# Transações em Espaços de Tuplas com Segurança de Funcionamento

**Fábio Favarim[1], Joni da Silva Fraga[1], Eduardo Adílio Pelinson Alchieri[1], Alysson Neves Bessani[2], Lau Cheuk Lung[3]**

[1]DAS - Departamento de Automação e Sistemas
[3]INE - Departamento de Informatica e Estatística
UFSC - Universidade Federal de Santa Catarina

[2]DI - Departamento de Informática
FCUL – Faculdade de Ciências da Universidade de Lisboa

***Resumo.*** *O uso de espaços de tuplas tem se mostrado uma solução atrativa para coordenação entre processos em sistemas distribuídos abertos. Dada a heterogeneidade e volatilidade destes sistemas, seus processos estão sujeitos a falhas que podem levar a inconsistências no estado do espaço de tuplas. O emprego de transações trata desse problema, pois permite que conjuntos de operações sejam executados de forma integral, mesmo em caso de falhas dos processos. Este trabalho apresenta um modelo de transações para espaços de tuplas que garante as propriedades ACID mesmo em ambientes sujeitos a falhas bizantinas. O modelo foi implementando e integrado ao* DEPSPACE*, uma implementação de espaços de tuplas que suporta atributos de segurança de funcionamento. Para validar o modelo e sua implementação, é apresentado um conjunto de experimentos que mensuram o custo das transações no* DEPSPACE.

## 1. Introdução

Tecnologias emergentes passaram a provocar o aparecimento de novos modelos de sistemas distribuídos (Redes Auto-organizáveis, Grades Computacionais, Redes P2P, etc.). Nestes modelos computacionais as execuções das aplicações são baseadas no uso de recursos distribuídos desconhecidos, e muitas vezes concretizadas de maneira completamente descentralizada. Tais modelos, portanto, podem envolver execuções com um número desconhecido de processos, executando em máquinas heterogêneas e não confiáveis, conectados geralmente por redes também não confiáveis, como a Internet.

Nestes sistemas, os mecanismos de coordenação usualmente empregados, como a comunicação direta por passagem de mensagens, nem sempre são os mais adequados. A exigência de que os pares comunicantes estejam disponíveis ao mesmo tempo na aplicação para interagirem segundo este tipo de comunicação pode ser muito restritiva em alguns sistemas. Outra dificuldade, devido às suas dimensões e volatilidade, é o conhecimento prévio da localização dos pares comunicantes. Manter informações atualizadas da localização de processos, em geral, é uma tarefa complexa nestes sistemas.

Neste contexto, de sistemas dinâmicos, o modelo de coordenação por espaços de tuplas [Gelernter, 1985] se torna um mecanismo bastante viável para a comunicação. Neste tipo de coordenação, os participantes de uma computação distribuída interagem através de um objeto de memória compartilhada, chamado de **Espaço de Tuplas**, em que estruturas de dados genéricas, chamadas de **tuplas**, são inseridas, lidas e removidas durante as interações. A coordenação ocorre de maneira desacoplada no tempo e no espaço: processos comunicantes não precisam saber a localização (endereço) um dos outros e nem estarem disponíveis simultaneamente para poderem interagir. Além disso, o número reduzido de operadores e sua generalidade implica em uma abordagem flexível

e de grande simplicidade (em termos de programação) na implementação de sistemas distribuídos abertos.

Recentemente, entre nossos esforços, está o projeto e desenvolvimento do DEPSPACE [Bessani et al., 2008] que é uma implementação de um espaço de tuplas visando a segurança de funcionamento (*dependability*). O DEPSPACE agrupa mecanismos de tolerância a faltas (como replicação) e de segurança (como criptografia) provendo seu serviço de forma contínua e correta, mesmo que uma parte de seus componentes falhem, sejam atacados, invadidos e controlados por adversários.

Uma das lacunas que ficou para ser preenchida no DEPSPACE é o suporte a transações atômicas. Este suporte é importante para a manutenção da consistência das aplicações que usam o espaço de tuplas em caso de falhas nos processos. Transações atômicas (doravante denominada apenas por transação) permitem que uma seqüência de operações no espaço de tuplas seja executada de forma atômica e que os resultados das mesmas sejam refletidos de maneira correta, mantendo o sistema consistente. Este trabalho tem a sua motivação na nossa experiência do GRIDTS [Favarim et al., 2007]. O GRIDTS é uma infra-estrutura, baseada em espaços de tuplas, que provê uma solução para o escalonamento em grades computacionais, onde os recursos selecionam as tarefas mais apropriadas para suas condições de execução. No GRIDTS, tuplas descrevem as tarefas que compõem a aplicação do usuário e são colocadas, através do *broker*, no espaço de tuplas para serem escalonadas. Os recursos computacionais da grade recuperam do espaço as tuplas que descrevem tarefas que os mesmos são capazes de executar. Um recurso, ao término de uma tarefa, deve colocar no espaço de tuplas o resultado deste processamento. Este resultado fica disponível, através do *broker*, para o usuário que submeteu a aplicação à grade. A necessidade de um suporte de transações era premente no GRIDTS. Se um recurso que estivesse executando uma tarefa falhasse, a tupla que descrevia a tarefa era perdida e não podia ser recuperada de modo que outro recurso pudesse executá-la. O uso do mecanismo de transações garante que, em caso de falha do recurso, a tupla descrevendo a tarefa seja recuperada, permitindo que outro recurso venha a executá-la.

A preocupação então, neste artigo, é descrever nossos esforços na proposição de um modelo de transações para ser integrado a espaços de tuplas confiáveis e seguros como o DEPSPACE. Esta nossa experiência não está muito distante das preocupações que nortearam outras integrações de transações em outras implementações de espaços de tuplas: JavaSpaces [Sun Microsystems, 2003] e TSpaces [Lehman et al., 2001]. Porém, o que difere a nossa proposta das citadas é que, no modelo proposto, garantimos as propriedades ACID (Atomicidade, Consistência, Isolamento, Durabilidade) das transações, em ambientes mais severos, como os sujeitos a falhas maliciosas (falhas bizantinas). O texto descreve, além do modelo e sua integração ao DEPSPACE, a avaliação do seu desempenho e aspectos de sua implementação. Apresentamos também a descrição de um teste realizado envolvendo uma aplicação em grades computacionais. Apesar do modelo ser focado em espaços de tuplas, através do mesmo podemos também tirar importantes conclusões sobre os requisitos necessários para a integração de transações em serviços confiáveis baseados na técnica de replicação Máquina de Estados [Schneider, 1990].

