Eduardo Adilio Pelinson Alchieri

Uma Infra-Estrutura com Segurança de Funcionamento para Coordenação de Serviços *Web* Cooperantes

FLORIANÓPOLIS
2007

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Uma Infra-Estrutura com Segurança de Funcionamento para Coordenação de Serviços *Web* Cooperantes

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica.

Eduardo Adilio Pelinson Alchieri

Florianópolis, Março de 2007.

Uma Infra-Estrutura com Segurança de Funcionamento para Coordenação de Serviços *Web* Cooperantes

Eduardo Adilio Pelinson Alchieri

'Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica, Área de Concentração em *Controle, Automação e Informática Industrial*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.'

Joni da Silva Fraga
Orientador

Nelson Sadowski
Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

Joni da Silva Fraga
Presidente

Luciano Paschoal Gaspary

Leandro Buss Becker

Ricardo José Rabelo

Aos meus pais Luiz e Aurora, por tudo...
Ao nosso Criador, por tornar tudo isso possível...

AGRADECIMENTOS

Primeiramente gostaria de agradecer a Deus, por tornar tudo isso possível. Agradeço também a minha família, principalmente aos meus pais, Luiz e Aurora, pelo incentivo e apoio que me deram durante a realização deste trabalho, além do exemplo de vida que representam para mim. Tudo que sou e tenho devo a eles.

Agradeço ao meu professor orientador Joni da Silva Fraga e ao doutor Alysson Neves Bessani, por todas as conversas e pela atenção a mim despendida durante o mestrado. Foi grande o aprendizado adquirido com eles durante este período de convivência.

Também quero agradecer aos colegas do DAS, pela grata experiência de tê-los como colegas de trabalho, onde muitas vezes me ajudaram com sugestões e opiniões. Com eles também tive a oportunidade de aprender várias coisas.

Não poderia deixar de agradecer a todos os meus amigos, tanto os de infância quanto os conquistados ao longo da graduação e do mestrado, pela força e parceria. Como todos sabem, amigos são os irmãos que temos a oportunidade de escolher. Poderia preencher várias páginas, mas citarei o nome de apenas três: Darlan, Geremia e Adrian; que estiveram junto comigo nesta caminhada, participando das mais diversas peripécias!!! Sem eles seria muito difícil suportar a distância da família.

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia Elétrica.

Uma Infra-Estrutura com Segurança de Funcionamento para Coordenação de Serviços *Web* Cooperantes

Eduardo Adilio Pelinson Alchieri

Março/2007

Orientador: Joni da Silva Fraga
Área de Concentração: Controle, Automação e Informática Industrial
Palavras-chave: Espaço de Tuplas, Segurança de Funcionamento, Serviços *Web*
Número de Páginas: xiv + 97

Devido ao acentuado aumento no uso da Internet, a maioria dos sistemas distribuídos modernos têm características de sistemas abertos, onde os participantes da computação distribuída são caracterizados pela heterogeneidade, capacidade de computação variável e pela não confiabilidade. Em vista disso, existe uma grande e importante demanda por ferramentas que permitam a construção de aplicações mais complexas de forma eficiente e rápida. Além disso, a segurança de funcionamento é um fator importante relacionado com estas aplicações, principalmente se considerarmos o ambiente no qual tais aplicações irão operar.

Quando se consideram sistemas distribuídos na Internet, a tecnologia dos serviços *web* tem se consolidado cada vez mais como um padrão *de facto*. Esta tecnologia concretiza o modelo de computação orientada a serviços sobre padrões largamente utilizados na *web*, o que proporciona grande facilidade de uso e baixo custo de implantação.

Sendo assim, atualmente tem surgido um grande esforço na especificação de mecanismos de coordenação de serviços *web* para resolução de tarefas que envolvem diversas organizações. Deste modo, estas tarefas são divididas em várias subtarefas que são executadas por vários serviços *web* independentes de forma coordenada, i.e., respeitando-se suas dependências.

Neste sentido, este trabalho apresenta uma infra-estrutura para coordenação de serviços *web* segura e confiável (i.e., tolerante a intrusões), que oferece um elevado grau de desacoplamento. Esta infra-estrutura se baseia no modelo de coordenação por espaço de tuplas e provê uma série de mecanismos de segurança, os quais permitem a coordenação dos serviços *web* mesmo na presença de partes maliciosas. Além disso, esta mesma infra-estrutura pode ser usada como repositório seguro e confiável de dados a serem compartilhados entre serviços *web* cooperantes. As várias fases envolvidas na concepção desta infra-estrutura são abordas neste trabalho, onde as principais decisões tomadas ao longo do projeto deste sistema são destacadas. Além disso, este trabalho investiga os custos envolvidos no uso deste suporte e possíveis aplicações que fazem uso desta infra-estrutura para realizar suas tarefas.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

A Dependable Infrastructure for Coordination of Cooperative Web Services

Eduardo Adilio Pelinson Alchieri

March/2007

Advisor: Joni da Silva Fraga
Area of Concentration: Control, Automation and Industrial Computing
Key words: Tuple Space, Dependability, Web Services
Number of Pages: xiv + 97

Due to the large increase in the use of the Internet, the majority of the modern distributed systems have open systems characteristics. In these systems, the participants of the distributed computation are heterogeneous, with variable computation capacity and untrustworthy. Because of this, there are an important demand for tools that allow the construction of complex applications quickly and efficiently. Moreover, the dependability is an important factor related with these applications, mainly if we consider the environment in which such applications are going to run.

The web service technology is becoming a standard in the development of distributed systems over the Internet. This technology implements the service oriented computation model over standards widely utilized in the web, what provides a large use facility and a low implementation cost.

Because of this, at present has arisen a large effort in the specification of web service coordination mechanisms to address the resolution of tasks that involve multiple organizations. Thus, these tasks are divided in several subtasks that are coordinately performed by several independent web services, i.e., considering the subtasks dependences.

In the same way, this work presents a dependable (i.e, intrusion-tolerant) infrastructure for web service coordination, that offers an elevated range of decoupling. This infrastructure is based on the tuple space coordination model and provides several security mechanisms, which allow the web service coordination even in the presence of malicious parts. Moreover, the same infrastructure can be used as a dependable data repository, that will be shared between cooperative web services. The several phases involved in the conception of this infrastructure are discussed in this work, highlighting the main decisions realized in the system project. Moreover, this work investigates the costs involved in the use of this support and possible applications that use this infrastructure to perform their tasks.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

Devido as evoluções tecnológicas ocorridas nos últimos tempos, a quantidade de usuários da Internet teve um aumento significativo. Desta forma, com a difusão do uso deste ambiente, muitas aplicações começaram a ser projetadas e desenvolvidas para operar na Internet. Estas aplicações tornaram-se cada vez mais complexas, chegando ao ponto de aplicações serem desenvolvidas para a realização de negócios através da Internet.

A maioria dos sistemas distribuídos desenvolvidos para operar neste ambiente têm características de sistemas abertos, onde um número desconhecido de processos, que são executados em ambientes heterogêneos e não confiáveis, interagem através de redes também heterogêneas e não confiáveis. Deste modo, os participantes desta computação distribuída são caracterizados pela heterogeneidade, capacidade de computação variável e pela não confiabilidade.

Além disso, as características exigidas pela maior parte destas novas aplicações, como o anonimato nas comunicações e o desacoplamento, fazem com que o modelo tradicional de comunicação através da troca direta de mensagens entre os participantes da computação distribuída não seja adequando, tornando necessário a adoção de outros modelos de coordenação entre estes participantes.

Outro fator importante, relacionado com estas aplicações, é a necessidade de segurança de funcionamento (*dependability*) [9], principalmente se considerarmos o ambiente no qual tais aplicações são endereçadas e o tipo de negócio que estas aplicações podem envolver. Esta característica está relacionada com a capacidade do sistema sobreviver a falhas em parte de seus componentes. Estas falhas podem ser de natureza acidental (ex. erros de programação e/ou uma falta de parada no servidor) ou maliciosas (ex. ataques bem sucedidos que levam à ocorrência de intrusões no sistema, comprometendo as propriedades de segurança e causando falhas nos componentes invadidos). Desta forma, a tolerância a faltas e intrusões é um requisito de qualidade de serviço fundamental para estas aplicações.

1.1 Motivação

Os padrões e protocolos relacionados com os *Web Services*, ou serviços *web*, estão se tornando a tecnologia mais empregada na concepção de aplicações para a Internet. Esta tecnologia pode ser entendida como uma instância do modelo de computação orientada a serviços, que utiliza apenas padrões amplamente difundidos na *web* (ex. XML e HTTP) e fornece mecanismos que possibilitam a realização de invocações aos serviços (através do envio de mensagens diretamente aos serviços).

Um fator importante, relacionado com a arquitetura dos serviços *web*, é a possibilidade de compor vários serviços (onde é necessário garantir a interoperabilidade entre estes serviços) com o intuito de realizar tarefas mais complexas [65]. Deste modo, uma tarefa complexa é dividida em várias subtarefas que são executadas por vários serviços *web* independentes. Estas subtarefas devem ser executadas de forma coordenada, respeitando-se as dependências entre as mesmas.

Os modelos existentes para especificar as dependências entre os serviços *web* (como o WS-ORQUESTRATION [4] e o WS-CHOREOGRAPHY [22]) suportam a descrição do fluxo das invocações aos vários serviços cooperantes, que executam subtarefas que compõem uma tarefa mais complexa. Esta integração de serviços *web* através da especificação de um fluxo de trocas de mensagens pode ser entendida como uma coordenação orientada a controle [63].

Esta dissertação propõe uma infra-estrutura de coordenação para serviços *web* mais ampla, que coordena tais serviços *web* através de um repositório de dados compartilhado, sendo portanto uma coordenação orientada a dados [63]. Desta forma, os serviços *web* não se coordenam através de trocas de mensagens diretas entre si, mas suas interações são controladas através de um repositório de dados compartilhado que pode ser usado tanto como mediador quanto para armazenamento de dados compartilhados, oferecendo comunicação desacoplada.

Esta infra-estrutura utiliza um modelo de coordenação orientada a dados baseado em espaço de tuplas [41]. Desta forma, esta abordagem apresenta todas as vantagens relacionadas com a coordenação através de espaços de tuplas (ex. desacoplamento) e ainda possui segurança de funcionamento (tolerância a faltas e intrusões), que é uma característica fundamental quando consideramos sistemas desenvolvidos para operar na Internet.

Conforme já mencionado, esta mesma infra-estrutura pode ser usada como repositório seguro e confiável de dados, acessíveis a todos os serviços *web* cooperantes na execução de determinada tarefa. Deste modo, esta infra-estrutura pode ser usada tanto na coordenação quanto na cooperação entre serviços *web*. Alguns exemplos de aplicações que podem se beneficiar desta infra-estrutura são apresentados na seção 5.7, onde é possível perceber que esta mesma infra-estrutura é genérica suficiente para ser usada em um grande número de aplicações.

1.2 Objetivos da Dissertação

O objetivo geral desta dissertação é propor uma infra-estrutura para coordenação e/ou cooperação de serviços *web*, que garanta segurança de funcionamento mesmo que ocorram falhas bizantinas [54]

em alguns componentes do sistema, e avaliar os custos envolvidos no seu acesso e suas causas.

No que tange a definição desta infra-estrutura, o objetivo geral da dissertação é o projeto e a concretização deste suporte de coordenação e/ou cooperação para serviços *web*. Este projeto é baseado no espaço de tuplas apresentado em [12], que, dentre outros mecanismos, utiliza replicação Máquina de Estados [68] para garantir segurança de funcionamento.

Baseado neste objetivo mais geral, uma série de objetivos específicos são definidos:

1. Desenvolvimento de um espaço de tuplas com segurança de funcionamento. Este espaço de tuplas foi projetado em [12] e agrega diversos mecanismos de tolerância a faltas (ex. replicação Máquina de Estados) e segurança (ex. controle de acesso). A concretização deste espaço de tuplas é chamada de DEPSPACE.

2. Projeto e desenvolvimento de um protocolo de difusão atômica tolerante a faltas bizantinas. Este protocolo é baseado no algoritmo de consenso PAXOS BIZANTINO [26, 83] e é usado na replicação do DEPSPACE.

3. Projeto e desenvolvimento de uma infra-estrutura de coordenação e/ou cooperação de serviços *web*. Esta infra-estrutura tem como base o DEPSPACE, onde alguns mecanismos precisam ser desenvolvidos para manter as propriedades de segurança de funcionamento do sistema. Além disso, esta dissertação tem como objetivo investigar alguns cenários reais em que este tipo de infra-estrutura pode ser usada, bem como sua relação com os principais padrões para cooperação entre serviços *web*. A concretização do modelo proposto para esta infra-estrutura é chamada WS-DEPENDABLESPACE.

4. Analisar os custos envolvidos no acesso ao WS-DEPENDABLESPACE. Além de determinar as causas relacionadas com estes custos, detalhando as várias fases que compõem o acesso ao sistema.

1.3 Organização do Texto

A organização deste texto reflete as diversas etapas cumpridas para alcançar os objetivos específicos listados na seção anterior.

O capítulo 2 apresenta o modelo de sistema adotado na concepção dos sistemas desenvolvidos nesta dissertação. Este capítulo também descreve os principais conceitos básicos sobre sistemas distribuídos, que são usados no decorrer da dissertação, dando ênfase para os problemas relacionados com tolerância a faltas e intrusões. Além disso, apresenta o modelo de coordenação através de espaços de tuplas, destacando as operações básicas suportadas por este modelo.

No capítulo 3, as principais características do DEPSPACE são descritas, abordando a qualidade dos serviços oferecidos pelo mesmo. Além disso, este mesmo capítulo apresenta a arquitetura deste

espaço de tuplas e destaca as principais decisões de projeto relacionadas com a implementação deste sistema.

