Coordenação Desacoplada Tolerante 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. *Existing distributed systems require communication mechanisms that satisfy requirements as anonymity and temporary disconnection. In this context, generative communication is becoming one of the coordination models capable to address these requirements once it is decoupled in time and space. This work presents the first proposal in the literature to consider the development of a byzantine fault tolerant generative coordination infrastructure. This construction is based on the byzantine quorum systems replication technique.*

Resumo. *Os sistemas distribuídos atuais requerem mecanismos de comunicação que atendam requisitos como anonimato e desconexão temporária. Neste contexto, a comunicação generativa vêm se afirmando como um dos modelos de coordenação capazes de atender esses requisitos uma vez que é desacoplada no tempo e no espaço. Este trabalho apresenta a primeira proposta da literatura a considerar a construção de uma infra-estrutura de coordenação generativa tolerante a faltas bizantinos. Esta construção se dá através da aplicação de replicação por sistemas de quóruns bizantinos.*

1. Introdução

O crescimento do uso dos computadores e da Internet em todas as áreas de conhecimento estimula o desenvolvimento de novas tecnologias e aplicações distribuídas. Os participantes destas novas aplicações são caracterizados pela heterogeneidade, capacidades de comunicação variável e por serem pouco confiáveis. Em um futuro próximo, a necessidade de sistemas para suporte a aplicações distribuídas em larga escala se tornará comum, e para tratar deste potencial problema, novos modelos e paradigmas de computação (p.ex. redes par a par, serviços web e computação em grade) estão sendo propostos. Em muitos aspectos, estes modelos/paradigmas se baseiam no conceito de sistemas abertos, que se caracterizam principalmente por serem compostos por um número desconhecido de processos heterogêneos que participam das interações em diferentes momentos.

Os mecanismos de coordenação [Gelernter and Carriero, 1992] usualmente empregados para a interação em sistemas distribuídos, como a comunicação por passagem de mensagens, não são adequados para os sistemas abertos do futuro. O requisito de anonimato e as

[*]Doutorando Bolsista PGI/CNPq.
[†]Bolsista PQ/CNPq.
[‡]Bolsista PQ/CNPq.

comunicações pouco confiáveis implicam diretamente na necessidade de interações desacopladas nestes sistemas. Assim, modelos de coordenação alternativos se fazem necessários. Dentre estes modelos, a comunicação generativa [Gelernter, 1985] se destaca pela sua flexibilidade e simplicidade, sendo utilizada em uma série de infra-estruturas de comunicação como o JAVASPACES [Sun Microsystems, 2003] e o TSPACES [Lehman et al., 2001]. Neste modelo, os processos interagem através de um espaço de memória compartilhado (espaço de tuplas) em que estruturas de dados genéricas (tuplas) são colocadas, lidas e coletadas durante as interações. A coordenação ocorre de maneira desacoplada no tempo (os participantes não precisam estar engajados ao mesmo tempo) e no espaço (não precisam conhecer uns aos outros) [Cabri et al., 2000]. Além disso, o número reduzido de operadores e sua generalidade implicam em uma grande simplicidade (em termos de programação) na implementação de sistemas distribuídos [Carriero and Gelernter, 1989, Sun Microsystems, 2003].

Independentemente do modelo de coordenação empregado nos sistemas distribuídos abertos, estas interações estão sujeitas a todos os tipos de falhas e ataques de segurança. Estes eventos podem ser agrupados e modelados como faltas bizantinas [Lamport et al., 1982], para que técnicas de tolerância a faltas possam ser empregadas visando melhorar a confiabilidade da infra-estrutura da coordenação. A aplicação destas técnicas permite que uma infra-estrutura de coordenação para aplicações críticas justifique a confiança depositada neste componente fundamental em sistemas abertos.

Este trabalho apresenta a primeira proposta de modelo de coordenação baseado em espaço de tuplas tolerante a faltas bizantinas. Esta proposta se baseia na emulação do espaço de tuplas sobre um sistema distribuído, sem memória compartilhada, onde os processos se comunicam por passagem de mensagens. A técnica utilizada para esta emulação são os sistemas de quóruns bizantinos [Malkhi and Reiter, 1998, Martin et al., 2001].

O desenvolvimento desta infra-estrutura de coordenação objetiva atacar o problema da tolerância a intrusões [Fraga and Powell, 1985, Veríssimo et al., 2003] em sistemas abertos. Para tanto, o ambiente considerado é bastante desfavorável: número desconhecido de processos sujeitos a faltas maliciosas[1] executando em um ambiente completamente assíncrono.

Este artigo está estruturado da seguinte forma: a seção 2 apresenta uma breve introdução ao modelo de coordenação generativa. A seção 3 apresenta as premissas de nosso modelo de sistema, o tipo de sistema de quóruns utilizado e os protocolos que implementam o espaço de tuplas tolerante a faltas bizantinas. A seção 4 apresenta um estudo a respeito da complexidade e dos custos dos protocolos desenvolvidos. Finalmente, as seções 5 e 6 apresentam alguns trabalhos relacionados presentes na literatura e as considerações finais deste artigo, respectivamente.

