem bem formada é uma mensagem cuja sintaxe segue a especificação do protocolo, e portanto não é considerada inválida pelo detector de mudez.

algoritmo de [Aguilera, 2004] utiliza um objeto de filiação, mais forte (complexo e custoso) que o conjunto ativo usados em nosso algoritmo. A segunda diferença diz respeito ao fato do nosso algoritmo tolerar faltas bizantinas assumindo e justificando a premissa 1 enquanto o algoritmo de [Aguilera, 2004] não tolera falhas.

A literatura sobre algoritmos de exclusão mútua é extremamente vasta, tanto em modelos de memória compartilhada como em passagem de mensagens [Raynal, 1986, Anderson et al., 2003]. Porém dentre estes algoritmos, poucos são tolerantes a faltas [Rivest and Pratt, 1976, Katseff, 1978] e, até onde sabemos, nenhum considera explicitamente a possibilidade de faltas bizantinas. Nossa percepção é que existe um consenso "não declarado" de que este problema não faz sentido no modelo de faltas bizantinas devido ao fato das propriedades de progresso e exclusão mútua dependerem da correta execução do algoritmo por parte dos processos. Conforme apresentado neste trabalho, com a utilização de detectores de faltas bizantinas, da mesma forma que em [Baldoni et al., 2003] para o problema de consenso, é possível resolver o problema de exclusão mútua desde que se implemente a memória compartilhada em um sistema de quóruns bizantinos.

O único sistema que provê objetos de exclusão mútua para processos sujeitos a falhas bizantinas considerando um modelo assíncrono é o Fleet [Malkhi and Reiter, 2000]. Este sistema disponibiliza objetos de exclusão mútua com a semântica "no máximo um" [Malkhi and Reiter, 1998b] onde, em caso de concorrência, ninguém obtém o recurso. A semântica "estritamente um" que é objetivo de qualquer algoritmo de exclusão mútua é implementada neste sistema sobre um algoritmo de consenso aleatório para memória compartilhada (emulada em um sistema de quóruns de mascaramento [Malkhi and Reiter, 1998a], que requer $|U| \geq 4f + 1$). Em relação ao algoritmo do confeiteiro bizantino, esta solução requer mais servidores, é bastante custosa e não garante nem terminação determinista (já que o algoritmo é aleatório) e nem progresso justo.

7. Conclusão

Este trabalho apresentou o algoritmo do confeiteiro bizantino para exclusão mútua em sistemas abertos não confiáveis. Este algoritmo é a primeira proposta na literatura a garantir progresso justo em um cenário assíncrono com infinitos processos sujeitos a faltas bizantinas. Nosso algoritmo é implementado sobre uma memória compartilhada confiável emulada em um sistema de quóruns bizantino de disseminação e garante exclusão mútua para infinitos processos (clientes) bizantinos desde que menos de $1/3$ dos servidores do sistema de quóruns sejam faltosos.

Além do algoritmo do confeiteiro bizantino em si, este artigo apresentou outras contribuições que são interessantes por elas próprias: conjunto ativo tolerante a faltas bizantinas implementado sobre um sistema de quóruns, detector de comportamento malicioso para o algoritmo do confeiteiro e um esquema para a implementação eficiente de algoritmos baseados em sistemas de quóruns bizantinos baseado no agrupamento de mensagens.

Referências

Afek, Y., Stupp, G., and Touitou, D. (1999). Long-lived adaptive collect with applications. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science - FOCS'99*, pages 262–272.

Aguilera, M. (2004). A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59.

Anderson, J. H., Kim, Y.-J., and Herman, T. (2003). Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(1):75–110.

Baldoni, R., Hélary, J.-M., Raynal, M., and Tangui, L. (2003). Consensus in byzantine asynchronous systems. *Journal of Discrete Algorithms*, 1(2):185–210.

Bessani, A. N., da Silva Fraga, J., and Lung, L. C. (2005). O confeiteiro bizantino: Exclusão mútua em sistemas abertos sujeitos a faltas bizantinas. http://www.das.ufsc.br/∼neves/reports/2005-1.pdf.

Bracha, G. (1984). An asynchronous [(n - 1)/3]-resilient consensus protocol. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 154–162. ACM Press.

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2).

Dijkstra, E. W. (1965). Solution of a problem in concurrent program control. *Comm. of the ACM*, 8(9):569.

Doudou, A., Garbinato, B., Guerraoui, R., and Schiper, A. (1999). Muteness failure detectors: Specification and implementation. In *Proceedings of the 3rd European Dependable Computing Conference*. Springer-Verlag.

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

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading.

Katseff, H. P. (1978). A new solution to critical section problem. In *Proceedings of the 10th ACM Symposium on Theory of Computing*, pages 86–88.

Lamport, L. (1974). A new solution for Dijkstra's concurrent programing problem. *Communications of the ACM*, 17(8):453–455.

Lamport, L. (1986). On interprocess communication (part ii: algorithms). *Distributed Computing*, 1(1):203–213.

Lamport, L., Shostak, R., and Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401.

Landwehr, C. E. (1981). Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278.

Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kauffman, San Francisco - California.

Malkhi, D. and Reiter, M. (1998a). Byzantine quorum systems. *Distributed Computing*, 11(4):203–213.

Malkhi, D. and Reiter, M. (1998b). Secure and scalable replication in phalanx (extended abstract). In *Proceedings of 17th Symposium on Reliable Distributed Systems*, pages 51–60. IEEE Computer Society.

Malkhi, D. and Reiter, M. K. (2000). An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202.

Martin, J.-P., Alvisi, L., and Dahlin, M. (2002). Minimal Byzantine storage. In *Distributed Computing, 16th international Conference, DISC 2002*, volume 2508 of *LNCS*, pages 311–325. Springer-Verlag.

Raynal, M. (1986). *Algorithms for Mutual Exclusion*. MIT Press, Cambridge - Massachussets.

Rivest, R. L. and Pratt, V. R. (1976). The mutual exclusion problem for unreliable processes: preliminary report. In *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, pages 1–8.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.