O capítulo 4 apresenta o protocolo de difusão atômica elaborado a partir do algoritmo de consenso PAXOS BIZANTINO. Este protocolo tolera faltas bizantinas e forma a base da replicação Máquina de Estados implementada no DEPSPACE. Este capítulo pode ser dividido em duas partes: a primeira discute o algoritmo PAXOS BIZANTINO, salientando suas características e destacando os principais pontos envolvidos na preservação das propriedades do consenso; a segunda parte aborda a transformação deste algoritmo em um protocolo de difusão atômica (também tolerante a faltas bizantinas), onde são apresentadas as alterações realizadas no algoritmo do PAXOS BIZANTINO, bem como os diversos mecanismos introduzidos na concretização deste protocolo.

O capítulo 5 começa apresentando alguns conceitos relacionados com a arquitetura orientada a serviços, dando ênfase à arquitetura dos serviços *web*. Este capítulo também discute algumas vantagens da utilização de um modelo baseado em espaços de tuplas na cooperação e/ou coordenação de serviços *web*. Após isso, apresenta o WS-DEPENDABLESPACE, uma infra-estrutura com segurança de funcionamento usada na coordenação e/ou cooperação de serviços *web*. A arquitetura deste sistema é discutida e os vários mecanismos desenvolvidos para garantir que este sistema funcione com segurança são abordados, onde alguns detalhes relevantes de implementação são destacados. Além disso, apresenta algumas aplicações que podem utilizar este modelo para a coordenação e/ou cooperação de vários serviços *web*, bem como sua relação com os principais padrões para cooperação entre serviços *web*.

Uma série de experimentos realizados com os protocolos e mecanismos que compõem os sistemas concretizados neste trabalho são discutidos no capítulo 6. Mais precisamente, a latência dos sistemas é investigada, sendo que para alguns cenários também o *throughput* é abordado. Além disso, o protocolo de difusão atômica apresentado neste trabalho, que forma a base destes sistemas, é examinado em diversos cenários, incluindo execuções com concorrência e com faltas.

Finalmente, no capítulo 7 são apresentadas as conclusões do trabalho, bem como as perspectivas futuras.

Capítulo 2

Conceitos Básicos

Este capítulo apresenta alguns conceitos básicos sobre sistemas distribuídos, que serão utilizados no decorrer deste texto. Em especial, são apresentados o conceito de espaço de tuplas e o modelo de sistema assumido nos projetos dos sistemas abordados neste texto (capítulos 3 e 5). Este capítulo também apresenta uma breve descrição de alguns problemas relacionados com tolerância a faltas e intrusões.

2.1 Modelo de Sistema

Nos projetos de sistemas, algumas premissas são assumidas em relação ao ambiente no qual estes sistemas serão executados. Estas premissas são definidas como o modelo de sistema e são fundamentais para o entendimento das características e limitações das soluções adotadas nestes projetos.

As seções seguintes abordam as premissas fundamentais assumidas no decorrer deste texto.

2.1.1 Processos

Assume-se que o sistema consiste de um conjunto (possivelmente infinito) de processos, dividido em subconjuntos: um contendo um número ilimitado de clientes $\Pi = \{c_1, c_2, ...\}$ e outro composto por n servidores $U = \{s_1, ..., s_n\}$. Além disso, no projeto do WS-DEPENDABLESPACE (capítulo 5), assume-se um subconjunto de n_g *gateways* $G = \{g_1, ..., g_{n_g}\}$.

2.1.2 Modelo de Falhas nos Processos

Todos os processos do sistema estão sujeitos a *faltas bizantinas* [54] (também conhecida como maliciosa ou arbitrária). Os processos podem ser corretos ou faltosos, sendo que os processos corretos comportam-se de acordo com sua especificação e os faltosos podem exibir um comportamento

arbitrário, i.e., podem parar, omitir o envio e/ou recebimento de mensagens, enviar mensagens inesperadas e/ou incorretas, etc.

As faltas bizantinas podem ser de natureza acidental (ex. erros de programação e/ou faltas de parada no servidor) ou maliciosa (ex. ataques bem sucedidos que levam a intrusões [76]). Durante todo o texto, o número de faltas suportadas pelo sistema (número de servidores faltosos) é representado pela letra f.

2.1.3 Modelo de Sincronismo

Vários modelos de sincronismo podem ser adotados no projeto de sistemas distribuídos, indo desde modelos onde a noção de tempo é completamente inexistente (ex. [39]) até modelos completamente síncronos, onde o sistema é completamente dependente do tempo (ex. [49]). Esta noção de tempo está relacionada tanto com o tempo necessário na comunicação entre os processos quanto ao tempo gasto em computações locais.

Neste trabalho, o modelo de sistema com *sincronia terminal* (*eventually synchronous system model*) [36] é adotado. Neste modelo, em todas as execuções do sistema, existe um limite Δ e um instante GST (*Global Stabilization Time*), tal que para toda mensagem enviada por um processo correto para outro processo correto no instante de tempo $u >$ GST é recebida antes de $u + \Delta$. É importante ressaltar que apesar do modelo estipular a existência destes limites, nenhum processo do sistema precisa conhecê-los, e tampouco serem os mesmos em diferentes execuções do sistema. Este modelo significa que o sistema pode funcionar de forma assíncrona na maior parte do tempo, mas existem períodos de estabilidade onde os atrasos nas comunicações são limitados (períodos de sincronia).

Assume-se também que todas as computações locais requerem intervalos de tempo desprezíveis. Esta premissa se fundamenta no fato de que, mesmo que o tempo requerido para algumas operações locais seja considerável (ex. operações criptográficas), tais computações estão menos sujeitas a interferências externas, e portanto seu caráter assíncrono acaba sendo pouco válido na prática.

Este modelo de sistema foi adotado devido as características do algoritmo de consenso usado na implementação do protocolo de difusão atômica apresentado no capítulo 4. Estas premissas são suficientes para garantir a terminação deste protocolo e, ao mesmo tempo, podem ser observadas mesmo em redes de larga escala como a Internet (que apesar da ausência de garantias opera a maior parte do tempo de forma estável).

2.1.4 Canais de Comunicação

Os processos do sistema comunicam-se através de *canais ponto a ponto confiáveis e autenticados*. Deste modo, todas as mensagens enviadas por um processo correto acabam por serem recebidas por todos os processos destinatários corretos. Além disso, estas mensagens não poderão sofrer alterações, sendo possível determinar a identidade do emissor das mesmas.

Estes canais são facilmente implementados através da combinação do protocolo TCP [45] e algumas funções criptográficas, como MACs (*Message Autentication Code*) [20]. Neste trabalho, utiliza-se um tipo especial de MAC, chamado HMAC (*Hashed* MAC) [75].

2.2 Tolerância a Faltas e Intrusões

A crescente dependência por serviços confiáveis que operem sem interrupções demanda sistemas com alta disponibilidade, que forneçam serviços de forma correta mesmo na presença de faltas e intrusões. Estes sistemas, capazes de permanecerem corretos mesmo em circunstâncias adversas, são ditos sistemas tolerantes a faltas e intrusões [40, 76]. Como exemplo deste tipo de sistemas pode-se citar: BFS [26], onde é apresentado um sistema de arquivos tolerante a faltas bizantinas; BASE [27], que aborda um banco de dados orientado a objetos implementado a partir de uma biblioteca para replicação tolerante a faltas bizantinas; DISTRACT [38], que apresenta um servidor *web* tolerante a intrusões; e DNSSEC [24], que discute o projeto e implementação de um serviço de nomes distribuído e seguro.

Esta seção descreve alguns problemas clássicos relacionados com tolerância a faltas e intrusões.

2.2.1 Consenso

Em um sistema distribuído, formado por vários processos independentes, o *problema do consenso* consiste em fazer com que todos os processos corretos acabem por decidir o mesmo valor, o qual deve ter sido previamente proposto por algum dos processos do sistema. Formalmente, este problema é definido em termos de duas primitivas [46]:

- *propose(G,v)*: o valor *v* é proposto ao conjunto de processos *G*.
- *decide(v)*: executado pelo protocolo de consenso para notificar ao(s) interessado(s) (geralmente alguma aplicação) que *v* é o valor decidido.

Estas primitivas devem satisfazer as seguintes propriedades de segurança (*safety*) e vivacidade (*liveness*) [7, 8]:

- **Acordo**: Se um processo correto decide *v*, então todos os processos corretos terminam por decidir *v*.
- **Validade**: Um processo correto decide *v* somente se *v* foi previamente proposto por algum processo.
- **Terminação**: Todos os processos corretos terminam por decidir.

A propriedade de acordo garante que todos os processos corretos decidem o mesmo valor. A validade relaciona o valor decidido com os valores propostos e sua alteração dá origem a outros tipos de consensos. As propriedades de acordo e validade definem os requisitos de segurança do consenso, já a propriedade de terminação define o requisito de vivacidade deste problema.

Uma variante deste problema é o *consenso uniforme* [29], onde todos os processos (sendo eles corretos ou não) precisam decidir o mesmo valor. No entanto, esta abordagem não faz muito sentido em sistemas bizantinos, visto que nenhum controle é assumido em relação ao comportamento dos processos faltosos.

Existem muitos trabalhos sobre consenso, isto se deve ao fato deste problema ser equivalente ao problema de difusão atômica (ver seção 2.2.2), que é a base para a replicação Máquina de Estados (ver seção 2.2.3), a qual é o método mais empregado na implementação de sistemas distribuídos tolerantes a faltas e intrusões. Grande parte destes trabalhos estão destinados a resolver o consenso em sistemas não bizantinos [52, 55], onde os integrantes do sistema distribuído podem falhar apenas por parada (*crash*). No entanto, neste trabalho consideramos este problema em sistemas bizantinos, onde as dificuldades e os cuidados necessários são muito maiores. Este texto apresenta um protocolo de difusão atômica baseado no algoritmo de consenso PAXOS BIZANTINO [26, 58, 83] (ver capítulo 4).

Um dos trabalhos mais interessantes envolvendo o problema do consenso é [39], onde é provado a impossibilidade de se resolver este problema em um sistema distribuído completamente assíncrono onde pelo menos um processo pode ser faltoso (qualquer tipo de falta). Em vista disso, a maioria das propostas encontradas na literatura assumem algum tipo de temporização no sistema (geralmente *timeouts*).

2.2.2 Difusão Atômica

O problema da *difusão atômica* [46], também conhecido como *difusão com ordem total*, consiste em fazer com que todos os processos corretos, membros de um grupo, entreguem todas as mensagens difundidas neste grupo na mesma ordem.

A difusão atômica é definida sobre duas primitivas básicas [46]:

- A-*multicast(G,m)* ou TO-*multicast(G,m)*: utilizada para difundir a mensagem *m* para todos os processos pertencentes ao grupo *G*.
- A-*deliver(p,m)* ou TO-*deliver(p,m)*: chamada pelo protocolo de difusão atômica para entregar à aplicação a mensagem *m*, difundida pelo processo *p*.

Formalmente, um protocolo de difusão atômica deve satisfazer as seguintes propriedades [46]:

- **Validade:** Se um processo correto difundiu *m* no grupo *G*, então algum processo correto pertencente à *G* terminará por entregar *m* ou nenhum processo pertencente à *G* está correto;

- **Acordo:** Se um processo correto pertencente a um grupo G entregar a mensagem m, então todos os processos corretos pertencentes a G acabarão por entregar m;
- **Integridade:** Para qualquer mensagem m, cada processo correto pertencente ao grupo G entrega m no máximo uma vez e somente se m foi previamente difundida em G (pelo emissor de m – *sender(m)*);
- **Ordem Total Local:** Se dois processos corretos p e q entregam as mensagens m e m' difundidas no grupo G, então p entrega m antes de m' se e somente se q entregar m antes de m'.

Quando consideramos que os processos estão sujeitos a faltas bizantinas, a propriedade de integridade precisa ser reformulada [46]:

- **Integridade:** Para qualquer mensagem m, cada processo correto pertencente ao grupo G entrega m no máximo uma vez e somente se o emissor de m (*sender(m)*) é correto, então m foi previamente difundida em G.

Com esta definição, a propriedade de integridade se refere apenas a difusão (*multicast*) e entrega (*deliver*) de mensagens por processos corretos[1]. No entanto, é difícil diferenciar um processo correto de um processo faltoso que está se comportando corretamente em relação ao protocolo. As outras propriedades não precisam sofrer alterações por estarem relacionadas apenas com processos corretos.

Um resultado teórico interessante é que a difusão atômica e o consenso são problemas equivalentes em sistemas onde os processos então sujeitos tanto a falhas de parada [28] quanto a falhas bizantinas [31].

2.2.3 Replicação Máquina de Estados

A replicação Máquina de Estados [68] é a abordagem mais abrangente e também a mais usada na implementação de sistemas tolerantes a faltas, tanto por parada [68] quanto bizantinas [26]. Esta abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados[2].

Para isso, a replicação Máquina de Estados exige que todas as réplicas, *(i.)* partindo de um mesmo estado e *(ii.)* executando o mesmo conjunto de requisições na mesma ordem, *(iii.)* cheguem ao mesmo estado final, o que define o determinismo de réplicas.

O item *(i.)* é facilmente garantido, bastando iniciar todas as réplicas com o mesmo estado (i.e, iniciar todas as variáveis que representam o estado com os mesmos valores nas diversas réplicas). Para prover o item *(ii.)*, é necessário a utilização de um protocolo de difusão atômica (ver seção

[1] O problema com a definição normal de integridade está em determinar se o processo que se diz emissor de m (*sender(m)*) realmente é este emissor.

[2] Em qualquer ponto da execução do sistema distribuído todas as réplicas devem possuir o mesmo estado.

2.2.2). Já para alcançar o item *(iii.)*, é necessário que as operações executadas pelas réplicas sejam deterministas, i.e, que a execução de uma mesma operação (com os mesmos parâmetros) resulte no mesmo resultado nas diversas réplicas. Além disso, o estado produzido (a mudança no estado) por esta operação deve ser o mesmo nas várias réplicas do sistema.

2.3 Espaço de Tuplas

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. Sendo assim, os processos de um sistema distribuído podem interagir através desta abstração de memória compartilhada.