## 2. Espaço de Tuplas com Segurança de Funcionamento

Um espaço de tuplas pode ser visto como um objeto de memória compartilhada que permite a interação entre processos distribuídos [Gelernter, 1985]. Neste espaço, estruturas de dados genéricas chamadas de **tuplas**, podem ser inseridas, lidas e removidas. Uma

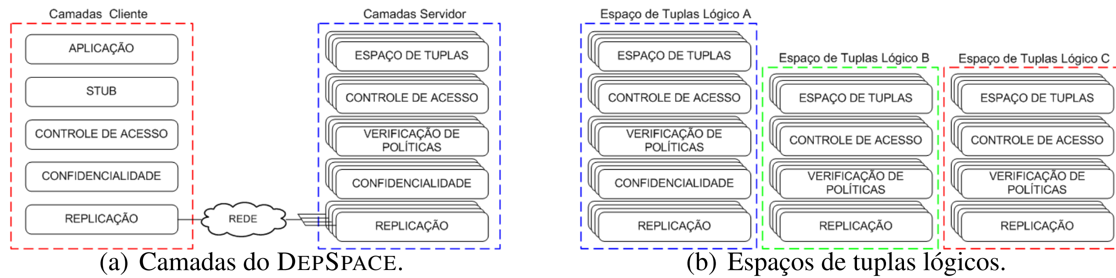
tupla $t$ é uma seqüência ordenada de campos $\langle f_1 ... f_n \rangle$, onde cada campo $f_i$ que contém um valor é dito *definido*. Uma tupla onde todos os campos são definidos é chamada de **entrada**. Uma tupla $\bar{t}$ é chamada de **molde** se alguns de seus campos não possuem valores definidos. Um espaço de tuplas somente pode armazenar entradas, nunca moldes. Os moldes são usados para acessar as tuplas do espaço. Diz-se que uma entrada $t$ e um molde $\bar{t}$ **combinam** se e somente se: $(i)$ ambos têm o mesmo número de campos, e $(ii)$ todos os valores dos campos definidos em $t$ possuem o mesmo valor dos campos correspondentes em $\bar{t}$. Por exemplo, uma tupla $\langle \text{SBRC}, \text{Rio}, 2008 \rangle$ combina com o molde $\langle \text{SBRC}, *, 2008 \rangle$ (onde '$*$' denota um campo não definido do molde).

Um espaço de tuplas provê três operações básicas [Gelernter, 1985]: $out(t)$ que adiciona uma tupla $t$ no espaço de tuplas; $in(\bar{t})$ que lê e remove, do espaço de tuplas, uma tupla $t$ que combine com o molde $\bar{t}$; e $rd(\bar{t})$ que tem um comportamento similar ao da operação $in$, mas que somente faz a leitura da tupla combinando com $\bar{t}$, sem removê-la do espaço. As operações $in$ e $rd$ são bloqueantes, i.e., se nenhuma tupla que combine com o molde $\bar{t}$ está disponível no espaço de tuplas, o processo fica bloqueado até que uma se faça disponível. Uma extensão típica deste modelo é a provisão de variantes não bloqueantes das operações de leitura: $inp$ e $rdp$ [Gelernter, 1985]. Estas operações funcionam exatamente como suas versões bloqueantes, a não ser pelo fato que retornam um valor de erro quando uma tupla que combine com o molde não esteja disponível no espaço de tuplas. Uma característica importante do espaço de tuplas é a natureza associativa do acesso: as tuplas não são acessadas por um endereço ou identificador, mas sim pelo seu conteúdo.

### 2.1. DEPSPACE: Um Espaço de Tuplas com Segurança de Funcionamento

Um espaço de tuplas é dito seguro e confiável caso suporte os seguintes atributos de segurança de funcionamento (*Dependability* [Avizienis et al., 2004]): **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; **confidencialidade** – o conteúdo de campos de uma tupla não pode ser revelado a partes não autorizadas. O DEPSPACE é a implementação de um espaço de tuplas confiável e seguro, que satisfaz estes atributos de segurança de funcionamento através da combinação de diversos mecanismos, tais como **replicação tolerante a faltas bizantinas** (seguindo o paradigma da replicação Máquina de Estados [Schneider, 1990, Castro and Liskov, 2002]) e **controle de acesso** (por credenciais e através da instalação de políticas de granularidade fina [Bessani et al., 2006]). Para mais detalhes sobre o projeto e implementação do DEPSPACE, ver [Bessani et al., 2008].

O DEPSPACE é organizado em camadas, onde cada uma destas provê uma funcionalidade diferente. A arquitetura do DEPSPACE é apresentada na Figura 1(a). Nesta figura, no topo da pilha do cliente está a aplicação (que acessa o espaço) e no servidor, no topo da pilha, está uma implementação local de um espaço de tuplas. No cliente, a estratificação apresenta ainda as camadas de controle de acesso, de confidencialidade e de replicação. No lado do servidor a arquitetura é similar, existindo ainda uma camada responsável pela verificação de políticas de granularidade fina. A aplicação interage com o sistema invocando as operações disponíveis através da camada *stub*. Esta camada provê as assinaturas usuais das operações que podem ser realizadas no espaço de tuplas ($in$, $out$,...), permitindo o acesso transparente ao espaço tuplas replicado. Um aspecto im-




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

**Figura 1. Características do DEPSPACE.**

portante do serviço oferecido pelo DEPSPACE é o suporte a múltiplos espaços de tuplas lógicos, permitindo que no sistema existam vários espaços de tuplas sem nenhuma relação uns com os outros. Para cada espaço espaço de tuplas lógico ativado no DEPSPACE pode-se definir quais camadas devem estar ativas. Com exceção da camada de replicação, que é obrigatória, todas as outras são opcionais. A Figura 1(b) ilustra espaços lógicos.