2. Coordenação Generativa

O modelo de **coordenação generativa** foi introduzida no contexto da linguagem de programação para sistemas paralelos LINDA [Gelernter, 1985]. O mecanismo de coordenação definido nesta linguagem permite aos processos distribuídos interagirem sobre um espaço de memória compartilhado em que estruturas de dados genéricas (tuplas) são adicionadas, lidas e removidas. A comunicação é chamada generativa devido ao fato de que uma vez criadas no espaço, as tuplas têm existência completamente separada dos processos

[1] Processos que utilizam o espaço de tuplas para coordenação.

que as geraram.

No modelo generativo, uma tupla $t = \langle f_1, f_2, ..., f_m \rangle$ é um conjunto de campos ordenados onde cada campo f_i está associado a um tipo (inteiro, lógico, *string*, etc...) e pode ter um valor definido que deve estar no domínio deste tipo. Uma tupla em que todos os campos têm valores definidos (campos atuais) é chamada **entrada** (*Entry*). Um **molde** (*template*), denotado por $\bar{t}$, é uma tupla que pode conter campos sem valor definido (campo formal).

O **tipo de um campo**, seja ele atual ou formal, é dado pela função $\tau : \mathcal{F} \rightarrow \mathcal{T}$, onde $\mathcal{F}$ é o conjunto de todos os possíveis campos, atuais ou formais, e $\mathcal{T}$ é o conjunto dos possíveis tipos de dados assumidos no modelo de computação usado. O **tipo de uma tupla** t, denotado por $type(t)$, é a sequência dos tipos dos campos de t. Diz-se que duas tuplas t e t' são do mesmo tipo ($type(t) = type(t')$) se os campos destas tupla são do mesmo tipo. Em princípio, duas tuplas combinam se elas são do mesmo tipo e têm valores compatíveis.

As manipulações realizadas no espaço de tuplas consistem de invocações de três operações básicas [Gelernter, 1985][2]:

- $out(t)$: Esta operação adiciona a tupla t no espaço de tuplas. A operação out é chamada usualmente de operação de escrita no espaço;
- $in(\bar{t})$: Esta operação retira do espaço de tuplas uma tupla que combine com o molde $\bar{t}$. Caso não exista nenhuma tupla que combine com $\bar{t}$ no espaço o processo fica parado esperando até que exista uma (operação bloqueante). A operação in é usualmente chamada de leitura destrutiva, remoção ou coleta;
- $rd(\bar{t})$: Operação que lê no espaço de tuplas uma tupla que combine com o molde $\bar{t}$. Caso não exista nenhuma tupla que combine com $\bar{t}$ no espaço, o processo fica parado esperando até que exista uma (operação bloqueante). A grande diferença desta operação para in é o fato dela não remover a tupla lida do espaço de tuplas.

São também definidas duas variantes das operações in e rd não bloqueantes, estas operações são chamadas inp e rdp e seus funcionamentos são exatamente iguais ao das operações originais, a não ser pelo fato de que elas sempre retornam um valor lógico: se não houver uma tupla no espaço que combine com o molde passado, estas operações retornam um valor de falha $false$, caso contrário, um valor $true$ é retornado juntamente com a tupla lida.

Uma característica muito importante do modelo generativo e das operações definidas para o espaço de tuplas é seu não determinismo inerente. Se várias tuplas existentes no espaço combinam com um molde passado como parâmetro em uma operação in ou rd, qualquer uma delas pode ser retornada. Da mesma forma, se dois ou mais processos estiverem parados esperando por tuplas com determinadas características (definidas nos moldes apresentados como argumentos nas operações) e uma entrada é inserida no espaço de tal forma que ela combina com os moldes de qualquer um dos processos esperando a tupla, qualquer um deles pode recebê-la, permanecendo os demais em estado de espera.

Outra característica fundamental da comunicação generativa é o acesso associativo a tuplas: os dados são acessados a partir de seu conteúdo, e não através de seu endereço. A idéia é que o espaço de tuplas é uma sacola onde ítens de dados (tuplas) são colocados, lidos e retirados de acordo com suas características (conteúdo). Estes dados, uma vez no espaço de

[2]Existem diversas definições da semântica de um espaço de tuplas, todas ligeiramente diferentes. Neste trabalho estamos assumindo a definição original adaptada para sistemas utilizados atualmente como o JAVASPACES [Sun Microsystems, 2003] e o TSPACES [Lehman et al., 2001].