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 (ou *template*) se algum de seus campos não tem valor definido. Diz-se que uma tupla t e um molde $\bar{t}$ combinam se e somente se eles têm o mesmo número de campos e todos os valores e tipos dos campos definidos em $\bar{t}$ são iguais aos valores e tipos dos campos correspondentes em t. Por exemplo, uma tupla $\langle \text{CLIENTE}, 1, dado \rangle$ combina com o molde $\langle \text{CLIENTE}, 1, * \rangle$ ('$*$' denota um campo sem definição do molde).

A coordenação por espaço de tuplas (ou coordenação generativa), introduzida pela linguagem de programação para sistemas paralelos LINDA [41], suporta comunicações desacopladas no espaço (os processos não precisam conhecer as localizações uns dos outros) e no tempo (os processos não precisam estar ativos ao mesmo tempo). Além disso, este modelo de coordenação fornece algum poder de sincronização entre processos.

As manipulações realizadas no espaço de tuplas consistem em invocações de três operações básicas [41]: $out(t)$ 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); $rd(\bar{t})$, usada na leitura de uma tupla que combina com 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). Note que, de acordo com as definições anteriores, o espaço de tuplas funciona como uma memória associativa: os dados são acessados a partir de seu conteúdo, e não através de seu endereço.

A figura 2.1 ilustra as três operações básicas que podem ser realizadas no espaço de tuplas[3]. A figura 2.1(a) mostra um cliente executando a operação de *out*, onde a tupla $\langle \text{CLIENTE}, 1, dado \rangle$ é inserida no espaço. Na figura 2.1(b), um cliente lê a tupla $\langle \text{CLIENTE}, 1, dado \rangle$ do espaço através da

[3] O estado do espaço de tuplas representado nesta figura reflete a execução em seqüência destas operações (*out*, *rd* e *in*).

(a) OUT. (b) RD. (c) IN.

Figura 2.1: Principais operações realizadas no espaço de tuplas.

operação *rd*, usando o molde $\langle \text{CLIENTE}, 1, * \rangle$ (recupera algum dado relacionado com o cliente 1). Finalmente, a figura 2.1(c) ilustra a operação *in*, onde de forma semelhante a operação *rd*, a tupla $\langle \text{CLIENTE}, 1, dado \rangle$ é lida e removida do espaço. A diferença entre as operações *rd* e *in* é que a operação *in* também remove a tupla do espaço.

2.4 Considerações Finais

Este capítulo apresentou o modelo de sistema considerado neste trabalho e os principais conceitos relacionados com sistemas distribuídos utilizados nos demais capítulos. Além disso, apresentou alguns conceitos relacionados com o modelo de coordenação por espaços de tuplas, destacando as principais operações que podem ser executadas neste espaço.

Capítulo 3

DEPSPACE: Um Espaço de Tuplas com Segurança de Funcionamento

Este capítulo descreve as principais características do DEPSPACE, abordando a qualidade dos serviços oferecidos pelo mesmo. Além disso, apresenta os principais aspectos relacionados com a implementação deste sistema, destacando algumas decisões de projeto.

3.1 Contexto e Motivação

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 (ou coordenação generativa) [41] destaca-se por oferecer uma comunicação onde as interações são desacopladas, sendo uma abordagem flexível e simples. Esta coordenação, realizada através de um espaço de tuplas, é desacoplada no tempo (os participantes não precisam estar ativos no mesmo instante) e no espaço (os participantes não precisam se conhecer).

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 (usando replicação) [10] quanto em mecanismos que permitam a construção de aplicações tolerantes a faltas sobre o espaço de tuplas (suportando transações) [48]. Com relação à segurança, espaços de tuplas seguros [23, 33, 77] foram propostos, os quais garantem que processos não executem operações no espaço de tuplas sem permissão, 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). No entanto, recentemente surgiu uma abordagem mais abrangente para a concepção de espaços de tuplas, que agrupa mecanismos de tolerância a faltas (como replicação) e de segurança (como criptografia) [12, 17]. Este modelo 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 [40, 76].

Este capítulo discute as principais características desta abordagem para espaços de tuplas, cuja arquitetura e mecanismos são propostos em [12], através da apresentação do DEPSPACE (*Dependable Tuple Space*) [13, 14], que é a concretização deste espaço de tuplas (realizada neste trabalho). O DEPSPACE forma a base do modelo de infra-estrutura para coordenação de serviços *web* cooperantes descrito no capítulo 5.

3.2 Características do DEPSPACE

Como já mencionado, o DEPSPACE utiliza mecanismos de tolerância a faltas e de segurança para fornecer serviços tolerantes a intrusões. Deste modo, este sistema possui segurança de funcionamento, que é uma característica fundamental dos sistemas ditos confiáveis e seguros [9].

Para um espaço de tuplas possuir segurança de funcionamento é necessário que o mesmo suporte os atributos de segurança de funcionamento [9] aplicáveis no contexto de um espaço de tuplas. Estes atributos 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;
- **Confidencialidade**: o conteúdo de campos de uma tupla não podem ser revelados a partes não autorizadas.

Uma discussão mais alongada sobre estes atributos, destacando suas interpretações no contexto de um espaço de tuplas pode ser encontrada em [12, 15].

3.3 Arquitetura do DEPSPACE

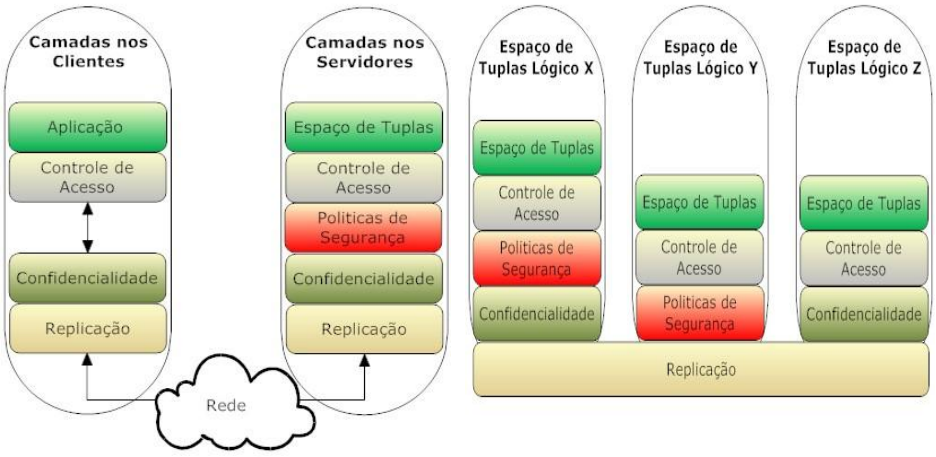
O DEPSPACE é a implementação de um espaço de tuplas com segurança de funcionamento [14], fornecendo um serviço que satisfaz todas as propriedades de interesse da segurança de funcionamento

anteriormente descritas. Para isso, o DEPSPACE foi projetado e construído em camadas, sendo que em cada camada uma funcionalidade diferente é adicionada.

A estrutura do DEPSPACE é apresentada na figura 3.1(a)[1]. Nos clientes encontram-se as camadas de replicação, confidencialidade e controle de acesso. Já os servidores possuem as camadas de replicação, confidencialidade, políticas de segurança e controle de acesso. 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 tem nenhuma relação uns com os outros.

Outro ponto interessante desta abordagem (construção em camadas) é que 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 (a camada de replicação é essencial), bem como suas configurações.

Estes aspectos podem ser observados na figura 3.1(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.



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

Figura 3.1: Arquitetura do DEPSPACE.

Os diversos mecanismos utilizados nas camadas do DEPSPACE para prover as funcionalidades necessárias para a segurança do funcionamento do espaço de tuplas são descritos a seguir.

[1] A camada "Espaço de Tuplas", presente na pilha dos servidores, representa uma implementação determinista de um espaço de tuplas local, i.e, as operações realizadas nos diversos servidores terão o mesmo resultado (ex. remoção de tuplas). A necessidade desta característica é explicada na seção 3.3.1.

3.3.1 Replicação Tolerante a Faltas Bizantinas

No DEPSPACE, o espaço de tuplas é mantido replicado em um conjunto de n servidores de tal forma que falhas (por parada ou maliciosas) em alguns deles (no máximo f, sendo $n \geq 3f + 1$) não ferem nenhum atributo de segurança de funcionamento do sistema. Este conjunto de servidores utiliza a replicação Máquina de Estados[2] (ver seção 2.2.3), uma solução clássica para implementar sistemas tanto tolerantes a faltas por parada (acidentais) [68] quanto faltas bizantinas (maliciosas) [26]. Esta abordagem garante linearização [47], que é uma propriedade de consistência na qual todos os servidores apresentam a mesma seqüência de estados.

Este mecanismo está relacionado principalmente com as propriedades de disponibilidade e confiabilidade, pois visa garantir que o espaço de tuplas continue executando as operações a ele endereçadas, comportando-se de acordo com sua especificação, mesmo que até f dos n servidores sejam faltosos (os servidores corretos mascaram o comportamento dos faltosos).

Como abordado na seção 2.2.3, este tipo de replicação requer que todas as réplicas (servidores), *(i.)* partindo de um mesmo estado e *(ii.)* executando o mesmo conjunto de requisições na mesma ordem, *(iii.)* cheguem ao mesmo estado final.

Para garantir o item *(i.)* basta iniciar todos os servidores do DEPSPACE com o mesmo estado (ex., sem nenhum espaço lógico ativo). No entanto, apesar da arquitetura do DEPSPACE ter sido proposta em [12], um protocolo (completamente funcional) para a ordenação das requisições não foi especificado. Deste modo, o item *(ii.)* só é alcançado através do uso do protocolo de difusão atômica especificado no capítulo 4, o qual garante que todos os servidores executarão o mesmo conjunto de requisições e na mesma ordem, desde que no máximo f servidores sejam faltosos. Já para garantir o item *(iii.)*, é necessário que as operações executadas tenham o mesmo resultado nos diversos servidores do DEPSPACE, desviando da idéia original para espaços de tuplas que não prevê este determinismo [41]. Além disso, todos os servidores devem executar a mesma política de segurança em cada espaço de tuplas lógico ou apresentar esta camada desativada (ver seção 3.3.2).

3.3.2 Controle de Acesso

Este mecanismo é usado para impedir que clientes não autorizados executem operações no sistema. O controle de acesso é um mecanismo fundamental para manutenção da integridade e confidencialidade das informações manipuladas pelo sistema, pois previne que clientes não autorizados obtenham acesso as tuplas, além de impedir que clientes faltosos saturem o sistema (escrevendo uma grande quantidade de tuplas nos espaços lógicos). Atualmente, o DEPSPACE implementa controle de acesso de duas formas:

- **Baseado em credenciais:** para cada tupla inserida no DEPSPACE é possível definir quais são as credenciais necessárias para acessá-la, tanto para leitura quanto para remoção (acesso em

[2] É possível implementar este tipo de replicação com apenas $n \geq 2f + 1$ servidores (réplicas), que são executados separadamente do protocolo de acordo (que exige $n \geq 3f + 1$) [82]. No entanto, no DEPSPACE isto não é considerado, pois o protocolo de acordo é executado pelos próprios servidores.

nível de tuplas). Estas credenciais são definidas pelo processo que insere a tupla. Existem dois níveis de acesso, sendo possível também definir quais são as credenciais necessárias para inserir uma tupla em cada espaço lógico (acesso em nível de espaço). Adicionalmente, este mesmo controle pode ser usado para determinar quais processos têm permissão para criar espaços de tuplas lógicos no sistema.

- **Políticas de granularidade fina:** o DEPSPACE suporta a definição de políticas de acesso de granularidade fina [16]. Estas políticas controlam o acesso a um espaço de tuplas lógico considerando três parâmetros: o identificador do cliente, a operação que será executada (juntamente com seus argumentos) e o estado deste espaço. Um exemplo de política é: *"uma operação* $out(\langle CLIENTE, id, x\rangle)$*) só pode ser executada se não houver nenhuma tupla que combina com* $\langle CLIENTE, id, *\rangle$ *no espaço"*.

As credenciais requeridas para inserção de tuplas em um espaço de tuplas lógico e sua política de segurança são sempre definidas no momento em que o espaço (lógico) é criado.

A implementação do DEPSPACE, apresentada neste texto, também pode ser chamada de *policy-enforced augmented tuple space* (PEATS), por suportar a definição de políticas de segurança de granularidade fina (*policy-enforced*) e fornecer uma operação adicional (*augmented*), chamada *conditional atomic swap* – $cas(\bar{t}, t)$ [70], que aumenta o poder de sincronização do espaço de tuplas.

A operação $cas(\bar{t}, t)$ funciona como uma execução indivisível do código: **if** $\neg rdp(\bar{t})$ **then** $out(t)$ ($\bar{t}$ é um molde e t uma entrada). Esta operação insere t no espaço somente se $rdp(\bar{t})$ não retorna 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 espaço de tuplas que a suporta seja capaz de resolver o problema do consenso [70], o qual é a base para a solução de muitos problemas de sincronização distribuída.

O DEPSPACE é a primeira implementação de um PEATS com segurança de funcionamento. Recentemente, foi provado que este espaço de tuplas (PEATS) é um objeto de memória compartilhada universal, i.e., pode ser usado para implementar qualquer objeto de memória compartilhada [16].

3.3.3 Criptografia

O DEPSPACE utiliza transformações criptográficas para garantir confiabilidade nas comunicações e confidencialidade das tuplas. O uso de criptografia nas comunicações está relacionado também com a autenticação dos canais que ligam os processos do sistema, tanto clientes quanto servidores. Esta autenticação é alcançada com a utilização de funções de resumos criptográficos na produção de códigos de autenticação (HMACs).

Garantir confidencialidade no DEPSPACE, sendo este um espaço de tuplas replicado, não é uma tarefa trivial, pois não é possível confiar nos servidores individualmente visto que até f podem ser faltosos, i.e, podem 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 scheme* – PVSS) [69]. 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 que será compartilhado entre os servidores. Um segredo pode ser remontado apenas com a combinação de $f+1$ destes *shares*, o que torna impossível que uma coalizão de servidores faltosos revele o conteúdo de uma tupla.

