DEPSPACE: Um *Middleware* para Coordenação em Ambientes Dinâmicos e Não Confiáveis*

Alysson Neves Bessani[1] , Eduardo Adilio Pelinson Alchieri[2], Miguel Correia[1] , Joni da Silva Fraga[2] , Lau Cheuk Lung[3]

[1]LaSIGE - Faculdade de Ciências da Universidade de Lisboa

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

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

Abstract. *Modern distributed systems have to deal with several contradicting challenges, like openness, mobility, security and tolerance to accidental faults. Tuple spaces are a promising communication model for those environments due to their time and space decoupling, and their synchronization power. This paper presents* DEPSPACE*, a fault and intrusion-tolerant tuple space. This system provides a communication infrastructure that can be used to implement trustworthy services, tolerating both accidental faults (e.g., crashes), malicious attacks and even intrusions in some of the system's components. The paper also presents an implementation of a synchronization barrier on top of* DEPSPACE *that tolerates malicious processes.*

Resumo. *Os sistemas distribuídos modernos têm de lidar com requisitos desafiadores como abertura, mobilidade, segurança a tolerância a faltas acidentais. Espaços de tuplas representam um modelo de comunicação promissor nesses ambientes devido a seu desacoplamento temporal e espacial, bem como seu poder de sincronização. Este artigo apresenta o* DEPSPACE*, um espaço de tuplas tolerante a faltas e intrusões. Este sistema provê uma infraestrutura de comunicação que pode ser usada para implementar serviços confiáveis, tolerando faltas, ataques e intrusões em alguns de seus componentes. O artigo apresenta também uma implementação de uma barreira de sincronização tolerante a faltas usando o* DEPSPACE*.*

1. Introdução

A maioria dos sistemas distribuídos modernos têm características de sistemas abertos, os quais tipicamente são compostos por um número desconhecido de processos, executando em ambientes heterogêneos e não confiáveis, conectados através de redes também heterogêneas e não confiáveis, como a Internet. Em vista disso, existe uma grande e importante demanda por ferramentas que permitam a construção de aplicações mais complexas de forma eficiente e rápida.

Dentre as abordagens empregadas na concepção destas ferramentas, o modelo de coordenação por espaço de tuplas [Gelernter 1985] destaca-se por oferecer uma comunicação onde as interações são desacopladas no tempo (os participantes não precisam estar ativos no mesmo instante) e no espaço (os participantes não precisam se conhecer), sendo uma abordagem flexível e simples.

Deste modo, vários trabalhos sobre a introdução de tolerância a faltas neste modelo foram propostos, tanto através da construção de espaços de tuplas tolerantes a faltas quanto na concepção de mecanismos que permitem a construção de aplicações tolerantes a faltas sobre o espaço de tuplas (ex. [Bakken and Schlichting 1995]). Com relação à segurança, espaços de

*Realizado com recursos do CNPq (projeto número 550114/2005-0).

tuplas seguros foram propostos (ex. [Busi et al. 2003]), os quais garantem que apenas processos autorizados podem executar operações no espaço de tuplas, através de mecanismos de controle de acesso (tanto em nível de espaço quanto de tuplas).

Estes trabalhos sobre tolerância a faltas e segurança para espaço de tuplas têm um foco limitado em pelo menos dois sentidos: eles consideram apenas faltas acidentais por parada e ataques simples (acesso inválido). Este artigo apresenta uma ferramenta que considera uma abordagem mais abrangente para a concepção de espaços de tuplas, agrupando mecanismos de tolerância a faltas (como replicação) e de segurança (como criptografia). Deste modo, a ferramenta apresentada, chamada DEPSPACE, representa um espaço de tuplas com segurança de funcionamento e permite a implementação de sistemas capazes de fornecer serviços corretamente mesmo que uma parte de seus componentes sejam atacados, invadidos e controlados por adversários, i.e., sistemas tolerantes a intrusões [Veríssimo et al. 2003]. O DEPSPACE está disponível na página do projeto JITT (*Java Intrusion Tolerance tools*): `http://www.das.ufsc.br/~neves/jitt/depspace.html`.