tuplas, não podem ser alterados. Desta forma, para se alterar uma tupla do espaço, é preciso removê-la, e criar outra tupla no espaço com os valores da antiga, porém alterada. Esta característica de memória associativa diferencia este modelo de coordenação dos demais modelos baseados em memória compartilhada. A associatividade da memória, implementada através de um esquema de nomeação estruturado [Gelernter, 1985] parecido com a operação *select* dos bancos de dados relacionais, permite que as operações in e rd acessem as tuplas do espaço através de seus valores e não de seus endereços. Fundamental para este mecanismo é o conceito de **combinação de tuplas**. Diz-se que duas tuplas (com o mesmo número de campos), uma entrada $t = \langle f_1, f_2, ..., f_m \rangle$ e um molde $\overline{t} = \langle \overline{f}_1, \overline{f}_2, ..., \overline{f}_m \rangle$, combinam[3] denotada por $m(t, \overline{t})$, se e somente se $\forall i = 1..m : \tau(f_i) = \tau(\overline{f}_i) \wedge (\overline{f}_i = \perp \vee f_i = \overline{f}_i)$. Esta condição define que a combinação existe se para todo o campo da tupla, o campo correspondente do molde é do mesmo tipo e tem o mesmo valor ou tem valor indefinido ($\perp$). A definição de combinação de tupla implica diretamente no fato de que $\forall t, \overline{t} : m(t, \overline{t}) \rightarrow (type(t) = type(\overline{t}))$.

3. Espaço de Tuplas Tolerante a Faltas Bizantinas

O ponto central do trabalho aqui apresentado é a provisão de um espaço de tuplas tolerante a faltas bizantinas. A replicação é a técnica chave nesse sentido, e a coordenação das réplicas do espaço (para manutenção de sua consistência) é o principal problema a ser resolvido.

3.1. Modelo de Sistema

O modelo de sistema adotado consiste de um conjunto arbitrário de processos clientes que se comunicam com servidores que emulam um espaço de tuplas. Todas as computações e comunicações ocorrem de forma **assíncrona** [Fischer et al., 1985]: não existem limites de tempo para sua terminação.

Em termos de falhas de processos, assumimos que um número arbitrário de clientes e um limite máximo de f servidores estão sujeitos a **falhas bizantinas**: podem desviar arbitrariamente de sua especificação e podem, inclusive, trabalhar em conluios maliciosos visando corromper o sistema. Assumimos ainda **independência de faltas**, obtida através da utilização de diferentes plataformas (hardware, SO, VM, etc) [Castro et al., 2003].

Todos os processos do sistema (clientes ou servidores) se comunicam através de canais ponto a ponto confiáveis com ordenação FIFO. Assume-se também que cada processo tem 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.

O espaço de tuplas emulado sobre um sistemas de quóruns bizantinos torna disponíveis as operações definidas na seção 2, porém estas não são executadas de forma atômica (indivisível) devido ao não determinismo das réplicas e as características do ambiente assíncrono. Assim, toda operação o tem um início e um fim (denotados $begin(o)$ e $end(o)$), que são os eventos que marcam a invocação da operação o e a definição de sua resposta no cliente, respectivamente. É possível montar um ordem total destes eventos em um sistema se tomarmos os instantes de tempo real que eles são executados. A esta sequência de inícios e fins de operações damos o nome de **execução**.

[3] A regra de combinação apresentada neste texto são adaptações livres das apresentadas em [Busi et al., 2003], visando torná-las compatíveis com a literatura corrente.

A partir desta ordem total, podemos definir, para quaisquer dois eventos e_1 e e_2, qual acontece antes de outro (denotado por $\rightarrow$). Por exemplo, para toda operação o, $begin(o) \rightarrow end(o)$. Dadas duas operações o_1 e o_2, dizemos que o_1 **precede** o_2 se $end(o_1) \rightarrow begin(o_2)$. Se $end(o_1) \nrightarrow begin(o_2)$ e $end(o_2) \nrightarrow begin(o_1)$, então o_1 e o_2 são ditas **concorrentes**. Para simplificar os algoritmos, assumimos que duas operações executadas por um mesmo processo **nunca** são concorrentes.

3.2. Sistemas de Quóruns Bizantinos

Os **sistemas de quóruns bizantinos** [Malkhi and Reiter, 1998] são uma alternativa 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 sub-conjuntos de servidores, chamados quóruns, que possuem intersecção entre si. 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 n servidores ($|U| = n$) e um conjunto arbitrário de clientes Π. O sistema de quóruns bizantinos é um conjunto de sub-conjuntos de servidores (quóruns) $\mathcal{Q} \subseteq 2^U$ onde todos os quóruns $Q \in \mathcal{Q}$ têm em suas intersecção 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, 1998].

Os servidores implementam objetos de memória compartilhada sendo organizados como um sistema de quóruns de mascaramento que tolera até f faltas [Malkhi and Reiter, 1998]. Em particular, utilizamos um **sistema de quóruns de mascaramento assimétrico** [Martin et al., 2001], que se distingue dos demais devido ao fato de usar diferentes tamanhos de quóruns em diferentes operações.