## 3. Modelo de Sistema

O sistema é formado por um conjunto ilimitado de clientes e $n$ servidores que implementam o espaço de tuplas seguro e confiável (DEPSPACE). Qualquer número de clientes e até $f$ servidores podem estar sujeitos a **faltas bizantinas**, ou seja, podem desviar de suas especificações de maneira arbitrária [Lamport et al., 1982]. Faz-se necessário no mínimo $n \geq 3f + 1$ servidores (réplicas) para tolerar $f$ servidores faltosos, de maneira a prover um espaço de tuplas com os atributos já mencionados. No modelo, também consideramos um processo que nunca falha como **correto**, enquanto um processo que falhe é dito ser **faltoso**. É assumido independência de faltas para os servidores: a probabilidade de um servidor sofrer uma falha é independente da probabilidade de outro falhando. Esta propriedade pode ser alcançada através do uso sistemático de diversidade [Obelheiro et al., 2005]. As comunicações entre clientes e servidores é feita através de canais ponto-a-ponto confiáveis e autenticados. Estes canais podem ser implementados através de SSL/TLS [Dierks and Allen, ].

É assumido que o sistema apresenta **sincronismo terminal** [Dwork et al., 1988]. Esta premissa é necessária pelo fato do DEPSPACE fazer uso de uma primitiva de difusão com ordem total tolerante a faltas bizantinas, baseada no algoritmo de consenso Paxos Bizantino [Castro and Liskov, 2002]. O sincronismo terminal é importante também para os mecanismos de suporte a transações, os quais somente podem levar a bom termo as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) [Haerder and Reuter, 1983] nos momentos em que o sistema apresenta comportamento síncrono. Por simplicidade, assumimos o modelo de transações planas (somente em um nível), deste modo, transações aninhadas não são tratadas neste texto.

## 4. Modelo de Transações no DEPSPACE

### 4.1. Aspectos de Organização do Modelo

**Transações em Espaços de Tuplas.** O conceito de transações surgiu em sistemas de gerenciamento de banco de dados [Gray, 1978] e foi estendido, posteriormente, para linguagens de coordenação [Jeong and Shasha, 1994]. Implementações de espaços de tuplas modernas, como o JavaSpaces [Sun Microsystems, 2003] e o TSpaces [Lehman et al., 2001] provêem suporte a transações.

Uma transação atômica é uma abstração que assegura a execução atômica de uma seqüência de operações sobre um sistema (neste caso, o espaço de tuplas), garantindo que

ou todas as operações da transação são refletidas corretamente no sistema (a transação é confirmada) ou nenhuma o será (a transação é cancelada ou por falha no processo de execução de suas operações ou pelo cancelamento espontâneo pelo cliente. Para garantir a consistência do espaço de tuplas uma transação deve satisfazer as propriedades ACID [Haerder and Reuter, 1983]. Portanto, uma transação no espaço de tuplas deve também ser uma seqüência de operações que leva o espaço de tuplas de um estado consiste para um outro estado também consistente.

As transações podem ser executadas de forma concorrente, desde que sejam serializáveis, i.e., suas operações podem ser executadas seguindo alguma ordem legal das operações, tendo o mesmo efeito final de execução seqüencial das transações. A garantia de que a transação não viola a propriedade do isolamento (serialização) é provida através de mecanismos de controle de concorrência no espaço de tuplas. Entre os mecanismos de controle de concorrência existentes na literatura, assumimos o modelo pessimista com a concorrência baseado em bloqueios (ou *locks*) [Gray, 1978]. Este modelo também é utilizado em outros trabalhos relacionados a espaços de tuplas [Lehman et al., 2001, Sun Microsystems, 2003]. Deste modo, o controle de concorrência pessimista do acesso a tuplas no espaço por transações concorrentes é provido através da redefinição da semântica das operações no espaço quando executadas dentro de uma transação:

- $out$: uma tupla inserida só fica visível no espaço de tuplas para outros processos após a transação ser confirmada. No entanto, logo após a execução do $out$, a tupla fica visível para o processo dentro da transação. Se esta tupla for removida na própria transação, então a mesma não será adicionada no espaço quando a sua transação for confirmada. O mesmo acontece se a transação for cancelada.
- $rd$ e $rdp$: a leitura de uma tupla pode se dar logo após sua inserção, tanto no contexto da própria transação, quanto no espaço de tuplas. Uma tupla lida do espaço de tuplas em uma transação pode também ser lida por outras transações concorrentes. Mas, a tupla lida pela transação não pode ser removida até que esta transação seja confirmada ou cancelada. A operação $rd$ sempre aguarda até que uma tupla combinando com o molde esteja disponível (seja por uma nova tupla inserida no espaço ou pela resolução do conflito com outra transação). A operação $rdp$ somente irá aguardar por uma tupla que combine com o molde em caso de conflito com outras transações.
- $in$ e $inp$: estas operações removem tuplas escritas tanto no contexto da transação, quanto no espaço de tuplas. Uma tupla removida do espaço de tuplas numa transação não pode ser lida e nem removida por outras transações concorrentes. Se a transação for cancelada, a tupla pode voltar a ser lida ou removida por outras transações. De maneira semelhante à $rd$ e $rdp$, as operações $in$ e $inp$ somente diferem na maneira que se comportam quando não existem tuplas disponíveis que combinam com seus moldes. A operação $in$ aguarda até que uma tupla combinando com o molde esteja disponível (pela inserção de uma nova tupla no espaço ou pela resolução do conflito com outra transação). A operação $inp$ somente irá aguardar por uma tupla que combine com o molde em caso de conflito com outras transações. Neste caso, a tupla pode estar sendo lida ou, mesmo removida por outras transações.

**Integração do Modelo à Arquitetura do DepSpace.** O suporte a transações no DEPSPACE é provido através da inclusão de uma camada adicional no lado servidor, cons-

tituindo o que chamamos de camada de Transação. No lado do cliente, a inclusão se dá no nível *stub* e corresponde a uma extensão da visão do cliente, acrescentando às operações iniciais do DEPSPACE algumas operações para gestão de transações (Figura 3). A arquitetura do DEPSPACE com a camada de transação proposta é apresentada na Figura 2.