Como não é possível enviar diferentes versões de uma requisição para diferentes servidores (contendo apenas seu *share* de segredo compartilhado), o cliente deve cifrar cada um dos *shares* com uma chave secreta compartilhada com o servidor que vai armazenar esse *share*. Deste modo, o cliente envia a requisição com todos os *shares* e cada servidor terá acesso apenas ao *share* a ele endereçado (caso contrário, um servidor faltoso teria acesso a todos os *shares* e poderia remontar o segredo e revelar a 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 [12]: **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*.

Sendo assim, nas requisições de inserção de tuplas, o cliente envia aos servidores: a tupla cifrada, os *shares* do segredo cifrados, as provas[3] de que estes *shares* são válidos e o *fingerprint* da tupla. A figura 3.2 ilustra uma operação de inserção de tupla, onde podemos observar que o cliente envia aos servidores os vários dados relacionados com esta operação. Nesta figura, ainda é possível verificar que cada servidor obterá acesso apenas ao *share* do segredo a ele endereçado.

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[4] (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 corresponde ao *fingerprint*. Caso isso não seja verificado, o cliente elimina esta tupla do espaço (se ainda não eliminou) e re-executa a operação. A eliminação de tuplas inválidas, somente necessária nas operações de leitura, é realizada em dois passos: *(1)* o cliente envia para os servidores todas as respostas recebidas, como prova que esta tupla é inválida (para isso ser possível, os servidores devem assinar as respostas às requisições de leitura); e *(2)* os servidores verificam a autenticidade das respostas e, se a tupla realmente é inválida, removem esta tupla dos seus espaços locais. Este esquema

[3] Estas provas possibilitam que os servidores gerem outra prova, indicando que o *share* por ele armazenado é válido. Esta nova prova, que é enviada aos clientes nas operações de leitura, permite que o cliente utilize apenas *shares* válidos na recuperação do segredo, evitando que um tupla válida seja determinada como inválida (por causa da utilização de *shares* inválidos) [12].

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

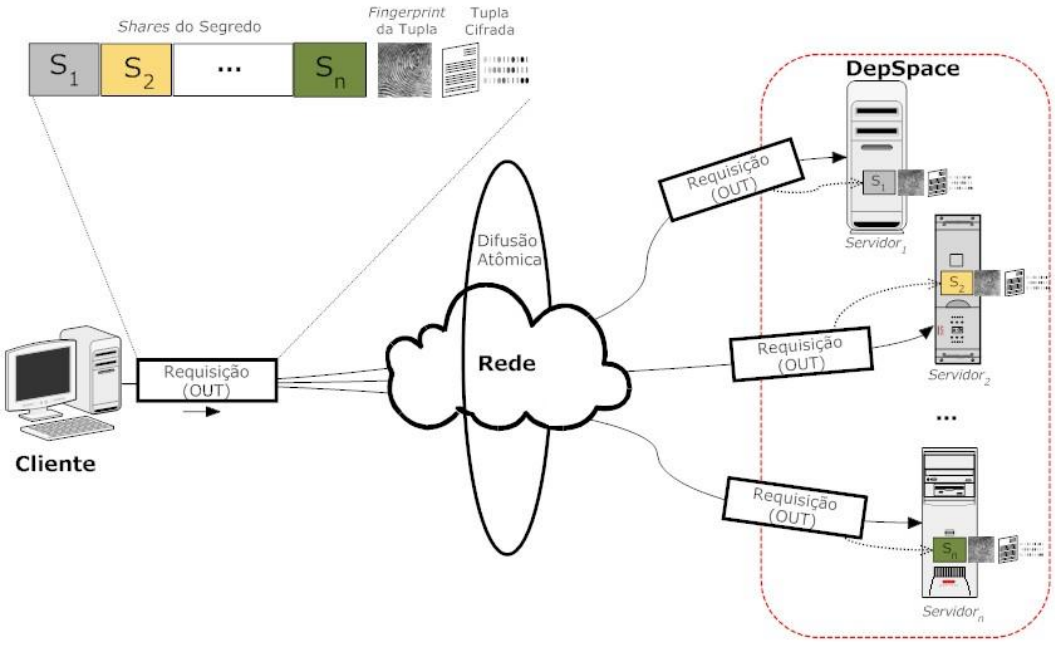


Figura 3.2: Esquema de confidencialidade.

de confidencialidade foi desenvolvido em [12, 15], onde uma discussão mais alongada sobre o PVSS pode ser encontrada.

3.4 Implementação do DEPSPACE

O DEPSPACE foi implementado na linguagem de programação Java, a qual foi escolhida principalmente pela sua característica de portabilidade. Atualmente o DEPSPACE é uma implementação completamente funcional de uma espaço de tuplas com segurança de funcionamento, embora ainda esteja em suas primeiras versões.

Os canais confiáveis e autenticados do sistema são implementados através do uso de *sockets* TCP e chaves de sessão baseadas no algoritmo *HmacSHA-1*, que também é usado na computação de *hashs* utilizados pelo mecanismo de confidencialidade (ver seção 3.3.3). Para criptografia simétrica é empregado o algoritmo *Triple DES*, enquanto que o RSA é usado em assinaturas digitais, necessárias por exemplo, na troca de líder do protocolo de difusão atômica (apresentado na seção 4.3.7). Ainda neste protocolo, a otimização referente ao uso de *hashs* para o acordo é implementada usando a função de *hash* MD5. Estas primitivas criptografias são fornecidas pelo provedor padrão da biblioteca JCE (*Java Cryptography Extensions*) – versão 1.5.

Vários pontos interessantes, relacionados com a implementação do DEPSPACE, são abordados a seguir. A forma de utilização deste sistema é discutida no apêndice A.

3.4.1 Camada de Replicação

Conforme mencionado anteriormente, a camada de replicação utiliza o protocolo de difusão atômica apresentado no capítulo 4. Deste modo, as requisições são recebidas e ordenadas por este protocolo, o qual garantirá que todos os servidores executem a mesma seqüência de requisições.

O protocolo de replicação é muito simples: o cliente (emissor) envia uma requisição (mensagem) usando o protocolo de difusão atômica e espera por $f+1$ respostas iguais de diferentes servidores. Como todos os servidores recebem o mesmo conjunto de requisições e na mesma ordem, e o espaço de tuplas é determinista, sempre terá no mínimo $2f+1$ servidores corretos que executarão a requisição e responderão com a mesma resposta.

Apesar do DEPSPACE suportar múltiplos espaços de tuplas lógicos, a camada de replicação é a mesma para todos os espaços lógicos ativos em um servidor (ver figura 3.1(b)). Desta forma, cada requisição é entregue na camada superior relacionada com o espaço de tuplas lógico ao qual tal requisição é endereçada.

Com o intuito de aumentar o desempenho do DEPSPACE, uma otimização de implementação é realizada nas operações de leitura ($rd()$/$rdp()$). Esta otimização consiste em tentar executar tais requisições sem a utilização do protocolo de difusão atômica, i.e., o cliente envia estas requisições aos servidores do DEPSPACE (que as executam sem que sejam ordenadas) e espera por $n-f$ respostas. Caso todas estas respostas venham a ser iguais, o valor nelas retornado é o resultado da operação. Caso contrário, a operação deverá ser processada da forma normal (utilizando o protocolo de difusão atômica descrito no capítulo 4). A necessidade de $n-f$ respostas iguais está relacionada com a manutenção da propriedade de linearização [47] do sistema (garantida pelo protocolo de replicação), de outro modo, apenas $f+1$ respostas iguais seria suficiente, já que é garantida a presença de uma resposta correta ente elas. Esta otimização é baseada no fato destas operações não alterarem o estado dos servidores, sendo que será mais efetiva em cenários com pouca concorrência e nenhum servidor faltoso no sistema.

3.4.2 Camada de Confidencialidade

Todas as primitivas de criptografia utilizadas pelo DEPSPACE são fornecidas pelo provedor padrão da biblioteca JCE (*Java Cryptography Extensions*). A única exceção é o mecanismo de confidencialidade, que foi implementado seguindo as especificações de [69], usando grupos algébricos de 192 bits (maior do que os 160 bits recomendados em [69]). A implementação deste mecanismo utiliza exaustivamente a classe `BigInteger`, fornecida pela API Java, que fornece vários métodos úteis na implementação de elementos criptográficos na linguagem Java.

Devido à questões de segurança, relacionadas com o mecanismo de confidencialidade [15], a implementação do DEPSPACE não prevê campos tipados para as tuplas, deste modo todos os campos são acessados por esta camada na forma de *strings*. Esta abordagem simplifica o projeto do DEPSPACE, visto que qualquer tipo pode ser facilmente convertido para *string* e vice-versa.

Conforme explicado na seção 3.3.3, as respostas das operações de leitura devem ser assinadas. Com o objetivo de aumentar o desempenho do DEPSPACE, é realizada uma otimização de implementação com relação a estas operações. Esta otimização consiste em fazer com que os clientes indiquem a necessidade de assinar as respostas. Deste modo, os clientes corretos apenas solicitarão respostas assinadas quando encontrarem alguma tupla inválida.

3.4.3 Camada de Controle de Acesso

A implementação desta camada é baseada no uso de listas de controle de acesso (ACLs – *Access Control Lists*). Desta forma, cada cliente deverá ter um identificador único, que é enviado aos servidores durante o estabelecimento do canal autenticado.

Como pode ser visto na figura 3.4, as credencias requeridas para remover ou ler uma tupla são especificadas na forma de um conjunto com os identificadores dos processos (clientes) que podem realizar estas operações. Na classe `DepTuple` estes conjuntos são representados pelos campos `c_rd` e `c_in`. De forma semelhante, também é especificado o conjunto de processos que têm permissão de inserir tuplas no espaço.

3.4.4 Camada de Políticas de Segurança

As políticas de acesso, usadas por esta camada, devem ser especificadas no momento da criação do espaço de tuplas lógico (caso deseja-se ativar esta camada). Atualmente, o sistema reconhece apenas políticas definidas na linguagem de programação GROOVY [30], que é uma linguagem de *script* suportada pela máquina virtual Java (JVM). Além desta linguagem estar em total conformidade com a linguagem Java, possui várias características que tornam a programação fácil e ágil.

A idéia é que uma política seja definida em uma classe GROOVY e enviada aos servidores na forma de uma *string*. Nos servidores, a política é transformada em *bytecode* Java e instanciada como um executor de políticas. Toda vez que uma operação é processada pelos servidores, algum método de autorização é acessado neste executor de políticas. Deste modo, a operação apenas é processada pela camada superior se o acesso for autorizado.

O *script* que define a política de acesso deve estender a classe abstrata `PolicyEnforcer`, que define os métodos de autorização `canExecuteOp`, `canExecuteRd`, `canExecuteIn` e `canExecuteCas`. A implementação padrão destes métodos sempre nega o acesso ao espaço.

Para garantir que nenhum código malicioso será executado pelo *script*[5], o carregador de classes (*class loader*) usado para instanciar o executor de políticas é protegido por um gerenciador de segurança, que apenas permite a leitura do conteúdo do espaço de tuplas. Operações de I/O e chamadas externas não poderão ser executadas pelo *script*.

Note que, após o *script* ser processado durante a criação do espaço lógico, sua execução inclui apenas chamadas a métodos normais Java, nenhuma interpretação de *script* precisa ser feita.

[5]Por exemplo, a chamada `System.exit(0)` que finalizará a réplica.

3.4.5 Interface das Camadas e Estrutura das Tuplas

Todas as camadas do DEPSPACE (ver figura 3.1(a)), tanto do lado cliente quanto do lado servidor, implementam a mesma abstração de espaço de tuplas. Esta abstração é representada pela interface `DepSpace`, que é implementada por todas as camadas do sistema. Esta interface, apresentada na figura 3.3, fornece todas as operações suportadas pelo DEPSPACE.

```
public interface DepSpace {

  void out(DepTuple tuple, Context ctx) throws DepSpaceException;

  DepTuple rd(DepTuple template, Context ctx) throws DepSpaceException;
  DepTuple in(DepTuple template, Context ctx) throws DepSpaceException;

  DepTuple rdp(DepTuple template, Context ctx) throws DepSpaceException;
  DepTuple inp(DepTuple template, Context ctx) throws DepSpaceException;

  DepTuple cas(DepTuple template, DepTuple tuple,
                                  Context ctx) throws DepSpaceException;
}
```

Figura 3.3: Interface `DepSpace`.

Outro ponto que merece destaque é que em todos os métodos (representando as operações do espaço de tuplas) têm um parâmetro extra, que representa o contexto da invocação (`Context`). Este contexto é usado pelos clientes e servidores para passar informações, relacionadas com a invocação corrente, entre as camadas. Por exemplo, o cliente utiliza o contexto para informar suas credenciais, usadas pelo mecanismo de controle de acesso. Já nos servidores, estas credenciais devem ser fornecidas à camada de controle de acesso e/ou de políticas de segurança, que determinará se o cliente tem permissão de acesso.

Como pode ser observado na figura 3.3, no DEPSPACE as tuplas e *templates* são representadas pela abstração `DepTuple`. A figura 3.4 apresenta uma visão resumida desta classe, que contem todas as informações relacionadas com uma tupla do DEPSPACE: os campos da tupla (representado por um *array* de `Object`), ACLs que controlam o acesso para leitura e remoção de tuplas (o identificador dos processos é representado por um inteiro – `int`, ver seção 3.4.3) e informações extras referentes ao esquema de confidencialidade (`PublishedShares` contém, por exemplo, as provas de validade dos *shares* juntamente com os *shares* cifrados, enquanto `Share` é o *share* decifrado que foi destinado a este servidor).