2. Espaço de Tuplas com Segurança de Funcionamento

Um espaço de tuplas pode ser visto (conceitualmente) como um objeto de memória compartilhada que fornece operações para armazenar e recuperar conjuntos de dados ordenados chamados de tuplas. Uma *tupla* t é uma seqüência ordenada de campos, onde um campo que contém um valor é dito *definido*. Um tupla onde todos os campos são definidos é chamada de *entrada*. Uma tupla $\bar{t}$ é chamada *molde* se algum de seus campos não tem valor definido. Uma tupla t e um molde $\bar{t}$ *combinam* se e somente se eles têm o mesmo número de campos e todos os valores dos campos definidos em $\bar{t}$ são iguais aos valores dos campos correspondentes em t. Por exemplo, uma tupla $\langle \text{CLIENTE}, 12, abc \rangle$ combina com o molde $\langle \text{CLIENTE}, *, abc \rangle$ ('$*$' denota um campo não definido do molde).

As manipulações realizadas no espaço de tuplas consistem em invocações de três operações básicas [Gelernter 1985]: $out(t)$, que adiciona a entrada t no espaço de tuplas (inserção); $in(\bar{t})$, que remove do espaço de tuplas uma tupla que combina com o molde $\bar{t}$ (leitura destrutiva); e $rd(\bar{t})$, usada na leitura de uma tupla que combina o molde $\bar{t}$, sem removê-la do espaço (leitura não-destrutiva). As operações in e rd são bloqueantes, i.e., se não houver uma tupla que combine com o molde no espaço, o processo fica bloqueado até que uma esteja disponível. Uma extensão comum a este modelo, é a inclusão de variantes não bloqueantes das operações de leitura, denominadas inp e rdp. Estas operações funcionam exatamente como as anteriores, a não ser pelo fato de retornarem mesmo não havendo uma tupla que combine com o molde usado (indicando esta inexistência).

Visando aumentar o poder de sincronização do espaço de tuplas, o DEPSPACE também implementa a operação $cas(\bar{t}, t)$ (*conditional atomic swap*). Esta operação funciona como uma execução indivisível do código: **if** $\neg rdp(\bar{t})$ **then** $out(t)$. Sendo assim, a tupla t será inserida no espaço somente se $rdp(\bar{t})$ não retornar alguma tupla, i.e., se não existir uma tupla no espaço que combine com $\bar{t}$. A operação cas é importante porque permite que o DEPSPACE seja capaz de resolver o problema do consenso em sistemas assíncronos [Bessani et al. 2006], o qual é a base para a solução de muitos problemas de sincronização distribuída.

Para um espaço de tuplas possuir segurança de funcionamento, que é uma característica fundamental dos sistemas ditos confiáveis e seguros, é necessário que o mesmo suporte os atributos de segurança de funcionamento [Avizienis et al. 2004] aplicáveis no contexto de um espaço de tuplas, que são: **confiabilidade**, as operações realizadas no espaço de tuplas fazem com que seu estado se modifique de acordo com sua especificação; **disponibilidade**, o espaço de tuplas sempre está pronto para executar as operações requisitadas por partes autorizadas;

integridade, nenhuma alteração imprópria no estado de um espaço de tuplas pode ocorrer, i.e., o estado de um espaço de tuplas só pode ser alterado através da correta execução de suas operações; e **confidencialidade**, o conteúdo de campos de uma tupla não podem ser revelados a partes não autorizadas. O DEPSPACE satisfaz todos esses atributos, desde que certas premissas sejam satifeitas.

3. Premissas de Funcionamento

O DEPSPACE assume uma série de características do ambiente onde é executado para garantir os atributos de segurança de funcionamento. Em primeiro lugar, o espaço de tuplas é replicado, sendo requeridos $n \geq 3f + 1$ servidores (réplicas do espaço), dos quais até f podem sofrer faltas bizantinas [Lamport et al. 1982], se comportando de forma arbitrária. É assumido independência de faltas, o que requer diversidade na instalação do sistema [Obelheiro et al. 2005].