Devido as exigências do sistemas de quóruns assimétricos, assumimos que $n \geq 3f + 1$ e que os dois tipos de quóruns presentes no sistema, de leitura (Q_r) e escrita (Q_w), contém $\lceil \frac{n+f+1}{2} \rceil$ e $\lceil \frac{n+f+1}{2} \rceil + f$ servidores, respectivamente. A figura 1 apresenta um exemplo de sistema de quóruns com $n = 4$ e $f = 1$.

Nesta figura temos representados o quórum de escrita $Q_w = \{s_1, s_2, s_3, s_4\}$ e um quórum de leitura[4] $Q_r = \{s_2, s_3, s_4\}$. Dois clientes, c_1 e c_2, interagem com o sistema de quóruns chamando operações de escrita e leitura nos quóruns Q_w e Q_r, respectivamente.

3.3. Protocolo

Nesta seção apresentamos os algoritmos para as operações definidas na comunicação generativa considerando um espaço de tuplas replicado em um sistema de quóruns bizantinos. Todos os algoritmos definidos consideram que cada servidor s do sistema tem uma cópia do espaço de tuplas, chamado espaço local e denotado por T_s.

[4]Todo conjunto de 3 servidores neste sistema é um quórum de leitura neste sistema.

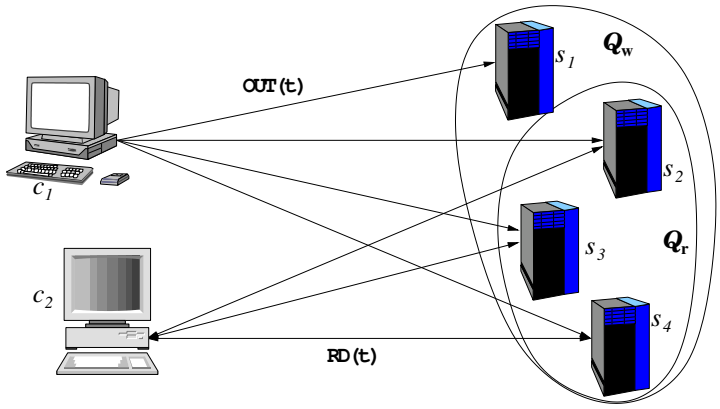

Figura 1: Sistema de quóruns mascaramento assimétrico ($n = 4$ e $f = 1$).

A definição formal das condições de correção e vivacidade destes protocolos bem como a prova de que os mesmos satisfazem essas condições pode ser encontrada na versão estendida desse artigo [Bessani et al., 2005a].

3.3.1. Operação out

A inclusão de uma tupla t no espaço de tuplas replicado é realizada através da invocação da operação out. Esta operação é implementada pelo algoritmo 1.

Algoritmo 1 Operação out (cliente c e servidor s).

1: {**Parte Cliente**}	1: {**Parte Servidor**}
2: **procedure** $out(t)$	**Require:** $receive(c, \langle \text{OUT}, t \rangle)$
3: $\forall s \in Q_w, send(s, \langle \text{OUT}, t \rangle)$	2: $T_s \leftarrow T_s \cup \{t\}$

Devido aos requisitos fracos em termos de sincronismo do modelo generativo e pela premissa de canais confiáveis o cliente não necessita de confirmações de que sua tupla foi recebida pelos servidores. Este tipo de protocolo de escrita, chamado **sem confirmação** [Martin et al., 2002], é mais leve e pode ser usado em sistemas onde o escritor não necessita ter certeza do momento em que sua escrita termina. No caso do algoritmo 1, a escrita termina quando um quórum de leitura completo de servidores corretos recebe a tupla, pois desta forma a tupla inserida poderá ser lida.

3.3.2. Operação rdp

A leitura não destrutiva de uma tupla t, que combina com um molde $\bar{t}$ passado pelo cliente c na invocação de uma operação $rdp(\bar{t})$, acontece através do algoritmo 2.

O funcionamento do algoritmo no cliente consiste basicamente em uma pesquisa nos servidores do sistema tentando obter as tuplas que casam com o molde $\bar{t}$ (linha 3). Cada servidor s acessado responde com um conjunto contendo suas tuplas que combinam com $\bar{t}$ ($T_s^{\bar{t}}$), conforme apresentado nas linhas 2 e 3 do algoritmo servidor. As respostas são coletadas[5] (laço da linha 4-10) até que o cliente consiga definir uma tupla que aparece em pelo menos

[5]O cliente considera apenas uma resposta por servidor.