Com o objetivo de simplificar o projeto, eliminando a necessidade de gerenciar várias classes, a classe `DepTuple` representa os diferentes tipos de tuplas (com/sem confidencialidade, com/sem controle de acesso) tornando desnecessária a construção de uma hierarquia de classes.

```
public class DepTuple implements Serializable {

    // Campos da tupla
    private Object[] fields;

    // ACLs para operações de leitura e remoção
    private final int[] c_rd;
    private final int[] c_in;

    // Dados relacionados com o esquema de confidencialidade
    private PublishedShares publishedShares;
    private Share share = null;

    // Métodos para criar vários tipos de tuplas
    public static DepTuple createTuple(Object... fields) {
        return new DepTuple(fields,null,null,null);
    }

    public static DepTuple createAccessControledTuple(
            int[] c_rd, int[] c_in, Object... fields) {
        return new DepTuple(fields,c_rd,c_in,null);
    }
    ...
}
```

Figura 3.4: Classe que representa tuplas e templates.

3.4.6 Criação de Espaços de Tuplas Lógicos

Como já mencionado, o DEPSPACE suporta a execução de múltiplos espaços de tuplas lógicos. Um espaço de tuplas lógico é criado através de uma mensagem enviada aos servidores (usando o protocolo de difusão atômica). Esta mensagem especifica o nome lógico do espaço (identificador), as credenciais requeridas para inserir tuplas no espaço e a qualidade de serviço que será fornecida pelo espaço. Esta qualidade de serviço define quais são as camadas que estarão ativas no espaço. Se a camada de política de segurança for ativada, a política para o acesso ao espaço deve ser enviada junto com esta mensagem.

Um cliente cria o espaço de tuplas lógico através da classe `DepSpaceAdmin`, que pode ser acessada por todos os clientes[6]. A assinatura dos métodos implementados por esta classe é apresentada na figura 3.5, onde encontramos métodos relacionados tanto com a criação de espaços lógicos (`createSpace`) quanto com a desativação destes espaços (`deleteSpace`).

Outro ponto que merece destaque é que quando um espaço de tuplas lógico é criado, o objeto `DepSpaceAccessor` usado para acessar este espaço também é criado, sendo o valor retornado nesta

[6]Adicionalmente, mecanismos de controle de acesso podem ser usados para permitir que apenas alguns clientes criem espaços de tuplas lógicos.

operação. Além disso, a classe `DepSpaceAdmin` fornece métodos para criar (`createAccessor`) e finalizar (`finalizeAccessor`) estes objetos usados no acesso aos espaços lógicos. Note que, para fornecer os métodos de acesso aos espaços lógicos, a classe `DepSpaceAccessor` implementa a interface `DepSpace` (figura 3.3).

```
public class DepSpaceAdmin{

    // Cria um espaço de tuplas lógico e um objeto para acessar este espaço.
    public DepSpaceAccessor createSpace(Properties props)
                                            throws DepSpaceException;

    // Cria um objeto para acessar o espaço de tuplas lógico definido
    // nas propriedades "props".
    public DepSpaceAccessor createAccessor(Properties props)
                                            throws DepSpaceException;

    // Desativa o espaço de tuplas lógico identificado por "name".
    public void deleteSpace(String name) throws DepSpaceException;

    // Finaliza o objeto "accessor" e o espaço de tuplas lógico que
    // este objeto estava acessando.
    public void finalizeAccessor(DepSpaceAccessor accessor)
                                            throws DepSpaceException;

}
```

Figura 3.5: Assinatura dos métodos da classe `DepSpaceAdmin`.

3.4.7 Suporte a Interceptadores

Além das funcionalidades que já foram apresentadas, o DEPSPACE também fornece suporte ao uso de interceptadores, os quais são projetados para interceptar o fluxo normal de uma seqüência requisição/resposta.

Esta interceptação ocorre em pontos pré-determinados, sendo que ao todo são quatro pontos de interceptação: no cliente – antes da requisição ser enviada e da resposta ser recebida (processada); nos servidores[7] – antes da requisição ser processada e da resposta ser enviada ao cliente (imediatamente acima da camada de replicação). Nestes pontos, é possível obter informações referentes a mensagem (requisição ou resposta) e manipular o fluxo normal da seqüência requisição/resposta (através de exceções). Além disso, é possível manipular informações relacionadas com o contexto da mensagem. A implantação deste suporte foi baseada nos interceptadores portáveis existente na arquitetura CORBA [61].

[7]De forma semelhante ao mecanismo de políticas de segurança (ver seção 3.4.4), é necessário garantir que nenhum código malicioso seja executado por estes interceptadores.

A interface destes interceptadores, chamada `DepSpaceInterceptor`, é apresentada na figura 3.6. Todas as classes que desejarem interceptar mensagens precisam implementar esta interface. As mensagens recebidas (requisições ou respostas) são interceptadas pelo método `interceptReceive`, onde todas as informações relacionadas com esta mensagem são encontradas na estrutura `TSMessage`. Já as mensagens enviadas são interceptadas pelo método `interceptSend`, onde os dados relacionados com a resposta e com a requisição que originou esta resposta (apenas nos servidores) podem ser obtidos. Note que, nestes dois métodos poderá ocorrer a exceção `InterceptionException`, através da qual o fluxo normal da seqüência requisição/resposta pode ser desviado. Por exemplo, quando uma requisição é interceptada em um servidor e esta exceção é lançada (ocorrer a exceção), tal requisição não é processada e o resultado enviado ao cliente é um valor relacionado com a exceção. Este valor, que é determinado pelo interceptador, pode ser qualquer objeto.

```
public interface DepSpaceInterceptor{

    // Intercepta o recebimento de uma mensagens (requisição ou resposta)
    public void interceptReceive(TSMessage message)
                                                throws InterceptionException;

    // Intercepta o envio da resposta "reply"
    // relacionada com a requisição "request"
    public void interceptSend(TSMessage reply, TSMessage request)
                                                throws InterceptionException;
}
```

Figura 3.6: Interface `DepSpaceInterceptor`.

3.4.8 Características Avançadas Java

Com o objetivo de maximizar o desempenho do sistema, tanto em relação a latência das operações quanto a escalabilidade do sistema, algumas características novas oferecidas pela plataforma Java são utilizadas: a API NIO (pacote *javax.nio*) e várias classes para controle de concorrência (pacotes *java.util.concurrent* e *java.util.concurrent.locks*).

A API NIO contém uma série de classes para o gerenciamento de *buffers* e conexões, como a classe `Selector` que fornece a mesma funcionalidade de seleção de canais da chamada de sistema UNIX `select`. Esta API apresenta um desempenho muito superior em relação a forma normal de se implementar sobre *sockets* na linguagem Java, principalmente em relação a escalabilidade.

Os pacotes relacionados com o controle de concorrência fornecem mecanismos, como a classe `BlockingQueue` e vários tipos de *locks*, que permitem uma sincronização entre *threads* mais eficiente e escalável do que a forma usual de realizar esta sincronização (baseada em monitores e na uso da palavra reservada *synchronized*).

3.5 Trabalhos Relacionados

Dentre os vários sistemas desenvolvidos com o intuito de introduzir segurança e/ou tolerância a faltas em espaços de tuplas, a implementação apresentada neste capítulo é a mais completa, i.e., é a única que considera tanto mecanismos para tolerância a faltas (como replicação) quanto para segurança (como criptografia e ACLs). Sendo assim, o DEPSPACE tolera o tipo mais abrangente de faltas, chamado de faltas bizantinas [54].

O sistema FT-LINDA [10] agrega suporte a tolerância a faltas através do uso de replicação Máquina de Estados [68], onde o protocolo de difusão atômica é baseado em um mecanismo de comunicação de grupo. Outra abordagem para tolerância a faltas em espaço de tuplas é apresentada em [80], a qual utiliza replicação através de sistemas de quóruns [42]. O fator negativo destas soluções é que ambas consideram apenas faltas por parada, não suportando intrusões.

Com relação ao controle de acesso, o sistema SECSPACES [23] suporta partições lógicas no espaço, o que possibilita o controle de acesso em nível de espaço. Além disso, este sistema também suporta controle de acesso em nível de tuplas. Existem outros sistemas que apresentam este controle, através da especificação de quais processos podem tanto executar operações no espaço de tuplas [33] (acesso ao espaço) quanto ler e/ou remover tuplas [77] (acesso às tuplas). No entanto, o uso de políticas de segurança prevista no DEPSPACE não é abordado por nenhum destes sistemas.

Recentemente, o sistema GIGASPACES [43, 71] também abordou o suporte à tolerância a faltas por parada, através dos conceitos de transações e de *clusters*, i.e., o espaço de tuplas é replicado (replicação primário – *backup*) em vários espaços de tuplas físicos. Este esquema utiliza o padrão *master/worker* para o balanceamento de carga. Além disso, este sistema também fornece mecanismos de controle de acesso, onde é possível especificar quais clientes podem executar operações de escrita, de leitura ou administrativas.

Outras implementações de espaços de tuplas que merecem destaque, são os sistemas TSPACES [74] e JAVASPACES [73] (também implementados na linguagem de programação Java), que fornecem algum suporte de tolerância a faltas para as aplicações, através de funcionalidades relacionadas com transações. O sistema TSPACES também apresenta um esquema de controle de acesso (baseado em ACLs), onde é organizado uma hierarquia de acesso. Neste sistema, os espaços de tuplas são divididos em domínios, onde um conjunto de clientes tem acesso permitido.

3.6 Considerações Finais

O desenvolvimento do sistema DEPSPACE apresentou vários pontos interessantes, principalmente no que diz respeito aos mecanismos relacionados com as camadas de confidencialidade e replicação. Várias decisões de projeto auxiliaram na concretização deste sistema, sendo que a mais importante delas foi a definição de uma interface padrão para as camadas, o que possibilitou que os espaços de tuplas lógicos sejam facilmente configurados de várias formas.

Além disso, as classes desenvolvidas para a administração e para o acesso aos espaços de tuplas lógicos facilitam consideravelmente as interações com os servidores do sistema, o que simplifica o uso deste sistema por aplicações.

Este capítulo apresentou alguns aspectos relacionados com a implementação e as características do DEPSPACE, o qual é um espaço de tuplas com segurança de funcionamento, implementando tanto mecanismos para tolerância a faltas quanto para segurança (tolerante a intrusões). Este espaço de tuplas será a base para a definição da infra-estrutura de coordenação de serviços *web* cooperantes, discutida no capítulo 5.

Capítulo 4

Difusão Atômica Baseada no Algoritmo Paxos Bizantino

Este capítulo apresenta o algoritmo PAXOS BIZANTINO, que é capaz de resolver o problema do consenso (ver seção 2.2.1) em sistemas sujeitos a faltas bizantinas [54]. Além disso, mostra como este algoritmo pode ser utilizado no desenvolvimento de um protocolo de difusão atômica que também tolera esta classe de faltas.

4.1 Contexto e Motivação

Muitos trabalhos sobre consenso são encontrados na literatura, isto se deve principalmente ao fato deste problema ser a base para a replicação Máquina de Estados [51, 68], a qual é um dos métodos mais empregados na implementação de sistemas distribuídos tolerantes a faltas. Neste tipo de replicação, é necessário o determinismo de réplicas, ou seja, a partir de um mesmo estado inicial, os estados de todas as réplicas devem ter a mesma evolução. Dentre as propriedades necessárias para o determinismo de réplicas, o consenso está relacionado com a difusão atômica, mais precisamente com a ordenação das mensagens, isto é, todos os processos servidores da replicação precisam entrar em consenso sobre a ordem de entrega (execução) das mensagens.

O protocolo de consenso a ser explorado neste texto é o PAXOS, o qual foi originalmente proposto para a resolução do problema do consenso em sistemas sujeitos a faltas de parada [52, 53], sendo estendido para ambientes sujeitos a faltas bizantinas [26, 35, 58, 83]. Na maioria destes trabalhos, os algoritmos de consenso apresentados são transformados em protocolos de difusão atômica.

Este capítulo apresenta um protocolo de difusão atômica tolerante a faltas bizantinas, desenvolvido a partir das variações do algoritmo PAXOS BIZANTINO apresentadas em [26, 83], onde algumas alterações e novos mecanismos foram introduzidos com o intuito de garantir as propriedades da difusão atômica (ver seção 2.2.2). Este protocolo de difusão atômica, que é usado pela camada de replicação do sistema DEPSPACE (apresentando no capítulo 3), assume canais confiáveis e, ao

contrário de [26], não requer o uso de *checkpoints* e nem tampouco tem complexidade de mensagens cúbica (como [83]). Sendo portanto mais simples e eficiente que estas propostas prévias para ambientes em que a premissa dos canais confiáveis se justifica.

4.2 PAXOS BIZANTINO

Esta seção estuda o protocolo de consenso PAXOS BIZANTINO com modificações para terminação rápida (em dois passos de comunicação) proposto em [83], o qual também é conhecido como PAXOS AT WAR (PAW). Este protocolo foi escolhido, para ser usado nas implementações, devido ao seu bom desempenho em casos livres de falhas e à sua resistência ótima [26] (o número de servidores n deve ser pelo menos $3f+1$ para que o sistema suporte até f faltas bizantinas). Além disso, em *rounds* favoráveis (ver adiante), não faz uso de criptografia.

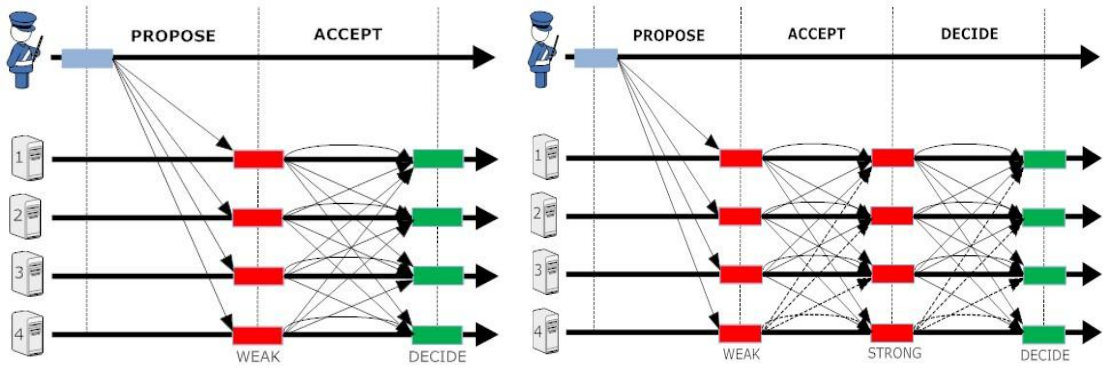
O algoritmo original do PAXOS considera três classes de agentes [52, 53]: proponentes (*proposers*), aceitantes (*acceptors*) e aprendizes (*learners*). Os proponentes propõem os valores, os aceitantes são responsáveis pela escolha de um único valor entre os valores propostos e os aprendizes são os agentes que precisam aprender o valor decidido (escolhido). Em nossa implementação, todos os servidores do sistema desempenham estes três papéis ao mesmo tempo; no entanto, distinções serão feitas para facilitar o entendimento do protocolo.