Tendo em vista o protocolo de coordenação de réplicas usado pelo DEPSPACE (PAXOS bizantino [Castro and Liskov 2002]), assume-se que o sistema apresenta sincronismo parcial: existem limites para o tempo necessário para a transmissão de uma mensagem e para a realização de qualquer computação que terminam por valer no sistema, no entanto, esses limites não são conhecidos [Dwork et al. 1988].

Finalmente, o correto funcionamento do sistema está condicionado a existência de pares de chaves público-privadas usadas para estabelecimento de canais autenticados e segredos compartilhados (para uso de criptografia simétrica) entre os processos. Este tipo de mecanismo pode ser provido por uma infraestrutura de chave pública [Bishop 2002].

4. Arquitetura do DEPSPACE

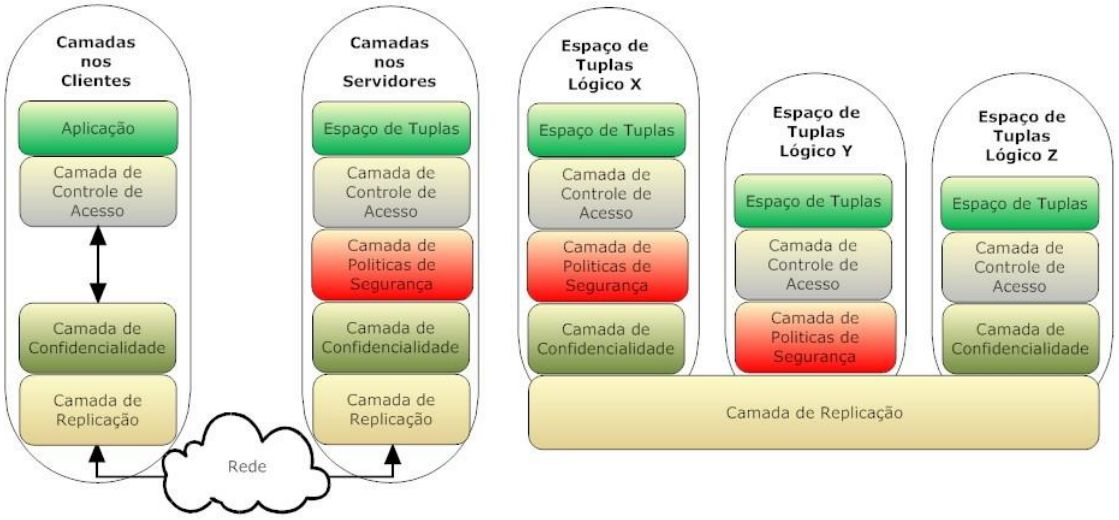
O DEPSPACE é composto por um conjunto de camadas, sendo que em cada camada uma funcionalidade diferente é concretizada. A estrutura do DEPSPACE é apresentada na figura 1(a), onde podemos observar quais são as camadas que compõem o sistema tanto nos clientes (que acessam o espaço replicado), quanto nos servidores. No topo da pilha do cliente temos a aplicação (que acessa o espaço) e no topo da pilha do servidor encontra-se uma implementação local de um espaço de tuplas. No cliente ainda encontram-se as camadas de controle de acesso, de confidencialidade e de replicação. No lado do servidor a arquitetura é similar, existindo ainda uma camada adicional responsável pela verificação de políticas de segurança. Estas diversas camadas serão apresentadas nas próximas seções.

Um aspecto chave do serviço oferecido pelo DEPSPACE é o suporte a múltiplos espaços de tuplas lógicos, i.e., o sistema fornece interfaces de administração que permitem a criação de espaços de tuplas e estes espaços não têm nenhuma relação uns com os outros. Além disso, o DEPSPACE pode ser configurado de acordo com as necessidades das aplicações, i.e., pode-se escolher quais serão as camadas que estarão ativas em um determinado espaço de tuplas lógico[1], bem como suas configurações. Estes aspectos podem ser observados na figura 1(b), que representa um servidor onde três espaços de tuplas lógicos foram criados. Note que, as camadas ativas não são as mesmas nos espaços lógicos suportados por este servidor. A configuração da pilha de camadas de um espaço lógico pode ser qualquer combinação com as camadas de confidencialidade, políticas de segurança e controle de acesso.