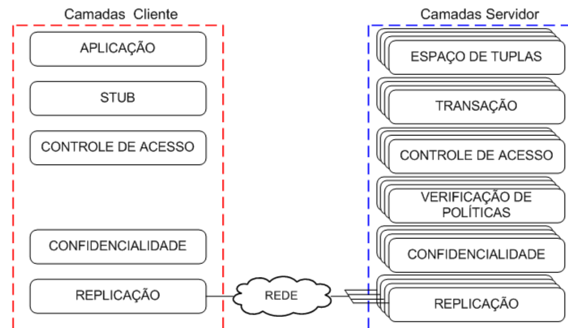



**Figura 2. Arquitetura do DepSpace com a camada de transações.**

Cada transação é criada e gerenciada pela camada de transação, a qual implementa as operações mostradas na Figura 3. A aplicação invoca a operação $beginTransaction$ para criar uma nova transação e um identificador ($TID$) para a mesma é retornado. Através do parâmetro $timeout$, o cliente especifica o tempo de expiração para a transação, i.e., o tempo máximo para a conclusão de uma transação. Mas, nem sempre $timeout$ solicitado é garantido. Isto acontece quando este parâmetro excede um tempo máximo permitido pelo suporte a transações. O valor de retorno $time$ da operação indica o tempo de expiração concedido para a transação. As invocações de todas as operações a serem executadas no contexto da transação criada devem conter o identificador $TID$ da transação.

---

$beginTransaction(timeout) \rightarrow (TID, time)$

Cria uma nova transação, com tempo de expiração $timeout$ e retorna um identificador único para a transação ($TID$) e o tempo $time$ concedido para a transação. O identificador é usado para identificar quais operações fazem parte desta transação.

$closeTransaction(TID) \rightarrow (commit\ ou\ abort)$

finaliza a transação, retornando o valor $commit$ se a transação foi confirmada ou o valor $abort$ se a transação não pode ser confirmada, sendo então cancelada.

$abortTransaction(TID) \rightarrow (true\ ou\ false)$

cancela a transação, retornando o valor $true$ se a transação foi cancelada ou o valor $false$ se a transação já havia sido confirmada ou cancelada.

$renewTransactionTimeout(TID, timeout) \rightarrow (time)$

renova o tempo de *timeout*, retornando o valor $time$ indicando o tempo que foi concedido para renovação, sendo $0 \leq time \leq timeout$, se o tempo for zero indica que foi impossível fazer a renovação do *timeout*.

---

**Figura 3. Operações de Transações em um Espaço de Tuplas.**

Ao final da transação, a aplicação invoca a operação $closeTransaction$ indicando a sua conclusão. Se a transação puder concluir normalmente, as modificações realizadas pelas operações dentro da transação são efetivadas e o valor de retorno da operação $closeTransaction$ é $commit$. No entanto, uma transação pode ser cancelada devido a várias razões. Por exemplo, devido a conflitos com outras transações, no caso de controle de concorrência otimista [Kung and Robinson, 1981], os resultados das operações da transação devem ser abandonados e, neste caso, o valor de retorno da operação $closeTransaction$ é $abort$.

Se, por alguma razão, a aplicação desejar cancelar uma transação, a operação $abortTransaction$ deve ser invocada. A transação também pode ser cancelada por decisão de um conjunto de servidores. Este caso acontece quando um cliente falha ou demora muito para confirmar a transação. O cliente no momento da criação da transação especifica um tempo de expiração para a transação, caso a mesma não seja concluída antes do tempo de expiração, o servidor então deve cancelá-la. Quando a transação é cancelada, seja por decisão do cliente ou de um servidor, todos os servidores asseguram que as modificações realizadas pelas operações dentro da transação são desfeitas. Uma transação também pode ter seu tempo de expiração renovado pelo cliente através da operação $renewTransactionTime$.

### 4.2. Gerenciamento da Execução das Transações

Nesta seção exploramos os principais aspectos relacionados à dinâmica da camada de transação. São apresentados os detalhes relacionados ao gerenciamento da concorrência, isto é, como são executadas as operações no espaço de tuplas de modo a garantir a semântica exposta das mesmas na Seção 4.1. Também é apresentado o efeito, no espaço de tuplas, das operações de controle de transações descritas a partir da Figura 3.

Do ponto de vista da aplicação, uma transação sobre as réplicas do espaço de tuplas deve aparecer como se estivesse sendo executada em um sistema não replicado. Uma propriedade fundamental para transações em servidores replicados é a *one-copy serializability* [Davidson et al., 1985], a qual estabelece que a atuação de uma transação sobre servidores replicados, assim como o controle de concorrência, deve ter o mesmo efeito de que se tivesse sido executada em um único servidor. Esta propriedade é garantida pela camada de replicação da arquitetura do DepSpace, que faz uso da técnica de replicação Máquina de Estados [Schneider, 1990], a qual determina que servidores corretos executam a mesma seqüência de operações e retornam, evoluindo portanto de forma sincronizada. Estas características são garantidas através do uso de um protocolo de difusão total tolerante a faltas bizantinas. Esta difusão é baseada no algoritmo de consenso Paxos Bizantino [Castro and Liskov, 2002].

**Início da transação.** Ao receber uma invoção da operação $beginTransaction(timeout)$, a camada de transação associa um identificador $TID$ para a transação e cria um espaço privado para a mesma. Este espaço contém três conjuntos de tuplas: $tempRd$, que armazena temporariamente as tuplas lidas (operação $rd$ ou $rdp$) no espaço de tuplas até que a transação seja completada ou cancelada; $tempIn$ que armazena as tuplas removidas (operação $in$ ou $inp$) do espaço de tuplas; e $tempOut$ mantém temporariamente as tuplas (operação $out$) a serem inseridas no espaço de tuplas. Estes conjuntos são utilizados para garantir o controle de concorrência.