Algoritmo 2 Operação rdp (cliente c e servidor s).

```
 1: {Parte Cliente}
 2: procedure rdp(t̄)
 3: ∀s ∈ U, send(s, ⟨RD, t̄⟩)
 4: repeat
 5:    wait until receive(s, ⟨RSP-RD, T_s^t̄⟩))
 6:    X ← X ∪ {T_s^t̄}
 7:    if ∃t : (|{T ∈ X : t ∈ T}| = |Q_r|) then
 8:       return t
 9:    end if
10: until |{T ∈ X : T = ∅}| = |Q_r| − f
11: return ⊥
```

```
 1: {Parte Servidor}
Require: receive(c, ⟨RD, t̄⟩)
 2: T_s^t̄ ← {t ∈ T_s : m(t, t̄)}
 3: send(c, ⟨RSP-RD, T_s^t̄⟩)
```

um quórum de leitura completo, sendo esta tupla o retorno da operação (linha 8), ou $|Q_r| - f$ processos retornem um conjunto vazio, indicando não haver tal tupla (linhas 10 e 11).

3.3.3. Operação inp

A operação inp é a que exige o protocolo mais complexo para sua implementação sobre sistemas de quóruns bizantinos. Esta complexidade se deve basicamente ao fato de uma mesma tupla nunca poder ser coletada do espaço de tuplas por duas operações inp. Este requisito implica em uma exclusão mútua natural entre clientes que tentam remover um mesmo tipo de tupla concorrentemente.

Nossa proposta de implementação para esta operação utiliza o algoritmo de exclusão mútua **Confeiteiro Bizantino** [Bessani et al., 2005b], implementado sobre o mesmo sistema de quóruns bizantinos em que estão localizadas as réplicas do espaço de tuplas. Este algoritmo requer também $n \geq 3f + 1$ e está definido sobre duas primitivas: $enter(r)$, que tenta obter acesso ao recurso r, e $exit(r)$, que libera o acesso ao recurso r. O algoritmo do confeiteiro bizantino satisfaz as seguintes propriedades [Lynch, 1996, Bessani et al., 2005b]:

Exclusão Mútua: Não existe um estado global do sistema onde dois processos corretos estão em seção crítica (executaram $enter(r)$ e não começaram a executar $exit(r)$);

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

Utilizando um algoritmo com estas propriedades, torna-se trivial a implementação da operação $inp(\bar{t})$: o processo tenta obter acesso ao tipo da tupla que deseja obter e quando consegue, lê a tupla (através de $rdp(\bar{t})$) e então envia uma mensagem aos servidores removendo a tupla. Ao receber a confirmação de um quórum completo de que a tupla foi apagada, ele libera o tipo da tupla. O algoritmo 3 apresenta esse protocolo.

Durante a execução do algoritmo 3 o cliente, após obter o acesso ao tipo do molde desejado (linha 3) e executar uma leitura com esse molde (linha 4), verifica se existe uma tupla que pode ser removida (o resultado da leitura deve ser diferente de $\perp$ - linha 5). Em caso afirmativo, uma requisição de remoção é enviada a um quórum de escrita de servidores (linha 6) e respostas são coletadas até que $|Q_r| - f$ servidores confirmem a remoção da tupla (linha 14) ou um quórum de leitura negue a remoção (linha 10). O quórum de $|Q_r| - f$ servidores corresponde exatamente a quantidade de servidores que removendo a tupla de

Algoritmo 3 Operação inp (cliente c e servidor s).

```
 1: {Parte Cliente}
 2: procedure inp(t̄)
 3: enter(type(t̄))
 4: t ← rdp(t̄)
 5: if t ≠ ⊥ then
 6:    ∀s ∈ Q_w, send(s, ⟨IN, t⟩)
 7:    repeat
 8:       wait until receive(s, ⟨RSP-IN, r⟩)
 9:       R ← R ∪ {⟨s, r⟩}
10:       if |{⟨s, r⟩ ∈ R : r = ⊥}| = |Q_r| then
11:          exit(type(t̄))
12:          re-begin procedure
13:       end if
14:    until |{⟨s, r⟩ ∈ R : r = t}| = |Q_r| − f
15: end if
16: exit(type(t̄))
17: return t
```

```
 1: {Parte Servidor}
Require: receive(c, ⟨IN, t⟩)
 2: if lockedFor(type(t), c) then
 3:    T_s ← T_s \ {t}
 4:    send(c, ⟨RSP-IN, t⟩)
 5: end if
Require: c ∈ susp_◇M ∧ lockedFor(type(t), c)
 6: unlock(type(t))
 7: send(c, ⟨RSP-IN, ⊥⟩)
```

seus espaços locais a tornam inexistente no espaço global, não podendo portanto ser lida ou removida em futuras operações.