4.1. Replicação Tolerante a Faltas Bizantinas

No DEPSPACE, o espaço de tuplas é mantido replicado em um conjunto de servidores de tal forma que falhas em alguns deles não ferem nenhum atributo de segurança de funcionamento do sistema. Este conjunto de servidores utiliza a replicação Máquina de Estados [Schneider 1990],

[1]Todas as camadas, com excessão da camada de replicação, são opcionais.



(a) Camadas do DEPSPACE. (b) Espaços de tuplas lógicos.

Figura 1. Características do DEPSPACE.

uma solução clássica para implementar sistemas tanto tolerantes a faltas por parada (acidentais) quanto faltas bizantinas (maliciosas) [Schneider 1990, Castro and Liskov 2002]. Este mecanismo está relacionado principalmente com as propriedades de disponibilidade e confiabilidade, pois garante que o espaço de tuplas executa as operações a ele endereçadas seguindo sua especificação, mesmo que até f de n réplicas sejam maliciosas (as réplicas corretas *mascaram* o comportamento das maliciosas).

O ponto fundamental da replicação Máquina de Estados é garantir que todas as réplicas corretas executam a mesma seqüência de operações e retornam, evoluindo portanto de forma sincronizada. Isto é garantido através do uso de um protocolo de difusão com ordem total[2]. Neste trabalho usamos como base o algoritmo PAXOS em sua versão tolerante a faltas bizantinas [Castro and Liskov 2002]. Outro ponto importante é o determinismo das réplicas: o resultado de uma operação em diferentes réplicas do espaço de tuplas deve ser sempre o mesmo.

O protocolo de replicação implementado é simples: o cliente envia uma requisição usando o protocolo de difusão com ordem total e espera por $f+1$ respostas iguais advindas de diferentes servidores. Como todos os servidores recebem o mesmo conjunto de requisições na mesma ordem, e o espaço de tuplas é determinista, sempre existirá pelo menos $2f+1$ servidores corretos que executarão a requisição e responderão da mesma forma.

4.2. Confidencialidade

Garantir confidencialidade no DEPSPACE, sendo este um espaço de tuplas replicado, não é uma tarefa trivial. O problema é que não é possível confiar nos servidores individualmente visto que até f podem falhar e revelar o conteúdo das tuplas a partes não autorizadas. Deste modo, a provisão desta propriedade deve ser confiada a um conjunto de servidores, ou seja, uma tupla não deve ser entregue (inteira) a um único servidor.

Sendo assim, a confidencialidade é conseguida através do uso de um $(n, f+1)$ – esquema de compartilhamento de segredo publicamente verificável (*public verificable secret sharing* – PVSS) [Schoenmakers 1999]. Os clientes, que são os distribuidores deste esquema, cifram as tuplas com um segredo por eles gerado. Após isso, geram um conjunto de n fragmentos (*shares*) deste segredo (neste processo utiliza as chaves públicas dos servidores). Um segredo pode ser

[2]Um protocolo deste tipo garante que mensagens difundidas para um grupo de processos são recebidas por todos os processos na mesma ordem.

remontado apenas com a combinação de $f + 1$ *shares*, o que torna impossível que uma coalisão de servidores faltosos revele o conteúdo de uma tupla. Este esquema utiliza um *fingerprint* da tupla para suportar a comparação entre tuplas e moldes, o qual é computado dependendo do tipo dos campos da tupla: **público**, o próprio valor do campo é o *fingerprint*; **comparável**, um *hash* do valor do campo é o *fingerprint* (para isso utiliza uma função de *hash* resistente a colisões); **privado**, um símbolo especial é o *fingerprint*.