**Execução de operações em transações.** A camada de transação controla a execução das operações no espaço de tuplas de modo a garantir as semânticas das operações expostas na seção 4.1. Este controle é efetuado através da utilização dos conjuntos apresentados no parágrafo anterior: $tempRd$, $tempIn$ e $tempOut$:

- **Inserção ($out$):** quando uma nova tupla é inserida, esta fica armazenada em $tempOut$ até que a sua transação seja concluída. Na confirmação da transação, as tuplas existentes em $tempOut$ são efetivamente inseridas no espaço de tuplas. Se a transação é cancelada, as tuplas em $tempOut$ são descartadas.
- **Leitura ($rd$ ou $rdp$):** ao ser lida, a tupla é removida do espaço de tuplas, inserida no conjunto $tempRd$ da transação e enviada ao cliente. Quando a transação

é confirmada ou cancelada, as tuplas existentes em $tempRd$ são inseridas (devolvidas) no espaço de tuplas. Na operação de leitura, o servidor faz a busca por tuplas na seguinte ordem: $(1)$ procura por tuplas no conjunto $tempRd$ e $tempOut$, verificando a possibilidade de ler uma tupla que já foi lida ou escrita na mesma transação, se nenhuma tupla é encontrada, então, $(2)$ procura por tuplas no espaço de tuplas e, neste caso, se nenhuma tupla é encontrada, então, $(3)$ procura por tuplas nos conjuntos $tempRd$ de todas as outras transações em execução. Ao encontrar tuplas nos conjuntos $tempRd$ de outras transações, faz a leitura da tupla $t$ de um destes conjuntos ($tempRd1$), retorna ao cliente e sinaliza em $tempRd1$ que a tupla $t$ foi lida pela transação $X$. Quando a transação referente a $tempRd1$ terminar, a tupla $t$ lida de $tempRd1$ é transferida para $tempRd$ da transação $X$. Caso a transação $X$ também tenha sido concluída, a tupla $t$ é inserida no espaço de tuplas. Caso não encontre nenhuma tupla, $rd$ e $rdp$ se comportam de maneiras distintas:

- Na operação $rd$ o processo fica aguardando aparecer uma tupla no espaço que combine com o molde passado.
- Na operação $rdp$, procura-se por tuplas no conjunto $tempIn$ de todas as outras transações em execução. A não ocorrência de tupla que combine com o molde em nenhum conjunto $tempIn$ retorna um valor de erro. A existência de tuplas que combinem com o molde nos conjuntos $tempIn$ faz com que a transação da operação $rdp$ fique bloqueada até que todas as transações referentes a estes conjuntos sejam confirmadas, para então retornar um valor de erro. Caso alguma destas transações seja cancelada, o processo tenta ler a tupla no espaço, caso não consiga continua aguardando um nova tupla no espaço.

- **Remoção ($in$ ou $inp$):** quando uma tupla é removida, esta é colocada no conjunto $tempIn$ de sua transação e enviada ao cliente. Se a transação é cancelada, as tuplas existentes em $tempIn$ são devolvidas ao espaço de tuplas. Na confirmação da transação, as tuplas existentes em seu $tempIn$ são descartadas. A busca por tuplas de um servidor, na operação de remoção, segue a seguinte ordem: $(1)$ procura por tuplas no seu conjunto $tempRd$ e $tempOut$, verificando a possibilidade de remover uma tupla que já foi lida ou escrita na mesma transação, se nenhuma tupla é encontrada, então, $(2)$ faz a busca no espaço de tuplas. Caso não encontre nenhuma tupla, $in$ e $inp$ se comportam de maneiras distintas:
  - Na operação $in$ o processo fica aguardando aparecer uma tupla no espaço que combine com o molde passado pela operação.
  - Na operação $inp$ procura-se por tuplas nos conjuntos $tempRd$ e $tempIn$ de todas as outras transações em execução. A não existência de tupla que combine com o molde em nenhum destes conjuntos retorna um valor de erro. A ocorrência de tuplas que combinem com o molde nos conjuntos $tempRd$ faz com que a transação da operação $inp$ fique bloqueada até que alguma das transações referentes a estes conjuntos sejam confirmadas ou canceladas, para então ter acesso a tupla. Neste caso, a transação executando $inp$ é ativada e tenta remover a tupla no espaço, caso não consiga continua aguardando. A ocorrência de tuplas que combinem com o molde nos conjuntos $tempIn$ faz com que o processo fique aguardando até que todas as transações referente a estes conjuntos sejam confirmadas antes de retornar um valor de erro. Caso alguma das transações destes conjuntos

$tempIn$ seja cancelada, o processo tenta remover a tupla do espaço, caso não consiga continua aguardando. Enquanto está aguardando, se aparecer uma nova tupla no espaço, a operação $inp$ pode ter acesso à mesma, sendo concluída com a remoção desta tupla.

Com estes conjuntos e os algoritmos envolvidos na execução das operações do espaço de tuplas, uma tupla lida por uma transação $X$ não pode ser removida por nenhuma outra transação $Y$, até que a transação $X$ tenha sido confirmada ou cancelada. Porém, é permitido que a transação $Y$ leia a mesma tupla que $X$ leu. Além disso, o uso destes três conjuntos garante a propriedade de atomicidade e isolamento das transações. Por exemplo, uma tupla inserida na transação $X$ só será visível por outra $Y$ após a transação $X$ ser confirmada, ou seja, após mover as tuplas de seu conjunto interno $tempOut$ para o espaço de tuplas.

**Confirmação da transação.** Quando a camada de transação de um servidor correto recebe do cliente a invocação $closeTransaction$, os resultados das operações realizadas no contexto da transação correspondente são efetivadas no espaço de tuplas. Esta efetivação inclui inserir no espaço as tuplas mantidas nos conjuntos da transação ($tempOut$ e $tempRd$). Ao inserir as tuplas mantidas em $tempRd$ no espaço, possíveis bloqueios sobre estas são desfeitos. As tuplas contidas em $tempIn$ são descartadas.

No DEPSPACE um servidor malicioso pode cancelar uma transação mesmo tendo recebido a requisição para confirmar a mesma. Com este comportamento, o servidor malicioso só consegue danificar o seu próprio estado pois, com no mínimo $n - f \geq 2f + 1$ servidores corretos processando a confirmação da transação, a continuidade correta do serviço está garantida. Além disso, mesmo que um cliente malicioso envie uma requisição para apenas alguns servidores, ou requisições conflitantes para diferentes servidores, este não conseguirá corromper os mesmos, provocando o cancelamento da transação por alguns servidores enquanto outros confirmam a mesma. Esta propriedade é garantida pela camada de replicação (através do protocolo Paxos Bizantino), a qual provê difusão atômica tolerante a faltas bizantinas [Castro and Liskov, 2002].