Este algoritmo é executado em *rounds*, sendo que, em cada *round* r, um proponente p_r (não necessariamente distinto) é escolhido líder. Este líder tem a responsabilidade de escolher e enviar uma proposta aos aceitantes, os quais tentarão fazer deste valor a decisão do consenso através de uma ou mais fases de trocas de mensagens visando garantir a propriedade de acordo (ver seção 2.2.1). Por fim, quando estabelecida, a decisão de consenso é enviada aos aprendizes.

As propriedades de segurança (ver seção 2.2.1) sempre são mantidas pelo protocolo, mesmo na presença de processos faltosos (desde que o limite f seja respeitado) e de assincronismos nas computações e comunicações. No entanto, o consenso somente terá progresso em *rounds* favoráveis, onde são necessários três passos de comunicação para decidir. Um *round* é considerado *favorável* quando seu líder é correto e o sistema está num período de sincronia: o limite Δ (apresentado na seção 2.1) é respeitado, i.e., as comunicações e computações ocorrem dentro de um período de tempo limitado. Nesta situação, um valor proposto é escolhido e aprendido dentro do período de um *round*. Adicionalmente, um *round* é dito *muito favorável* se for favorável e não existem falhas nos aceitantes, nestes casos o consenso é decidido em apenas dois passos de comunicação. Caso um *round* r não seja favorável, um novo *round* é iniciado com um novo líder e assim sucessivamente até que um valor seja aprendido.

As figuras 4.1(a) e 4.1(b) ilustram alguns cenários de execução do PAW. A figura 4.1(a) mostra uma execução, em que o protocolo executa um *round* muito favorável e consegue terminar em apenas dois passos[1]. Este padrão segue as otimizações definidas em [58, 83]. O caso normal de operação

[1]Note que, como os próprios aceitantes são os aprendizes, não é necessário difundir o valor decidido para estes últimos.



(a) Muito Favorável. (b) Favorável com a_4 faltoso.

Figura 4.1: Execuções do PAXOS.

do PAW, onde o primeiro *round* é favorável, é apresentado na figura 4.1(b): um *round* do algoritmo consolida uma decisão em três passos de comunicação [26] (ver seção 4.2.2). Caso uma decisão não possa ser determinada em um *round* (dentro de um determinado período de tempo), um novo *round* é iniciado através de um protocolo de transição, que requer dois passos de comunicação (ver seção 4.2.3). Este processamento de troca de *rounds* é a única fase do protocolo PAW onde criptografia de chave pública é necessária. Sendo assim, o protocolo não requer este mecanismo em execuções favoráveis.

4.2.1 Varáveis e Conjuntos

Para facilitar o entendimento deste texto, algumas considerações sobre os conjuntos e as variáveis, bem como a notação utilizada nos algoritmos abordados nesta seção são necessárias. O símbolo i, que aparece associado a conjuntos e variáveis, é o identificador do consenso ao qual estes dados pertencem, deste modo podemos suportar execuções simultâneas de várias instâncias do PAW. Cada processo (servidor), envolvido na execução do consenso, possui as seguintes variáveis:

- A: Representa o conjunto dos processos aceitantes do sistema.
- A': Representa o conjunto dos processos aceitantes do sistema menos o próprio processo, i.e., $A \setminus \{myself\}$.
- $input_i$: Cada processo tem um conjunto de variáveis $input_i$, onde cada variável contém o valor inicialmente proposto v, para o consenso identificado por i.
- $W_i^r[a]$: Vetor onde são armazenados os valores fracamente aceitos (*weak*) como valor de decisão para o *round* r do consenso identificado por i, a é o identificador do processo que aceitou fracamente o valor armazenado nesta posição do vetor.
- $S_i^r[a]$: Vetor onde são armazenados os valores fortemente aceitos (*strong*) como valor de decisão para o *round* r do consenso identificado por i, a é o identificador do processo que aceitou fortemente o valor armazenado nesta posição no vetor.

- $D_i^r[a]$: Vetor onde são armazenados os valores decididos no *round r* do consenso identificado por *i*, *a* é o identificador do processo que decidiu o valor armazenado nesta posição do vetor.

- F_i^r: Representa um conjunto que contém os identificadores dos processos que localmente congelaram (*freeze*) o *round r* do consenso identificado por *i*.

Os conjuntos apresentados a seguir são utilizados apenas no processo de troca de líder, apresentado na seção 4.2.3.

- $\overline{W}_i^R$: Neste conjunto são armazenados todos os valores fracamente aceitos como valor de decisão em cada *round* (*R* representa todos os *rounds*) do consenso identificado por *i*.

- $\overline{S}_i^R$: Este conjunto possui a mesma semântica do anterior. No entanto, neste conjunto são armazenados os valores fortemente aceitos como valor de decisão.

- $CW_i[a]$: Neste vetor são armazenados todos os valores fracamente aceitos como valor de decisão em cada *round* ($\overline{W}_i^R$) de todos (ou de uma parte deles) os processos participantes do consenso identificado por *i*, *a* é o identificador do processo que notificou este aceite dos valores armazenados nesta posição do vetor. Isto permitirá que o novo líder escolha o valor a ser proposto no próximo *round*.

- $CS_i[a]$: Este vetor possui a mesma semântica do anterior. No entanto, neste conjunto são armazenados os valores fortemente aceitos como valor de decisão.

4.2.2 Algoritmo PAW

As trocas de mensagens observadas na figura 4.1 refletem as computações e comunicações apresentadas pelo algoritmo 1, o qual não contempla a troca de *rounds* que será discutida na seção 4.2.3. Este algoritmo representa uma releitura do protocolo original do PAW [83].

O ponto de entrada para a execução do PAW é o procedimento *consensus* (linhas 1 - 4 do algoritmo 1). A idéia é que os participantes do sistema distribuído (possivelmente a partir de uma camada superior) executem este procedimento, o qual tem como parâmetros o identificador *i* do consenso (instância do PAW) e o valor *v* a ser proposto. Note que, este processamento está relacionado com a primitiva *propose* do problema do consenso (ver seção 2.2.1).

Na execução deste procedimento, o processo define sua variável $input_i$ com o valor *v*, e caso tal processo seja o líder[2] do *round 0* do consenso *i*, uma proposta é enviada a todos os processos pertencentes ao conjunto *A* (os processos que estão participando do consenso) e está dado início ao consenso.

[2] *leader(i,r)* é uma função deterministica que retorna o líder do *round r* do consenso *i*. Para o primeiro *round* ($r = 0$), o resultado desta função será o líder que decidiu o consenso $i-1$. Para *rounds* seguintes ($r > 0$), o retorno desta função será *((líder de* $r-1$*) + 1) mod n*.

Deste modo, os processos receberão uma proposta contendo os identificadores do consenso, do *round* e do proponente (i, r e p_r, respectivamente), além do valor proposto v e das provas que este valor é bom. Quando esta mensagem é recebida por algum processo (linhas 5 - 8 do algoritmo 1), primeiramente é determinado se o proponente (que enviou esta proposta) é o líder do *round* em questão e se o valor proposto é um "valor bom"[3]. Caso isto seja verificado, o valor v é fracamente aceito por este processo, o qual armazena v no vetor W e envia uma mensagem (WEAK) aos outros processos notificando que aceitou fracamente v no *round* r do consenso identificado por i.

Algoritmo 1 Algoritmo PAW executado pelo processo *"myself"*

procedure $consensus(i,v)$
1: $input_i \leftarrow v$
2: **if** $leader(i,0) = myself$ **then**
3: $send(\text{PROPOSE}, i, 0, myself, input_i, \perp, \perp)$ *to* A
4: **end if**
Require: $receive(\text{PROPOSE}, i, r, p_r, v, CW_i, CS_i)$
5: **if** $(leader(i,r) = p_r) \wedge good(v, r, input_i, CW_i, CS_i)$ **then**
6: $W_i^r[myself] \leftarrow v$
7: $send(\text{WEAK}, i, r, myself, v)$ *to* A'
8: **end if**
Require: $receive(\text{WEAK}, i, r, a, v)$
9: $W_i^r[a] \leftarrow v$
10: **if** $(\#_v W_i^r > \frac{n+f}{2})$ **then**
11: **if** $S_i^r[myself] = \perp$ **then**
12: $S_i^r[myself] \leftarrow v$
13: $send(\text{STRONG}, i, r, myself, v)$ *to* A'
14: **end if**
15: **if** $(\#_v W_i^r > \frac{n+3f}{2})$ **then**
16: $D_i^r[myself] \leftarrow v$
17: $send(\text{DECIDE}, i, r, myself, v)$ *to* A'
18: $decide(i,v)$
19: **end if**
20: **end if**
Require: $receive(\text{STRONG}, i, r, a, v)$
21: $S_i^r[a] \leftarrow v$
22: **if** $(\#_v S_i^r > 2f) \wedge (D_i^r[myself] = \perp)$ **then**
23: $D_i^r[myself] \leftarrow v$
24: $send(\text{DECIDE}, i, r, myself, v)$ *to* A'
25: $decide(i,v)$
26: **end if**
Require: $receive(\text{DECIDE}, i, r, a, v)$
27: $D_i^r[a] \leftarrow v$
28: **if** $(\#_v D_i^r > f) \wedge D_i^r[myself] = \perp$ **then**
29: $D_i^r[myself] \leftarrow v$
30: $send(\text{DECIDE}, i, r, myself, v)$ *to* A'
31: $decide(i,v)$
32: **end if**

Quando um mensagem WEAK é recebida por algum processo (linhas 9 - 20 do algoritmo 1), o valor v associado a esta mensagem é armazenado no vetor W (representando que v foi fracamente aceito por a). Após isso, duas ações (não excludentes) podem ser tomadas, dependendo do número

[3] A forma de calcular um "valor bom" (função *good*) será explicada na seção 4.2.4. No primeiro *round* de cada instância do PAW , qualquer valor é considerado bom.

de processos que aceitaram fracamente v ($\#_v$) como valor de decisão para este consenso no *round r*:

- $\#_v > \frac{n+f}{2}$: Nesta situação, o valor v é fortemente aceito por este processo. Sendo assim, um teste é realizado para determinar se já existe um valor (que obrigatoriamente será v) fortemente aceito para este consenso e *round*. Caso este valor ainda não exista, o valor v é armazenado no vetor S e uma mensagem (STRONG) notificando que v foi fortemente aceito é enviada aos outros processos (A'). Note que, para um valor ser fortemente aceito, primeiramente é necessário ser fracamente aceito por mais de $\frac{n+f}{2}$ processos, isto garante as propriedades de segurança, mesmo que o líder seja faltoso e proponha diferentes valores para os processos, pois assegura que este valor foi fracamente aceito pela maioria dos processos aceitantes corretos.

  Ilustração: Considere $n = 3f + 1$, então: $\frac{n+f}{2} = 2f + \frac{1}{2}$. Sabemos que é necessário $\#_v > \frac{n+f}{2}$, então: $\#_v > 2f + \frac{1}{2}$. Assim, será necessário que no mínimo $2f + 1$ processos aceitem fracamente v. Então, no pior caso, teremos f processos faltosos e $f + 1$ corretos, i.e., a maioria dos processos aceitantes corretos.

- $\#_v > \frac{n+3f}{2}$: Nesta situação, este processo já pode decidir o consenso com o valor v, pois mesmo que alguns processos do sistema não consigam decidir nesta etapa (neste passo ou até mesmo *round*), o único valor que eles poderão decidir em etapas futuras (passos ou *rounds*) será v. Sendo assim, v é armazenado no vetor D e uma mensagem (DECIDE) é difundida para A' notificando esta decisão. Por fim, a função *decide* é executada para notificar à aplicação que o consenso com identificador i foi resolvido com valor da decisão v (este processamento está relacionado com a primitiva *decide* do problema do consenso – seção 2.2.1). Esta é a alteração para terminação rápida apresentada em [83], a qual garante uma melhor performance em execuções muito favoráveis, pois é assegurado que a decisão é alcançada em apenas dois passos de comunicação.

  Ilustração: Considere $n = 3f + 1$, então: $\frac{n+3f}{2} = 3f + \frac{1}{2}$. Sabemos que é necessário $\#_v > \frac{n+3f}{2}$, então: $\#_v > 3f + \frac{1}{2}$. Assim, será necessário que no mínimo $3f + 1$ processos aceitem fracamente v, ou seja, todos os processos. Por isso, o consenso somente será decidido em dois passos de comunicação se não houverem faltas no sistema. No entanto, se aumentarmos o número de processos para $n = 5f + 1$, teremos: $\frac{n+3f}{2} = 4f + \frac{1}{2}$, então: $\#_v > 4f + \frac{1}{2}$. Assim, será necessário que no mínimo $4f + 1$ processos aceitem fracamente v, ou seja, f processos podem falhar e mesmo assim o consenso será decidido em apenas dois passos de comunicação.

Quando uma mensagem STRONG é recebida por algum processo (linhas 21 - 26 do algoritmo 1), seu valor associado v é armazenado no vetor S (representando que o valor v foi fortemente aceito por a). Depois disso, caso o número de processos que já aceitaram fortemente v (como valor de decisão para o consenso no *round r*) for maior do que $2f$, este consenso é decidido com o valor v. Assim, caso ainda não tenha decidido, v é armazenado no vetor D, uma mensagem é enviada aos outros processos notificando esta decisão, e a função *decide* é executada. É necessário esperar no mínimo $2f + 1$ ($> 2f$) notificações, pois neste caso é garantido que no mínimo $f + 1$ processos corretos aceitaram fortemente v. Deste modo, é garantido que nenhum valor diferente de v poderá ser escolhido como "valor bom" em possíveis *rounds* futuros [83] (ver seção 4.2.4).