Como não é possível enviar diferentes versões de uma requisição para diferentes servidores (contendo apenas seu *share* da tupla), o cliente deve cifrar cada um dos *shares* com uma chave secreta compartilhada com o servidor que vai armazenar esse *share*. Deste modo, cada servidor terá acesso apenas ao *share* a ele endereçado (caso contrário, um servidor faltoso teria acesso a todos os *shares* e poderia remontar e revelar a tupla). Assim, nas requisições de inserção de tuplas, o cliente envia aos servidores a tupla cifrada, os *shares* cifrados, as provas de que estes *shares* são válidos e o *fingerprint* da tupla. Para acessar uma tupla, o cliente envia o *fingerprint* do molde e espera pelas respostas dos servidores. A resposta de cada servidor contém o *fingerprint* da tupla (que combina com o *fingerprint* do molde), a tupla cifrada e o *share* armazenado por este servidor[3] (juntamente com a prova de sua validade). O cliente decifra os *shares*, verifica suas validades e combina $f + 1$ deles para obter o segredo e decifrar a tupla. Note que, um cliente malicioso pode inserir uma tupla e informar um *fingerprint* que não corresponde ao *fingerprint* da tupla. Deste modo, após obter a tupla, o cliente deve verificar se a tupla correponde ao *fingerprint*. Caso isso não aconteça, o cliente precisa eliminar esta tupla do espaço (se ainda não eliminou) e re-executar a operação. A eliminação de tuplas inválidas é realizada em dois passos: *(1.)* o cliente envia todas as respostas recebidas para os servidores como prova que esta tupla é inválida (para isso ser possível, os servidores devem assinar as respostas a estas requisições); e *(2.)* os servidores verificam a autenticidade das respostas e, se a tupla realmente é inválida, removem-na de seus espaços locais.

4.3. Controle de Acesso

O controle de acesso é um mecanismo fundamental para manutenção da integridade e confidencialidade das informações (tuplas) armazenadas no DEPSPACE, pois previne que clientes não autorizados obtenham acesso as tuplas, além de impedir que clientes faltosos saturem o espaço de tuplas enviando uma grande quantidade de tuplas. Atualmente, o DEPSPACE implementa controle de acesso de duas formas:

Baseado em credenciais: para cada tupla inserida no DEPSPACE pode-se definir quais são as credenciais necessárias para acessá-la, tanto para leitura quanto para remoção (acesso em nível de tuplas). Estas credenciais são definidas pelo processo que insere a tupla. Também é possível definir, quando o espaço de tuplas lógico é criado, quais são as credenciais necessárias para inserir uma tupla no espaço (acesso em nível de espaço).

A implementação atual desta camada é feita através da associação de listas de controle de acesso a cada espaço lógico e tupla, definindo quais clientes podem inserir tuplas (no caso do espaço de tuplas) e ler e/ou remover uma determinada tupla.

Políticas de granularidade fina: o DEPSPACE suporta a definição de políticas de acesso de granularidade fina [Bessani et al. 2006], que devem ser especificadas no momento da criação do espaço de tuplas lógico. Estas políticas controlam o acesso ao espaço considerando três parâmetros: o identificador do cliente, a operação que será executada (juntamente com seus argumentos) e o estado do espaço. Um exemplo de política será apresentado na seção 6.

Atualmente, o sistema reconhece apenas políticas definidas na linguagem de programação GROOVY [Codehaus 2006], que é uma linguagem de *script* suportada pela

[3]O *share* é cifrado com a chave secreta compartilhada com o cliente para evitar *eavesdropping* das respostas.

máquina virtual Java (JVM) e possui várias características que tornam a programação fácil e ágil. Os scripts com a política de acesso do espaço são enviados aos servidores na forma de uma *string* durante a criação do espaço lógico. Nos servidores, o script é transformado em *byte-code* Java e instanciado como um monitor de referência que verifica todas as operações a serem executadas no espaço (o script implementa uma operação de autorização para cada operação suportada pelo sistema). Este método é bastante eficiente já que após a criação do espaço, a política se torna código binário dentro da pilha de mecanismos dos servidores DEPSPACE.