**Cancelamento da transação.** Uma transação pode ser cancelada tanto pela decisão do cliente ou do servidor. Quando um cliente invoca a operação $abortTransaction$, a camada de replicação do DEPSPACE (através do uso de difusão atômica) garante que todas as réplicas corretas do espaço de tuplas receberão e executarão esta operação, assim como acontece na confirmação da transação. Um servidor pode iniciar o processo de cancelamento de uma transação quando o $timeout$ da mesma expira.

A detecção da expiração do $timeout$ de uma transação por um servidor não implica o cancelamento imediato da mesma. Devido à inexistência de um relógio global único, a expiração do $timeout$ nos diferentes servidores poderá ser percebida em momentos diferentes nos mesmos, e portanto, faz-se necessário que todos os servidores corretos também tenham detectado o esgotamento do prazo da transação, para então efetivarem o seu cancelamento. Para a concretização deste procedimento os servidores detectores enviam, usando difusão atômica, para as outras réplicas a requisição solicitando o cancelamento da transação. Cada servidor deve esperar o recebimento de $f + 1$ mensagens de solicitação de cancelamentos para uma transação, garantindo assim a presença de pelo menos um servidor correto nesta decisão. Com este limite, o sistema tolera um conluio de até $f$ servidores maliciosos, sem que estes consigam cancelar uma transação não cancelada e ainda não expirada.

O uso da difusão atômica e a espera do quórum de $f+1$ servidores garante que, ou todos os processos corretos cancelam uma transação, ou nenhum cancela. Isto ocorre pois, se um processo correto cancela uma transação devido a um $timeout$, isto implica que ele recebeu mensagens de cancelamento de $f+1$ réplicas antes de receber uma mensagem com a requisição de $closeTransaction$ do processo cliente. Como todas as mensagens são entregues as réplicas na mesma ordem a todas as réplicas corretas (devido à difusão atômica [Castro and Liskov, 2002]), a ação tomada por uma réplica correta se repete em todas as réplicas corretas.

Um servidor malicioso pode também confirmar uma transação mesmo tendo recebido requisição para cancelar a mesma. Com esta decisão, este servidor malicioso só pode danificar o seu próprio estado. Assim como acontece na confirmação de uma transação, haverá no mínimo outros $2f+1$ servidores corretos que processam o cancelamento da transação e estão aptos a continuar provendo corretamente o serviço.

No momento em que uma transação é cancelada devido à ação dos servidores ou uma requisição $abortTransaction$ do cliente, a camada de transação deste desfaz as modificações realizadas pelas operações encapsuladas na transação. Este procedimento corresponde a inserir no espaço de tuplas aquelas tuplas mantidas nos conjuntos $tempIn$ e $tempRd$ da transação. Ao inserir as mesmas no espaço, eventuais bloqueios sobre algumas destas tuplas são desfeitos, permitindo que outras transações possam ler ou remover estas tuplas do espaço. As tuplas contidas em $tempOut$ são descartadas.

**Usando Timeouts.** O parâmetro $timeout$ da operação $beginTransaction$ expressa o tempo para a conclusão da transação segundo o desejo do cliente (ver Figura 3). O sistema confirma este valor através do parâmetro de retorno $time$. Porém, este $timeout$ pode não ser atendido pela camada de transação. Para isto, basta que o $timeout$ pedido supere o tempo máximo permitido pelo espaço de tuplas. Neste caso, o cliente terá no parâmetro de retorno $time$ a expressão deste máximo que a camada pode suportar. Este controle sobre os $timeouts$ especificados visa evitar que clientes maliciosos criem transações e nunca façam a confirmação das mesmas, impedindo que outros clientes acessem as tuplas bloqueadas nestas transações "falsas". O cliente também pode renovar o tempo concedido (através do parâmetro de retorno $time$) pelo suporte para a conclusão de sua transação através da operação $renewTransactionTime$. As concessões de renovação do tempo de expiração também estão limitadas na sua soma pelo tempo máximo permitido pelo espaço de tuplas.

## 5. Avaliação Experimental

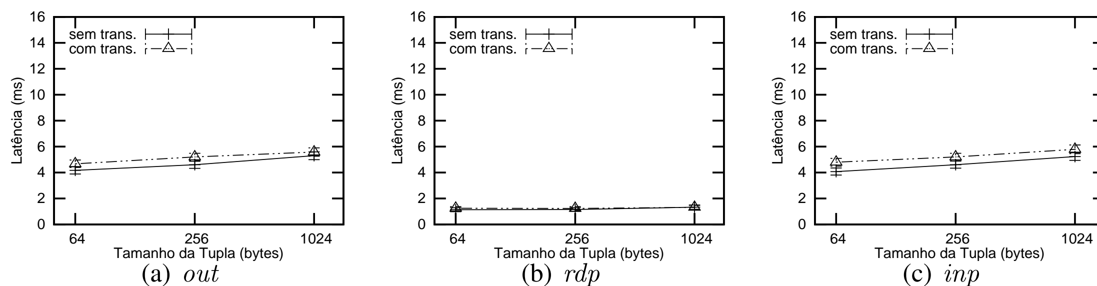
O modelo de transações proposto foi implementado na linguagem de programação Java e integrado a implementação do DEPSPACE disponível[1] [Bessani et al., 2008]. Devido à limitação de espaço, os detalhes de implementação foram omitidos neste texto.

**Configuração do ambiente.** Os experimentos foram realizados no Emulab [White et al., 2002], em um ambiente consistindo de 13 máquinas pc3000 (3.0Ghz Pentium Xeon com 2Gb de memória e interface de rede gigabit) conectadas a um *switch* gigabit. A rede local é emulada como uma VLAN em um *switch* Cisco 4509 com latência próxima a zero. O ambiente de *software* instalado nas máquinas foi o S.O. Had Hat Linux 6 com *kernel* 2.4.20 e máquina virtual Java de 32 bits versão 1.6.0_02. Em todos os experimentos, as camadas de controle de acesso, de verificação de políticas e de confidencialidade do DEPSPACE foram desabilitadas. Deste modo, duas configurações foram

[1] Disponível em `http://www.navigators.di.fc.ul.pt/software/depspace/`

criadas para os experimentos: uma com a camada de transação ativada e outra com esta camada desativada. Em todos os experimentos o DEPSPACE foi replicado em 4 servidores (tolerando 1 falha bizantina).