Quando um processo recebe uma mensagem DECIDE (linhas 27 - 32 do algoritmo 1), seu valor associado *v* é armazenado no vetor *D* como o valor que foi decidido pelo processo *a*. Depois disso, caso o número de processos que decidiram o consenso no *round r* com valor *v* seja maior do que *f*, este consenso é decidido com o valor *v*. Assim, caso ainda não tenha decidido, *v* é armazenado no vetor *D*, uma mensagem é enviada aos outros processos notificando esta decisão, e a função *decide* é executada. É necessário esperar no mínimo $f + 1$ ($> f$)notificações, pois neste caso é garantido que no mínimo um processo correto decidiu o consenso no *round r* com valor *v*.

4.2.3 Congelamento e Troca de *Rounds*

Não é garantido que um *round* tenha progresso em execuções não favoráveis. Isto pode acontecer, por exemplo, quando um líder faltoso enviar diferentes propostas aos processos aceitantes ou não enviar proposta para alguns aceitantes. Sendo assim, um processo pode, quando suspeitar que não terá progresso no *round*, congelar localmente tal *round* e tentar dar início a um novo *round*. Um *round* é considerado congelado quando é localmente congelado por todos os processos corretos. Outro ponto a destacar, é que os processos corretos não alteram suas variáveis e conjuntos relacionados com *rounds* localmente congelados.

Quando ocorrer o *timeout* para um *round r* (linhas 1 - 6 do algoritmo 2), *r* é localmente congelado. Deste modo, um teste é realizado para verificar se o consenso ainda não foi decidido e *r* ainda não foi localmente congelado pelo processo em questão. Caso obtenha sucesso neste teste, *r* é localmente congelado e uma mensagem é enviada aos outros processos notificando este congelamento. Por fim, o conjunto *F* é atualizado.

Note que esta abordagem (uso de *timeouts*) não diferencia um líder faltoso de uma rede lenta, isto faz com que alguns processos corretos decidam em um *round r* e outros em *rounds* posteriores. No entanto, todos os processos corretos decidirão o mesmo valor. Isto se deve ao fato de que, após um valor *v* ser decidido em *r* por algum processo correto, *v* será o único "valor bom" que poderá ser proposto em *rounds* futuros (ver seção 4.2.4).

Quando um processo recebe uma mensagem FREEZE para um *round r* (linhas 7 - 20 do algoritmo 2), primeiramente o conjunto *F* é atualizado. Após isso, duas ações podem ser tomadas:

- Caso o número de processos que localmente congelaram *r* seja maior do que *f* e o processo em questão ainda não congelou *r*, então *r* é localmente congelado conforme descrito anteriormente (ocorrência de *timeouts*).

  Esta necessidade surge da possibilidade de em algumas situações um processo, mesmo que já tenha decidido o consenso, precisar congelar localmente um *round*[4]. Então, quando mais de *f* processos reportarem o congelamento local de um *round*, o mesmo é congelado localmente

[4]Isto se deve ao fato de não ser possível, para o líder, escolher uma "proposta boa" para um *round r'* ($r' > 0$), sem que o *round* anterior seja congelado (seção 4.2.4) e não ser garantido que todos os processos irão decidir no mesmo *round*.

também por este processo, pois é garantida a presença de no mínimo um processo correto entre os que localmente congelaram tal *round*.

Note que, se mais de f processos corretos localmente congelarem um *round*, então todos os processos corretos também acabarão por congelá-lo. No entanto, se no máximo f processos corretos localmente congelar determinado *round*, então mais de f (n - f - $f > f$) processos corretos decidirão neste *round*, e através das mensagens DECIDE (linhas 27 - 32 do algoritmo 1) todos os processos corretos acabarão decidindo no mesmo *round*.

Algoritmo 2 Protocolo de congelamento e troca de *rounds* executado pelo processo *"myself"*

procedure $timerExpited(i,r)$
1: **if** $(D_i^r[myself] = \perp) \wedge (myself \notin F_i^r)$ **then**
2: $S_i^r.freeze$
3: $W_i^r.freeze$
4: $send(\text{FREEZE}, i, r, myself)$ *to* A'
5: $F_i^r \leftarrow F_i^r \cup \{myself\}$
6: **end if**
Require: $receive(\text{FREEZE}, i, r, a)$
7: $F_i^r \leftarrow F_i^r \cup \{a\}$
8: **if** $|F_i^r| > f \wedge myself \notin F_i^r$ **then**
9: $S_i^r.freeze$
10: $W_i^r.freeze$
11: $send(\text{FREEZE}, i, r, myself)$ *to* A'
12: $F_i^r \leftarrow F_i^r \cup \{myself\}$
13: **end if**
14: **if** $\forall s \leq r : |F_i^s| > 2f$ **then**
15: $startTimer(i, r+1)$
16: $p_{r+1} \leftarrow leader(i, r+1)$
17: $\overline{W}_i^R \leftarrow W_i^s, \forall s <= r$ {todos os valores fracamente aceitos}
18: $\overline{S}_i^R \leftarrow S_i^s, \forall s <= r$ {todos os valores fortemente aceitos}
19: $send(\text{COLLECT}, i, r, myself, \overline{W}_i^R, \overline{S}_i^R)_{\sigma\_myself}$ *to* p_{r+1}
20: **end if**
Require: $receive(\text{COLLECT}, i, r, a, \overline{W}_i^R, \overline{S}_i^R)_{\sigma\_a}$
21: **if** $leader(i, r+1) = myself$ **then**
22: $CW_i[a] \leftarrow \overline{W}_i^R$
23: $CS_i[a] \leftarrow \overline{S}_i^R$
24: **if** $\exists v : good(v, r, input_i, CW_i, CS_i)$ **then**
25: $send(\text{PROPOSE}, i, r+1, myself, v, CW_i, CS_i)$ *to* A
26: **end if**
27: **end if**

- Caso todos os *rounds* até *r* tenham sido localmente congelados por mais de *2f* processos, então é iniciado o *timer* do próximo *round* (*r + 1*), pois neste caso mais de f processos corretos localmente congelaram todos os *rounds* até *r*, o que garante que todos os processos corretos também acabarão por localmente congelá-los. Isto têm duas conseqüências: *(1)* O líder do próximo *round* acabará por conseguir escolher um "valor bom" como proposta para o *round r + 1* (ver seção 4.2.4); *(2)* Todos os processos corretos acabarão por executar a função *startTimer*[5] para o *round r + 1*, o que garante sincronização entre estas ações nos diferentes processos.

[5]Esta ação associará um *timeout* ao próximo *round*, evitando que o consenso não tenha progresso devido a escolha de um líder faltoso (por exemplo, que não envia a proposta para o novo *round*).

Por fim, uma mensagem COLLECT é enviada ao novo líder, a fim de que este possa escolher um "valor bom" para propor no próximo *round*. Nesta mensagem, que deve ser assinada[6], são enviados os conjuntos de valores que foram fracamente e fortemente aceitos em todos os *rounds* até r ($\overline{W}$ e $\overline{S}$, respectivamente).

Para que um novo *round* seja iniciado, após o *round* anterior ser congelado, um valor a ser proposto deve ser determinado. Isto é possível graças ao recebimento (pelo novo líder) de mensagens COLLECT (linhas 21 - 27 do algoritmo 2). Sendo assim, quando um processo recebe esta mensagem, primeiramente verifica se realmente é o líder do próximo *round*. Se sim, armazena os valores recebidos ($\overline{W}$ e $\overline{S}$) e tenta obter o valor a ser proposto no novo *round*, a partir dos valores que já foram recebidos e da variável $input_i$ (ver seção 4.2.4). Se este valor for obtido, a nova proposta é enviada aos processos, juntamente com as provas de que o valor desta proposta é bom.

4.2.4 Computando um Valor Bom

Nesta seção, será discutido como é encontrado um "valor bom", o qual será proposto pelo líder. A mesma lógica é usada pelos aceitantes para verificar se um valor proposto é bom.

Para um valor v ser considerado bom para o *round* r, ele precisa atender a duas propriedades:

- *Ser admissível*: nenhuma decisão a não ser v deve ter sido feita por processos corretos em qualquer *round* anterior a r.
- *Ser válido*: v foi proposto em algum *round* $s : s \leq r$.

Supondo-se que R seja o conjunto de todos os *rounds* até r (1,2,...,r-1). O predicado *good* é computado usando os vetores CW e CS, sendo que:

- $CW[a] = \overline{W}^R = \{W^1, W^2, ..., W^{r-1}\}$, ou seja, todos os valores que foram fracamente aceitos em cada *round* na visão do processo a.
- $CS[a] = \overline{S}^R = \{S^1, S^2, ..., S^{r-1}\}$, ou seja, todos os valores que foram fortemente aceitos em cada *round* na visão do processo a.

Para auxiliar no desenvolvimento desta função, dois conjuntos são construídos para cada *round* $r' \in R$. Considerando: $|CW[A]^{r'}(\overline{v})|$ = número de processos de A em que $W^{r'} \subseteq \{v\}$, $|CS[A]^{r'}(\overline{v})|$ = número de processos de A em que $S^{r'} \subseteq \{v\}$ e $|CW[A]^{r'}(v)|$ = número de processos de A em que $W^{r'} = \{v\}$. Temos:

- $Poss^{r'}$: conjunto que contém todas as decisões que podem ser feitas no *round* r'. $Poss^{r'}$ = $\{v: |CS[A]^{r'}(\overline{v})| > f \vee |CW[A]^{r'}(\overline{v})| > \frac{n+f}{2}\}$.

[6] As mensagens COLLECT devem ser assinadas, pois são usadas como provas pelo novo líder.

- $Acc^{r'}$: conjunto que contém somente os valores que foram fracamente aceitos no *round* r'. $Acc^{r'} = \{v: |CW[A]^{r'}(v)| > f\}$.

Sendo assim, um valor v será considerado bom para o *round* r se ao menos uma das seguintes condições forem verificadas:

- $v \in input$ e $Poss^{r'} = \emptyset$ para todo *round* $r' < r$. Esta condição implica que o valor $v \in input$ e nenhuma decisão pôde ser tomada em *rounds* anteriores, o que significa que v é bom no *round* r.
- Existe um *round* $r' < r$ tal que $v \in Acc^{r'}$ e $Poss^t \subseteq \{v\}$ para todo t: $r' \leq t < r$. Esta condição implica que v foi fracamente aceito no *round* r', então v é bom (i.e., valido e admissível) neste *round*. Validade no *round* r' implica em também ser válido no *round* r. Admissível no *round* r', junto com o fato que nenhuma decisão diferente de v pôde ser feita nos *rounds* t: $r' \leq t < r$, implica em v ser admissível no *round* r.

Note que, o conteúdo dos vetores *CW* e *CS*, envolvidos na computação deste valor, é enviado ao novo líder assinado (mensagens COLLECT). Isto faz com que o líder seja capaz de provar que o valor escolhido é bom, além de impedi-lo de escolher qualquer valor, pois estes vetores são enviados junto com a nova proposta.

Este processamento de troca de líder pode ser implementado sem o uso de criptografia de chave pública [83] ao custo de um passo a mais de troca de mensagens (todos enviam mensagens a todos). No entanto, como somente é utilizado em *rounds* não favoráveis, o seu custo não causa um impacto significativo no desempenho do sistema como um todo.

4.3 Protocolo de Difusão Atômica

Um protocolo de difusão atômica deve garantir que todos os processos corretos, membros de um grupo, entreguem todas as mensagens difundidas neste grupo na mesma ordem (ver seção 2.2.2). Este problema é equivalente ao problema do consenso (ver seção 2.2.1), i.e., um protocolo de consenso pode ser empregado na resolução do problema da difusão atômica e vice-versa. Considerando o uso de um protocolo de consenso na elaboração de um protocolo de difusão atômica, esta equivalência ocorre na medida em que o consenso é necessário para que os processos, membros de um grupo, possam chegar a um acordo acerca da ordem de entrega das mensagens (e das próprias mensagens) difundidas neste grupo.

A implementação da difusão atômica usando o PAW se baseia na execução de uma instância deste algoritmo para cada mensagem a ser ordenada. Desta forma, uma mensagem m é a i-ésima mensagem a ser entregue para a aplicação se e somente se for o resultado da execução i do PAW. A figura 4.2 ilustra a difusão atômica de uma mensagem usando o PAW em uma execução muito favorável.

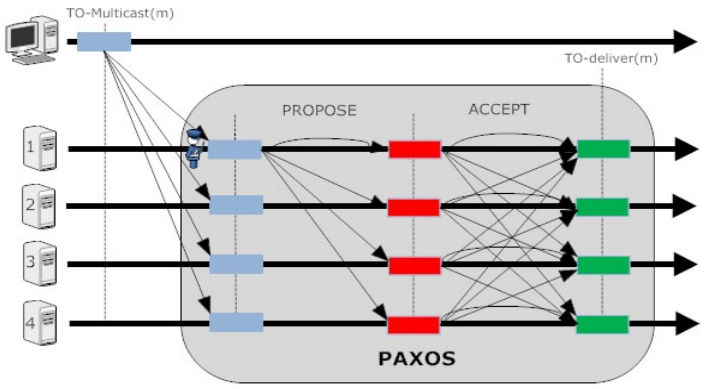


Figura 4.2: Difusão atômica baseada no PAW.

Em sistemas sujeitos a faltas bizantinas, alguns cuidados são necessários no desenvolvimento deste protocolo, que funciona da seguinte forma: um emissor, que deseja difundir uma mensagem m no grupo G com ordem total, deve enviar m a todos os servidores[7] pertencentes à G. O líder desses servidores, ao receber m, inicia a execução de número i do PAW propondo m como o valor de acordo. Ao final da execução do algoritmo de acordo, todos os processos saberão que m é a i-ésima mensagem a ser entregue. Sendo assim, um servidor entrega a mensagem m quando (1) o valor da instância i do PAW é m e (2) todas as mensagens decididas por instâncias $j < i$ do algoritmo já foram entregues.