As linhas 2-5 da parte do servidor do algoritmo 3 correspondem a resposta do servidor para a requisição de remoção. Note que um servidor só responde a uma requisição deste tipo se o tipo da tupla requisitada está "travado" para o processo requisitante. Como um cliente faltoso pode obter acesso a um tipo de tupla e não liberá-lo (devido ao seu caráter malicioso ou a uma eventual falta de parada), impedindo que outros processos possam remover tuplas deste tipo, faz-se necessário um mecanismo adicional que permita aos servidores manter a acessibilidade do espaço. O mecanismo mais adequado para este fim é a introdução de premissas de sincronismo fraco encapsuladas em um detector de falhas da classe $\diamondsuit\mathcal{PM}$ [Bessani et al., 2005b], uma classe de detectores de mudez [Doudou et al., 1999] "equivalente" ao detector de faltas de parada $\diamondsuit\mathcal{P}$ [Chandra and Toueg, 1996]. Assim, cada servidor do sistema utiliza um detector desta classe para monitorar seus clientes. Se um cliente é suspeito pelo detector de um servidor enquanto está de posse do tipo da tupla o servidor revoga esta posse ("destravando" sua cópia local) e envia uma notificação ao cliente contendo $r = \perp$ (linhas 6 e 7 da parte servidor).

A utilização do detector de falha pelos servidores se reflete no use de *timeouts* na implementação do algoritmo. Desta forma, o cliente que obtém acesso a um tipo de tupla tem um limite de tempo finito para realizar a remoção da tupla escolhida. Caso este cliente (faltoso ou não) não consiga realizar esta remoção, ele volta ao início e tenta re-executar todo o algoritmo novamente (linhas 11 e 12). Note que o detector de mudez é usado apenas para garantir que todo processo que obtém acesso a um tipo de tupla termine por liberá-lo, garantindo assim a vivacidade do algoritmo.

Utilizando o algoritmo do confeiteiro bizantino para exclusão mútua, o algoritmo 3 requer 9 passos de comunicação[6] para terminar quando executado sem concorrência e 11 quando houver mais de um processo tentando obter acesso ao mesmo tipo de tupla. Este número, inaceitável na prática, pode ser diminuído para 6 (sem concorrência) e 8 passos

[6]Métrica que considera o número de comunicações sequenciais entre processos.

de comunicação fazendo-se com que a leitura seja feita durante a execução do algoritmo de exclusão mútua (utilizando-se as mesmas mensagens) e a mensagem de saída da região crítica seja enviada juntamente com a mensagem a notificação de remoção (linha 6). A figura 2 ilustra a execução deste protocolo sem e com a otimização[7]. As trocas de mensagens do confeiteiro bizantino são descritas em [Bessani et al., 2005b].

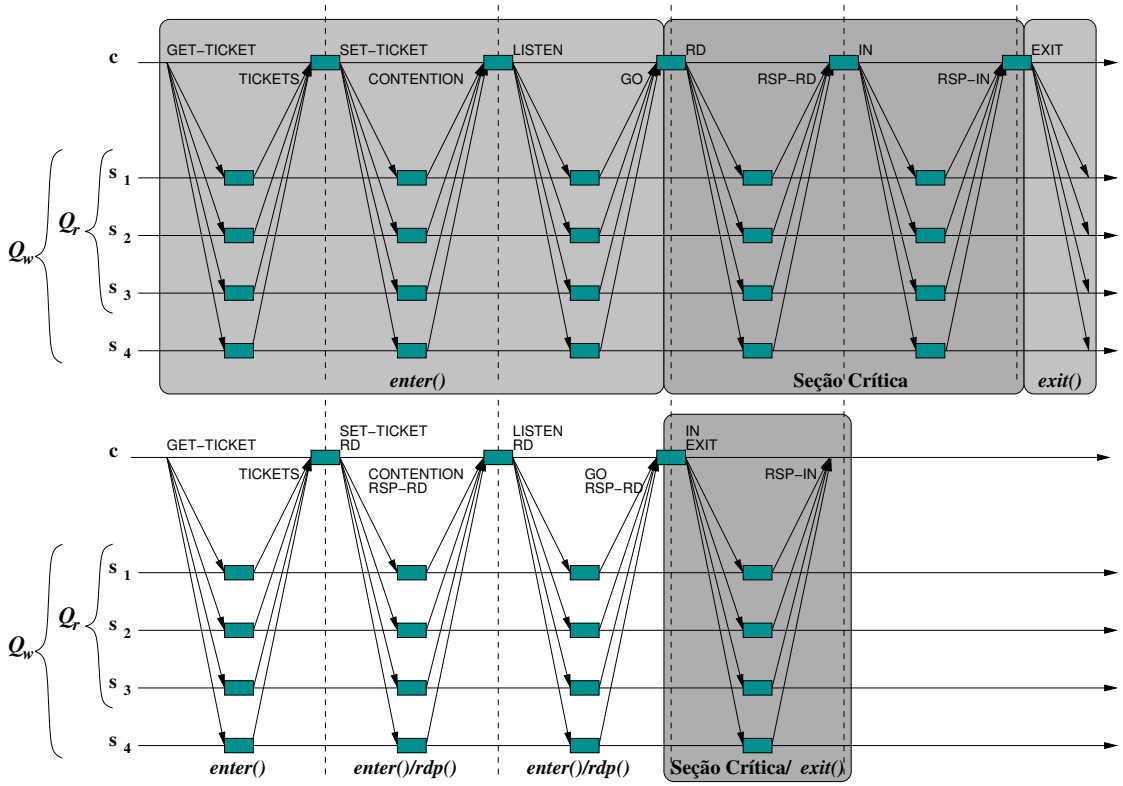

Figura 2: Execuções da operação inp sem e com otimização ($n = 4$ e $f = 1$).