**Custo da camada de transações.** O primeiro experimento avalia o custo da adição de transações no DEPSPACE. Este custo foi mensurado através da latência média observada na execução de operações sobre o espaço de tuplas na instância do DEPSPACE com e sem a camada de transação. Todos os valores reportados aqui compreendem o tempo médio necessário para a execução de uma operação por um cliente do sistema (situado em uma máquina diferente dos servidores), recolhido a partir de 1000 execuções da operação e excluindo-se os 5% dos valores com maior desvio. A Figura 4 apresenta a latência média para a execução de três operações ($out$, $rdp$, $inp$) no espaço de tuplas, variando-se o tamanho das tuplas, com o suporte a transações ativado e desativado. Quando ativada a camada, para cada operação, as 1000 execuções foram realizadas dentro do contexto de uma transação.



(a) $out$ (b) $rdp$ (c) $inp$

**Figura 4. Custo da adição do suporte a transações no DEPSPACE.**

Os resultados apresentados na Figura 4 mostram que as operações executadas dentro de um transação incorrem em pouquíssimo aumento da latência se comparadas com as mesmas executadas sem o suporte de transações. Além disso, pode-se perceber que a latência apresentou um crescimento suave com o aumento do tamanho das tuplas.

**Custo das operações de gestão de transações.** Na realização do primeiro experimento, foi observado que os maiores custos no uso de transações estão nas operações de início ($beginTransaction$), confirmação ($closeTransaction$) e cancelamento ($abortTransaction$) das mesmas. Para comprovar este custo adicional, foi realizado um experimento adicional, onde foram criadas várias transações com configurações diferentes, ou seja, em cada transação foi executada uma seqüência com um número diferente de operações sobre o espaço de tuplas. Nestes experimentos, tivemos a operação $abortTransaction$ invocada sempre no mesmo ponto em que ativávamos a operação $closeTransaction$.

Com estes experimentos, foi constatado que o custo da operação $beginTransaction$ apresenta um tempo fixo de aproximadamente 20ms. Este valor se justifica pelo tempo gasto para criar as estruturas de controle da mesma e o seu contexto. Já os custos nas operações $closeTransaction$ e $abortTransaction$ variam com cada transação, pois dependem do conjunto de operações executadas sobre o espaço de tuplas no contexto da transação considerada e do tamanho dos conjuntos de tuplas que devem retornar para o espaço de tuplas. Para ambas operações, foram observados tempos variando entre o mínimo de 8 ms e o máximo de 21 ms. Os custos relacionados a estas últimas operações refletem, portanto, as implicações do controle de concorrência e a dimensão das transações.

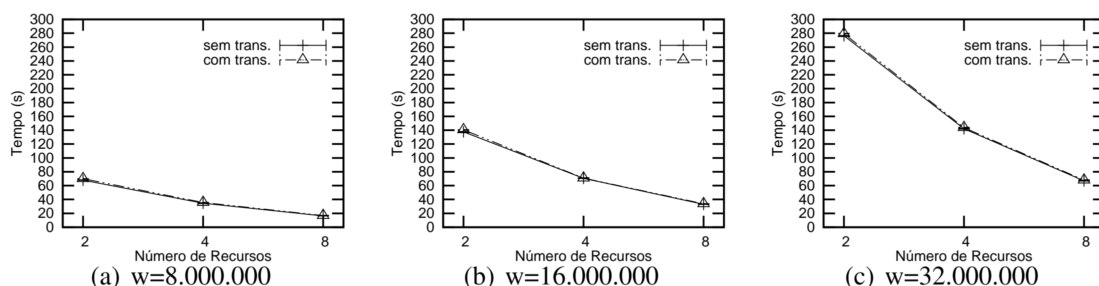
**Custo das transações em uma aplicação em grade.** O último experimento avalia o

custo percebido por uma aplicação de grade computacional que faz uso de uma instância DEPSPACE incluindo a camada de transações. A aplicação escolhida foi a quebra através de força bruta de uma chave gerada pelo algoritmo RC5 [Rivest, 1995]. A aplicação é decomposta em um conjunto de tarefas onde cada uma destas contém um subespaço com $w$ chaves do total de chaves possíveis. A chave correta se encontra no meio de um subespaço de chaves. Por exemplo, considerando que a chave correta se encontra na $29^{\underline{a}}$ tarefa e que a busca é feita na ordem de geração das tarefas, um único recurso precisaria executar $28,5$ tarefas até encontrar a chave correta.

A distribuição das tarefas da aplicação foi feita segundo o modelo de escalonamento em grades apresentado em [Favarim et al., 2007]. No caso da aplicação citada, o *broker* insere $ck$ tarefas no espaço de tuplas, sendo que a $ck$-ésima tarefa contém a chave correta. Após isto, o *broker* fica aguardando a chave ser encontrada. Cada recurso computacional da grade recupera do espaço as tuplas de tarefas que descrevem a procura no espaço de chaves. Se um recurso encontra a chave correta através de sua tarefa, a tupla de resultado a ser inserida no espaço, contém a chave correta e a identificação da tarefa onde a mesma foi encontrada. Caso não encontre a chave, o recurso insere, uma tupla de resultado contendo a identificação da tarefa e a informação que a chave não foi encontrada.

Quando usado o suporte a transações, o *broker* executa uma transação que consiste da inserção de todas as tarefas no espaço de tuplas; os recursos por sua vez, executam transações que consistem na remoção de uma tarefa do espaço e que finalizam com o término da tarefa e a inserção da tupla de resultado (com ou sem a chave) no espaço de tuplas.

A Figura 5 mostra o tempo necessário para encontrar a chave correta, variando o número $r$ de recursos e a quantidade $w$ de chaves em cada sub-espaço (tarefa), com e sem o uso do suporte a transações no DEPSPACE. Os experimentos realizados consideram $ck = 29$. Os valores reportados aqui compreendem o tempo médio recolhido a partir de 10 execuções.