5. Aspectos de Implementação e Otimizações

O DEPSPACE foi implementado usando a linguagem de programação Java. Todas as primitivas e algoritmos criptográficos utilizadas no DEPSPACE são fornecidas pelo provedor padrão da biblioteca JCE (*Java Cryptography Extensions*). Dois componentes do sistema são interessantes por si só e merecem maior destaque. O primeiro é o mecanismo de confidencialidade, que foi implementado seguindo as especificações de [Schoenmakers 1999], usando grupos algébricos de 192 bits. Esta implementação deu origem a uma biblioteca para compartilhamento de segredos chamada JSS (*Java Secret Sharing*). O outro componente bastante complexo do DEPSPACE é o protocolo de difusão com ordem total baseado no PAXOS bizantino. A concretização deste componente consumiu boa parte do esforço de implementação do DEPSPACE, dando origem a uma biblioteca que provê um serviço de difusão com ordem total chamada JBP (*Java Byzantine Paxos*). Ressaltamos que tanto o JSS quanto o JBP são as primeiras implementações destes componentes em Java conhecidas, e estão livremente disponíveis em `http://www.das.ufsc.br/~neves/jitt/`.

Duas otimizações de implementação são importantes no desempenho do sistema: *(1.)* executar as operações de $rd()$ e $rdp()$ sem utilizar o protocolo de difusão atômica e esperar por $n - f$ respostas. Caso todas as respostas forem iguais, o valor nelas retornado é o resultado da operação, no contrário a operação deve ser executada pelo protocolo normal; *(2.)* assinar as respostas de operação de leitura apenas quando o cliente explicitar esta necessidade (caso encontre tuplas inválidas).

Devido as limitações de espaço, não apresentamos maiores detalhes de implementação nem tampouco medidas de desempenho para o DEPSPACE. Maiores informações acerca desses dois pontos podem ser encontradas em [Bessani et al. 2007].

6. Aplicação Exemplo: Barreira de Sincronização

Nesta seção apresentamos um serviço de coordenação distribuída construído sobre o DEPSPACE: uma abstração de barreira de sincronização que pode ser usada para sincronização de processos distribuídos em um ambiente não confiável.

A barreira talvez seja a primitiva de sincronização mais simples existente, pois consiste basicamente em um ponto onde processos executando tarefas concorrentes se encontram. Uma abstração desse tipo fornece uma operação *enter()*, a qual é executada pelos diversos processos concorrentes e só retorna quando todos os processos requeridos chegam na barreira (i.e., invocam *enter()*).

Nossa implementação sobre o DEPSPACE é bastante simples e consiste basicamente na inserção de uma tupla (com rótulo `BARRIER`) que define o nome da barreira, o conjunto de processos esperados e um parâmetro f (definido mais a frente) para a criação da barreira e na implementação da operação *enter()*, compreendendo a inserção de uma tupla de entrada no espaço (com rótulo ENTER) e na inspeção periódica do mesmo até que os outros processos esperados também tenham inserido suas tuplas. O pseudo-código para uma classe Java que

implementa esse serviço[4] é apresentado na figura 2.

```
public class Barrier {
  private DepSpace ts; //referência para o espaço de tuplas lógico
  private String b; //nome da barreira
  private Collection p; //conjunto de processos a serem esperados
  private int f; //número esperado de processos faltosos

  public Barrier(DepSpace ts, String b, Collection p, int f) {
    if(!ts.rdp(<BARRIER,b,?p,?f>) {
      ts.out(<BARRIER,b,p,f>); //barreira b é criada
    }
    this.ts = ts; this.b = b; this.p = p; this.f = f;
  }

  public enter(int myself) {
    if(ts.rdp(<BARRIER,b,*,*>)) {
      ts.out(<ENTER,b,myself>); //myself entra na barreira
      repeat { //verifica as tuplas pendentes
        for(int i:p) if(ts.rdp(<ENTER,b,i>)) p.remove(i);
        wait(POOLING_TIME);
      } until(p.size() <= f);
    }else throw new RuntimeException("Barrier does not exists.");
  }
}
```

Figura 2. Pseudo-código para barreira de sincronização.