3.3.4. Operações Bloqueantes

As operações bloqueantes rd e in podem ser implementadas fazendo-se com que o cliente execute rdp e inp repetidamente até conseguir a tupla [Xu and Liskov, 1989]. Apesar desta estratégia não ser ótima em termos de desempenho, ela é válida e está de acordo com o não determinismo inerente do espaço de tuplas [Gelernter, 1985].

4. Análise dos Protocolos

A tabela 1 apresenta o custo dos protocolos apresentados. As métricas consideradas são: tamanho das mensagens (T.M.), complexidade de mensagens (C.M.) e passos de comunicação (P.C.).

Os valores da tabela 1 consideram apenas execuções em que os detectores de mudez dos servidores não suspeitam de nenhum processo correto. As operação out e rdp são bastante simples, e portanto têm um custo bastante reduzido se comparadas à operação inp. Esta operação tem seu custo domindo pelo algoritmo de exclusão mútua e portanto exige vários

[7]O padrão de comunicação apresentado na figura 2 não leva em consideração o fato do sistema ser assíncrono. Esta simplificação foi feita visando melhorar a legibilidade da figura.

Operação	T.M.	C.M.	P.C.
out	$O(1)$	$O(n)$	1
rdp	$O(1)$	$O(n)$	2
inp	$O(n)$	$O(n)$	$6/8$

Tabela 1: Custo das operações do espaço de tuplas.

passos de comunicação extras (ver figura 2). Uma importante característica dos protocolos baseados em quóruns é sua complexidade de mensagens linear ($O(n)$), que em geral permite uma utilização mais racional da rede de comunicação.

5. Trabalhos Relacionados

Em face a diversidade de novas aplicações e modelos de computação distribuída, dois modelos de coordenação que apresentam alto desacoplamento têm se tornado muito populares nos últimos anos: o *Publish/Subscribe* [Eugster et al., 2003] e o generativo [Papadopolous and Arbab, 1998, Cabri et al., 2000].

Os sistemas baseados em *Publish/Subscribe* destacam-se em cenários de larga escala onde podem se valer de redes sobrepostas para a distribuição de mensagens entre os processos. Já o modelo generativo, se mostra atraente pela sua simplicidade de programação (apenas 5 primitivas), seu poder (memória associativa) e sua capacidade de sincronizar processos distribuídos.

Existem várias propostas para replicação de espaço de tuplas. Algumas se baseiam em replicação ativa [Bakken and Schlichting, 1995] e outras em sistemas de quóruns [Xu and Liskov, 1989], entretanto, nenhuma delas considera a ocorrência de faltas bizantinas. O trabalho apresentado em [Chiba et al., 1992] explora a semântica da aplicação executada sobre o espaço de tuplas, definindo diferentes protocolos para manutenção da consistência na replicação considerando os padrões de comunicação esperados durante a execução da aplicação. Esta estratégia não pode ser usada em sistemas onde os processos estão sujeitos a faltas bizantinas, uma vez que não se pode presumir que um processo faltoso seguirá algum padrão de comunicação da aplicação.

Atualmente, uma série de trabalhos têm se concentrado na integração de mecanismos de segurança no modelo generativo. Esta preocupação é legítima na medida em que o modelo vem sendo cada vez mais utilizado em ambientes "hostis" como a Internet. Dentre as propostas presentes na literatura, algumas procuram reforçar políticas de segurança pré-estabelecidas de acordo com o comportamento esperado da aplicação executada [Minsky et al., 2000], outras no entanto se concentram na manutenção da confidencialidade e a integridade das tuplas [De Nicola et al., 1998, Vitek et al., 2003, Busi et al., 2003]. Dentre estes trabalhos, nenhum considera a disponibilidade do espaço de tuplas, objetivo principal dos mecanismos de tolerância a faltas, e muito menos aplicam qualquer abordagem mais robusta relacionada a tolerância a intrusões.

6. Considerações Finais e Trabalhos Futuros

Os requisitos dos modernos sistemas distribuídos abertos, como as redes *ad hoc* e as grades computacionais, têm exigido modelos cada vez mais desacoplados de comunicação. Este trabalho apresenta uma proposta de modelo de coordenação generativa (baseada em espaço de tuplas) que tolera faltas bizantinas a partir da aplicação de replicação por sistemas de

quóruns bizantinos. Até onde sabemos, nossa proposta é a primeira a oferecer este nível de segurança de funcionamento, e nesse fato reside sua maior contribuição. Adicionalmente, os protocolos apresentados definem um objeto de memória compartilhada estritamente mais forte (em termos da hierarquia livre de espera [Herlihy, 1991]) que os registradores usuais implementados sobre sistemas de quóruns.