(a) w=8.000.000 (b) w=16.000.000 (c) w=32.000.000

**Figura 5. Tempo médio de execução para encontrar a chave RC5.**

Como pode ser observado na Figura 5, o uso de transações causa um aumento desprezível no custo da execução da aplicação em relação a instância do espaço que tem a camada de transação desabilitada. Além disso, o uso de transações traz benefícios a execução da aplicação, agregando a qualidade de tolerante a faltas. Neste experimento, por exemplo, se o recurso falhasse e a tarefa executada pelo mesmo fosse justamente a que continha a chave, esta chave nunca seria encontrada sem o suporte de transações descrito neste trabalho [Favarim et al., 2007].

## 6. Trabalhos Relacionados

Alguns trabalhos surgiram a fim de prover suporte a execução atômica de uma seqüência de operações em espaços de tuplas, FT-Linda [Bakken and Schlichting, 1995] e PLinda [Jeong and Shasha, 1994] são alguns exemplos destes esforços.

O FT-Linda [Bakken and Schlichting, 1995], apresenta uma forma restrita de execução atômica de operações através do uso de *atomic guarded statments* (AGS). Esta construção consiste basicamente em mandar uma série de operações a serem executadas de forma atômica no espaço de tuplas. O modelo de AGS é bastante restrito quando comparado com um suporte a transações na medida em que computações arbitrárias não podem ser executadas entre as operações de um AGS. O FT-Linda provê tolerância a faltas dos espaços de tuplas através do uso da técnica de replicação Máquina de Estados, no entanto, esta tolerância é restrita a faltas por parada. O modelo de transações introduzido no presente trabalho, além de não ter as limitações do AGS, tolera comportamentos maliciosos tanto dos clientes quanto dos servidores.

PLinda [Jeong and Shasha, 1994] introduz o conceito de transações em espaço de tuplas, assegurando a execução atômica de todas das operações no espaço de tuplas que são delimitadas pelas operações *xstart* e *xcommit*. O PLinda provê tolerância a faltas do espaço de tuplas através do armazenamento periódico (*checkpointing*) de todo espaço em memória persistente. Além disso, PLinda garante que somente armazena as transações já confirmadas de modo que as informações de tuplas mantidas no armazenamento persistente definem sempre em um estado consistente do espaço de tuplas.

O modelo de transações suportado por espaços de tuplas modernos como o JavaSpaces [Sun Microsystems, 2003] e do TSpaces [Lehman et al., 2001] são muito similares ao usado neste artigo, porém, nestes sistemas a implementação do modelo é simplificada pelo fato de não suportarem replicação, não sendo portanto tolerantes a faltas, e nem tampouco a possibilidade do espaço ser acessado por processos maliciosos.

A característica fulcral que diferencia significativamente o trabalho apresentado neste artigo em relação a estas experiências anteriores com transações em espaços de tuplas reside no fato de em nosso trabalho são tratados os aspectos referentes a aplicação do modelo de transações em espaços de tuplas replicados segundo a técnica de replicação Máquina de Estados. Isto nos permite integrar mecanismos de segurança e de tolerância a falhas de uma maneira única e inovadora. A garantia das propriedades ACID nas nossas transações envolve ambientes mais severos: as transações do DEPSPACE evoluem mesmo na presença de falhas maliciosas (bizantinas). Até onde sabemos, na literatura não existem experiências similares.

## 7. Conclusão

Este trabalho apresentou um modelo de transações em espaço de tuplas seguro e confiável. O modelo foi concebido para ser integrado como uma nova camada à arquitetura do DEPSPACE, uma implementação de um espaço de tuplas que já possuía mecanismos de segurança e de tolerância a falhas maliciosas, mas que no entanto não dispunha de facilidades que permitissem a construção de aplicações tolerantes a faltas de maneira simplificada. Com o modelo de transações proposto, asseguramos agora no DEPSPACE também as propriedades ACID de transações em ambientes sujeitos a falhas maliciosas, permitindo que as aplicações tirem proveito da abstração de transações.

A fim de avaliar os custos da integração das transações no DEPSPACE, alguns experimentos foram realizados, demonstrando que, o modelo implica em pouco custo adicional no tempo necessário para execução de uma operação no espaço de tuplas. Os valores mais significativos de latência foram verificados nas operações de criação e de confirmação de uma transação. No entanto, este custo pode ser minimizado se considerarmos que transações podem definir seqüências de operações sobre o espaço de tuplas relativamente grandes, de modo que o tempo nestas computações, dentro de uma

transação, seja predominantemente bem maior que o das operações limites. Isto seria aceitável, principalmente se compararmos a qualidade do serviço adicional que está sendo fornecida através do suporte a transações.

## Referências


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 programming in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 06(3):287–302.

Bessani, A. N., Alchieri, E., Correia, M., and Fraga, J. S. (2008). DepSpace: A Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM/SIGOPS/EuroSys European Conference on Computer Systems (EuroSys 2008)*.

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

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

Davidson, S. B., Garcia-Molina, H., and Skeen, D. (1985). Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370.

Dierks, T. and Allen, C. The TLS Protocol Version 1.0 (RFC 2246).

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

Favarim, F., da Silva Fraga, J., Lung, L. C., and Correia, M. P. (2007). GridTS: A New approach for fault tolerant scheduling in grid computing. In *Proceedings of the 6th IEEE International Symposium on Network Computing and Applications (NCA'2007)*, pages 187–194.

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

Gray, J. (1978). Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481.

Haerder, T. and Reuter, A. (1983). Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317.

Jeong, K. and Shasha, D. (1994). Plinda 2.0: A transactional/checkpointing approach to fault tolerant Linda. In *Proc. of the 13th Symposium on Reliable Distributed Systems*, pages 96–105.

Kung, H. T. and Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Transaction on Database Systems*, 6(2):213–226.

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.

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*.

Rivest, R. L. (1995). The rc5 encryption algorithm. In Preneel, B., editor, *Fast Software Encryption*, volume 1008 of *Lecture Notes in Computer Science*, pages 86–96. Springer.

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

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

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. In *Proc. of 5th Symposium on Operating Systems Design and Implementations*.