Como consideramos um ambiente sujeito a processos maliciosos, é possível que nem todos os processos esperados cheguem à barreira (ou informem que chegaram à barreira), por isso, a implementação da figura 2 suporta um parâmetro f que define o número máximo de processos que assumimos que podem não chegar a barreira. Desta forma, uma barreira é liberada quando todos os processos menos f inserirem tuplas no espaço. Note que isso enfraquece a semântica da barreira (nem sempre todos os processos corretos vão se sincronizar), no entanto, segundo [Albrecht et al. 2006], o uso eficiente desta abstração em ambientes abertos só faz sentido desta forma. Além disso, para evitar inconsistências nas tuplas usadas nas barreiras, um conjunto de regras de controle de acesso devem ser especificadas na política. Estas regras são definidas na figura 3 (operações sem regras definidas não são permitidas).

```
//Todo processo pode ler toda tupla
boolean can_rdp(client,t){return true;}
//Não deixar inserir duas barreiras com o mesmo nome
boolean can_out(client,<BARRIER,b,p,f>){
  return !ts.rdp(<BARRIER,b,*,*>);
}
//Somente o processo pode invocar sua entrada na barreira (uma única vez)
//e ele deve ser um dos processo aguardados
boolean can_out(client,<ENTER,b,myself>){
  return ts.rdp(<BARRIER,b,?p,*>) && client == myself &&
         p.contains(client) && !ts.rdp(<ENTER,b,myself>);
}
```

Figura 3. Política de controle de acesso para barreira de sincronização.

[4]O código não contém tratamento de erros e a sintaxe de manipulação das tuplas está bastante simplificada. Por exemplo, um valor `?x` em um campo de um molde significa que o valor do campo da tupla lida (que combina com este molde) será atribuído a variável `x`.

Vale notar que esta implementação é muito simples. No entanto, dada a versatilidade do DEPSPACE, mesmo características avançadas das barreiras, como as apresentadas em [Albrecht et al. 2006], podem ser implementadas. Neste mesmo trabalho podem ser encontradas várias aplicações para as barreiras com semântica fraca.

7. Conclusões

Este artigo apresentou a ferramenta DEPSPACE, que é a implementação de um espaço de tuplas com segurança de funcionamento, cuja arquitetura proposta integra mecanismos de tolerância a faltas e de segurança.

Foi apresentado também um exemplo de serviço implementado usando o DEPSPACE: uma barreira de sincronização tolerante a faltas bizantinas. Este exemplo demonstra a facilidade de se implementar serviços de propósito geral usando a ferramenta apresentada. Outros exemplos de serviços que podem ser implementados são serviço de nomes, provisão de travas (*lock service*) e localização de servidores (ex. *anycast*).

Referências

Albrecht, J., Tuttle, C., Snoeren, A. C., and Vahdat, A. (2006). Loose synchronization for large-scale networked systems. In *Proceedings of the 2006 Usenix Annual Technical Conference – Usenix'06*.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

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., Alchieri, E. A. P., Fraga, J. S., and Lung, L. C. (2007). Design and implementation of a dependable tuple space. In *Proceedings of the WRAITS'07: 1st Workshop on Recent Advances on Intrusion Tolerant Systems (with EuroSys 2007)*.

Bessani, A. N., Correia, M., Fraga, J. S., and Lung, L. C. (2006). Sharing memory between Byzantine processes using policy-enforced tuple spaces. In *Proceedings of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*.

Bishop, M. (2002). *Computer Security: Art and Science*. Addison-Wesley.

Busi, N., Gorrieri, R., Lucchi, R., and Zavattaro, G. (2003). SecSpaces: a data-driven coordination model for environments open to untrusted agents. *Electronic Notes in Theoretical Computer Science*, 68(3):310–327.

Castro, M. and Liskov, B. (2002). Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions Computer Systems*, 20(4):398–461.

Codehaus (2006). Groovy programing language homepage. Avaliable at *http://groovy.codehaus.org/*.

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322.

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

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

Obelheiro, R. R., Bessani, A. N., and Lung, L. C. (2005). Analisando a viabilidade da implementação prática de sistemas tolerantes a intrusões. In *Anais do V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg 2005*.

Schneider, F. B. (1990). Implementing fault-tolerant service using the state machine aproach: A tutorial. *ACM Computing Surveys*, 22(4):299–319.

Schoenmakers, B. (1999). A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'99*, pages 148–164.

Veríssimo, P., Neves, N. F., and Correia, M. P. (2003). Intrusion-tolerant architectures: Concepts and design. In Lemos, R., Gacek, C., and Romanovsky, A., editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*. Springer-Verlag.