A construção apresentada neste artigo forma a base para nosso trabalho em coordenação desacoplada tolerante a intrusões para sistemas abertos. Nesse contexto, os trabalhos futuros seguem em várias direções: implementação e simulação dos protocolos (atividade já em andamento), implementação dos protocolos sobre outros tipos de quóruns que provêem um melhor balanceamento de carga como os baseados em grades e partições [Malkhi and Reiter, 1998] e finalmente, a extensão dos protocolos apresentados considerando temas relacionados a segurança (e tolerância a intrusões) como a provisão de confidencialidade das tuplas (utilizando criptografia de limiar).

Referências

Bakken, D. E. and Schlichting, R. D. (1995). Supporting fault-tolerant parallel programing in linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302.

Bessani, A. N., da Silva Fraga, J., and Lung, L. C. (2005a). Coordenação desacoplada tolerante a faltas bizantinas. http://www.das.ufsc.br/∼neves/reports/2005-2.pdf.

Bessani, A. N., da Silva Fraga, J., and Lung, L. C. (2005b). O confeiteiro bizantino: Exclusão mútua em sistemas abertos sujeitos a faltas bizantinas. In *Anais do 23o. Simpósio Brasileiro de Redes de Computadores - SBRC 2005*, Fortaleza, CE, Brasil.

Busi, N., Gorrieri, R., Lucchi, R., and Zavattaro, G. (2003). Secspaces: a data-driven coordination model for environments open to untrusted agents. In Brogi, A. and Jacquet, J.-M., editors, *Electronic Notes in Theoretical Computer Science*, volume 68. Elsevier.

Cabri, G., Leonardi, L., and Zambonelli, F. (2000). Mobile agents coordination models for internet applications. *IEEE Computer*, 33(2):82–89.

Carriero, N. and Gelernter, D. (1989). Linda in context. *Communications of the ACM*, 32(4):444–458.

Castro, M., Rodrigues, R., and Liskov, B. (2003). Base: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269. A preliminar version appeared *18th Symposium on Operating Systems Principles*, 2001.

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

Chiba, S., Kato, K., and Masuda, T. (1992). Exploiting a weak consistency to implement distributed tuple space. In *Proceedings of the 12th International Conference on Distributed Computing Systems - ICDCS'92.*, pages 416–423, Yokohama, Japan. IEEE Computer Society Press.

De Nicola, R., Ferrari, G. L., and Pugliese, R. (1998). Klaim: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330.

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.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131.

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.

Fraga, J. and Powell, D. (1985). A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd International Conference on Computer Security*, pages 203–218.

Gelernter, D. (1985). Generative communication in linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112.

Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Communications of ACM*, 35(2):96–107.

Herlihy, M. (1991). Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1):124–149. Dijkstra Prize 2003 winner.

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

Lehman, T. J., Cozzi, A., Xiong, Y., Gottschalk, J., Vasudevan, V., Landis, S., Davis, P., Khavar, B., and Bowman, P. (2001). Hitting the distributed computing sweet spot with TSpaces. *Computer Networks*, 35(4):457–472.

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

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

Martin, J.-P., Alvisi, L., and Dahlin, M. (2001). Small byzantine quorum systems. In *Dependable Systems and Networks, DSN 01*. IEEE Computer Society.

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.

Minsky, N. H., Minsky, Y. M., and Ungureanu, V. (2000). Making tuple spaces safe for heterogeneous distributed systems. In *Proceedings of the 2000 ACM symposium on Applied computing*, pages 218–226. ACM Press.

Papadopolous, G. and Arbab, F. (1998). Coordination models and languages. In *The Engineering of Large Systems*, volume 46 of *Advances in Computers*. Academic Press.

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.

Sun Microsystems (2003). Javaspaces service specification. Disponível em http://www.jini.org/nonav/standards/davis/doc/specs/html/js-spec.html.

Veríssimo, P., Neves, N. F., and Correia, M. P. (2003). Intrusion-tolerant architectures: Concepts and design. Technical Report DI-FCUL TR-03-5, Universidade de Lisboa, Lisboa, Portugal.

Vitek, J., Bryce, C., and Oriol, M. (2003). Coordination processes with secure spaces. *Science of Computer Programming*, (46):163–193.

Xu, A. and Liskov, B. (1989). A design for a fault-tolerant, distributed implementation of linda. In *Proceedings of the 19th Symposium on Fault-Tolerant Computing - FTCS'89.*, pages 199–206. IEEE Computer Society Press.