Além disso, um valor proposto m apenas é aceito por um servidor se, além das condições normais de aceite do PAW, o número de instância i do algoritmo está dentro de um intervalo de valores de seqüência previamente definido e a autenticidade de m for comprovada, o que pode ser feito de duas maneiras: caso o servidor recebeu a mensagem m do emissor; ou caso $f+1$ servidores tenham aceito este valor, garantindo que no mínimo um servidor correto autenticou m.

Estas condições extras objetivam impedir que um servidor líder malicioso invente mensagens ou defina números de seqüência muito elevados para mensagens recebidas. Além disso, aceitar apenas valores dentro de um intervalo impede que um servidor malicioso sature o *buffer* dos outros servidores com mensagens inválidas [26]. Outra forma de impedir que um servidor malicioso defina números fora da seqüência para mensagens é fazer com que os servidores executem apenas uma instância do algoritmo por vez, não iniciando a instância de número $i+1$ antes de terminar a de número i. Desta forma, propostas para números de seqüência fora da ordem não seriam executadas e o consenso não teria progresso (nem seria iniciado pelos servidores corretos). Esta é a abordagem utilizada neste trabalho, a qual é discutida com maiores detalhes nas próximas seções.

Note que, o emissor de uma mensagem não precisa pertencer ao grupo de receptores (servidores), o que caracteriza que este protocolo suporta grupos abertos.

[7] Note que, os servidores são os processos que estão envolvidos no consenso e devem entregar as mensagens na difusão com ordem total, i.e., são os receptores da difusão com ordem total.

4.3.1 Considerações Gerais Sobre o Protocolo Desenvolvido

A seguir, algumas características relevantes do protocolo de difusão atômica desenvolvido neste trabalho são apresentadas:

- Um servidor correto executa apenas uma instância do PAW por vez, i.e., o consenso de número $i+1$ somente tem início após a decisão do consenso i. Assim, sempre teremos $2f+1$ servidores (i.e., no mínimo $f+1$ corretos) executando ou o consenso i ou o $i+1$.

- Para um *round* ser congelado é necessário que no mínimo $2f+1$ servidores congelem localmente tal *round*. Assim, sempre teremos no mínimo $f+1$ servidores corretos processando estas mensagens relacionadas com o mesmo *round* de um consenso.

- O modelo de sistema assume canais ponto a ponto confiáveis (ver seção 2.1), i.e., não ocorrem perdas de mensagens nas comunicações entre processos corretos. Deste modo, as informações referentes a execução de um consenso já decidido por um servidor são descartadas assim que o mesmo for informado que no mínimo $f+1$ servidores corretos decidiram tal consenso (enviaram a mensagem DECIDE), pois é garantido que todos os servidores corretos também acabarão por decidir este consenso, sem a necessidade de congelar *rounds*[8]. Sendo assim, um consenso i será considerado *estável* (já foi decidido por no mínimo $f+1$ servidores corretos) por um servidor, quando o mesmo receber no mínimo: $2f+1$ mensagens de WEAK para algum *round* do consenso $i+1$; ou $f+1$ mensagens de STRONG para algum *round* do consenso $i+1$; ou ainda $2f+1$ mensagens de DECIDE para algum *round* do consenso i.

- Mensagens recebidas (relacionadas com o algoritmo PAW) apenas são consideradas (executadas ou armazenadas) se o consenso a que se referem estiver dentro de uma faixa de valores. O limite inferior é determinado pelo último consenso a tornar-se estável ($lastStable$) e o superior ($highMarc$) determinará quanto um servidor poderá se atrasar em relação aos outros sem que ocorram descartes. No entanto, se um servidor se atrasar muito em relação aos outros, mas receber as mensagens em uma certa ordem (ex., não receber todas as mensagens de um servidor e depois as de outro) evoluirá naturalmente, como se não estivesse atrasado e não ocorrerão descartes. Sendo assim, quando uma mensagem para o consenso i for recebida, é verificado se: $(i > lastStable) \wedge (i < lastExecuted + highMarc)$. Caso uma mensagem recebida esteja dentro dos limites, mas refere-se a um consenso que ainda não teve início, a mesma é armazenada para ser processada assim que o determinado consenso seja iniciado (ex., linha 2 do algoritmo 5).

- Na computação de um valor bom, além das condições normais exigidas pelo PAW (ver seção 4.2.4), a mensagem que está sendo ordenada deve ser autêntica, i.e., o recebimento da mesma deve ser reportado por no mínimo $f+1$ servidores (ver seção 4.3.4) ou for determinado que $f+1$ servidores aceitaram fracamente esta proposta (garantido que pelo menos um servidor correto a autenticou). Esta segunda condição é suficiente, porém caso não seja verificada, o novo líder deve determinar a autenticidade da mensagem através da primeira condição.

[8] As informações referentes a execução de um consenso i devem ser armazenadas, mesmo após i ser decidido, para responder a possíveis pedidos de congelamento de *rounds* de i.

- A qualidade de ordem FIFO (*first input first output*) pode ser facilmente incorporada no protocolo apresentado nesta seção. Para isso, basta que os servidores aceitem (WEAK) uma mensagem para ser ordenada apenas quando a mensagem anterior (relacionada com o emissor) já tenha sido aceita. Desta forma, as mensagem são ordenadas e entregues com ordem FIFO em relação a cada emissor.

As seções seguintes apresentam o processamento relacionado com a difusão (envio) de mensagens por emissores, juntamente com o recebimento, ordenação e entrega das mesmas por um grupo de servidores (receptores). Este processamento pode ser entendido com um cliente (emissor) difundindo uma requisição (mensagem) para ser ordenada e executada (entregue) por este grupo de servidores (receptores).

4.3.2 Processamento nos Emissores

O processamento nos emissores, representado pelo algoritmo 3, é bastante simples. A idéia é que uma camada superior (possivelmente a aplicação) execute este procedimento, passando como parâmetros a mensagem e o grupo G no qual esta mensagem será difundida[9]. Sendo assim, a mensagem é enviada a todos os membros do grupo G com um identificador único, formado pelo identificador do emissor e por um contador (linha 2).

Algoritmo 3 Processamento no emissor "myself"

Inicialização
$count \leftarrow 0$
procedure $TO-multicast(G, message)$
1: $count \leftarrow count + 1$
2: $send(\text{TO-MESSAGE}, \langle message, myself, count \rangle)$ *to* G

Note que, um emissor pode falhar (falha de parada) no meio do processo de difusão de uma mensagem, enviando-a para apenas alguns servidores. Além disso, um emissor faltoso (falta maliciosa) também pode enviar uma mensagem para apenas alguns servidores. Este assunto é tratado nas seções 4.3.5 e 4.3.6, onde é especificado que para garantir que uma mensagem seja entregue a mesma deve ser enviada para no mínimo $2f+1$ servidores corretos. Como consideramos que o sistema possui $3f+1$ servidores (i.e., no mínimo $2f+1$ servidores corretos) e as comunicações ocorrem através de canais ponto a ponto confiáveis, um emissor correto sempre conseguirá se comunicar com pelo menos os $2f+1$ servidores corretos. Isto é necessário para provar a autenticidade da mensagem.

4.3.3 Processamento nos Servidores (Receptores)

O processamento nos servidores é um pouco mais complicado, pois é aqui onde as propriedades da difusão atômica (ver seção 2.2.2) devem ser garantidas. Sendo assim, através de algumas alterações

[9]Este parâmetro é utilizado apenas para seguir as primitivas de difusão atômica apresentadas na seção 2.2.2, pois as mensagens são difundidas no grupo de servidores que estão executando o PAW. Deste modo, o parâmetro G sempre será o grupo A.

no algoritmo PAW (ver seção 4.2) e do desenvolvimento de alguns mecanismos, definiu-se um protocolo onde as mensagens recebidas pelos servidores são entregues obedecendo as propriedades da difusão atômica.

Os servidores recebem as mensagens através do algoritmo 4. Quando isto acontece, a mensagem recebida é armazenada, juntamente com seu identificador, em um conjunto que contém as mensagens que precisam ser ordenadas (*unordered*). O identificador de uma mensagem é composto pela identidade do emissor e por um contador relacionado com suas mensagens (ver seção 4.3.2).

Além disso, a função *startTimer* (linha 2) é executada para associar um *timeout* com a mensagem recebida. Isto é necessário em situações onde o líder é faltoso e não propõe a ordem para esta mensagem ou o emissor é faltoso e não envia esta mensagem para o líder. O procedimento executado quando esgotar este tempo sem que a mensagem seja ordenada tentará trocar de líder (ver seção 4.3.5).

Algoritmo 4 Recebimento de mensagens (enviadas pelos emissores) no processo "myself"

Require: $receive(\text{TO-MESSAGE}, \langle message, sender, id \rangle)$
1: $unordered \leftarrow unordered \cup \{\langle message, sender, id \rangle\}$
2: $startTimer(\langle message, sender, id \rangle)$
3: **if** $inExecution \neq -1$ **then**
4: $consenso \leftarrow getConsenso(inExecution)$
5: **if** $consenso.w[0][myself] = \perp \wedge consenso.v[0] \in unordered$ **then**
6: $consenso.w[0][myself] \leftarrow consenso.v[0]$
7: $send(\text{WEAK}, inExecution, 0, myself, consenso.v[0])$ *to* A'
8: **end if**
9: **else if** $leader(lastExecuted + 1, 0) = myself$ **then**
10: $inExecution \leftarrow lastExecuted + 1$ {O identificador da primeira instância do PAW é 0.}
11: $send(\text{PROPOSE}, inExecution, 0, myself, unordered.getOlder(), \perp)$ *to* A
12: **end if**

Por fim, é verificado se existe uma execução pendente (linhas 3 - 8), i.e., se uma instância do PAW está sendo executada para ordenar a mensagem recebida e o servidor em questão estava aguardando a confirmação da autenticidade de tal mensagem para aceitar fracamente a proposta (como valor de decisão). Este processamento está relacionado apenas com o primeiro *round* (0), pois para *rounds* subseqüentes a autenticidade da proposta é determinada pelas provas, como veremos adiante.

No entanto, se não existe uma instância do PAW em execução e o servidor em questão for o líder da próxima instância, uma ordem é proposta para a execução da mensagem mais antiga recebida ($unordered.getOlder()$), que é a mensagem que foi recebida neste momento (linhas 9 - 12). Caso existisse uma mensagem mais antiga do que a recebida neste instante, uma instância do consenso estaria sendo executada para ordenar tal mensagem. Note que, este processamento reflete a função *consensus* do algoritmo 1.

O algoritmo 5 é acionado quando uma proposta é recebida, a qual é processada caso esteja relacionada com: o próximo consenso (se não existe um em execução); o consenso em execução; ou com consensos anteriores que foram congelados, mesmo estando decididos neste servidor. A proposta não é processada, sendo apenas armazenada em $pendentProp$ (para processamento posterior), caso seja referente à consensos futuros .

Algoritmo 5 Recebimento de mensagens PROPOSE no aceitante *"myself"*

Require: $receive(\text{PROPOSE}, i, r, p_r, v, proofs)$
1: **if** $i > lastExecuted + 1$ **then**
2: $\quad pendentProp \leftarrow pendentProp \cup \{\langle i, r, p_r, v, proofs\rangle\}$
3: **else**
4: $\quad executePropose(i, r, p_r, v, proofs)$
5: **end if**

Uma proposta apenas é considerada caso a identidade do líder seja verificada (linha 1 do algoritmo 6). Neste processamento, primeiramente é determinada a autenticidade da mensagem que está sendo ordenada ($checkMessageAuthentication$ – linha 3) e verificada a possibilidade de descartar dados relacionados a instâncias de consensos ($checkConsensusDiscards$ – linha 4). Estas funções, que apenas são executadas em *rounds* onde provas são necessárias (todos com exceção do primeiro), são explicadas na seção 4.3.4.

Algoritmo 6 Processamento de uma proposta pelo aceitante "myself"

procedure $executePropose(i, r, p_r, v, proofs)$
1: **if** $leader(i, r) = p_r$ **then**
2: $\quad$ **if** $proofs \neq \perp$ **then**
3: $\quad\quad checkMessageAuthentication(v, proofs)$
4: $\quad\quad checkConsensusDiscards(proofs)$
5: $\quad$ **end if**
6: $\quad$ **if** $good(v, r, proofs)$ **then**
7: $\quad\quad consenso \leftarrow getConsenso(i)$
8: $\quad\quad$ **if** $consenso = \perp$ **then**
9: $\quad\quad\quad consenso \leftarrow createConsenso(i)$
10: $\quad\quad\quad checkOutOfContext(i)$
11: $\quad\quad$ **end if**
12: $\quad\quad consenso.v[r] \leftarrow v$
13: $\quad\quad$ **if** $(v \in unordered) \vee (proofs \neq \perp)$ **then**
14: $\quad\quad\quad consenso.w[r][myself] \leftarrow v$
15: $\quad\quad\quad send(\text{WEAK}, i, r, myself, v)\ to\ A'$
16: $\quad\quad$ **end if**
17: $\quad\quad$ **if** $i = lastExecution + 1$ **then**
18: $\quad\quad\quad inExecution \leftarrow i$
19: $\quad\quad$ **end if**
20: $\quad$ **end if**
21: **end if**

Após estas verificações iniciais, caso a validade da proposta seja confirmada (linha 6), é dado início a um *round* para este consenso, que é determinado pelo processamento das linhas 7 - 12 deste algoritmo. Caso o servidor recebeu a mensagem que está sendo ordenada, a proposta é aceita fracamente (linhas 13 - 16)[10]. Note que, para *rounds* subseqüentes ao primeiro ($proof \neq \perp$) este valor é aceito mesmo que a mensagem ainda não foi recebida pelo servidor. Pois neste caso, a autenticidade da mensagem é determinada pelas provas; além disso, esta instância do PAW pode já ter sido decidida neste servidor (a mensagem foi entregue e eliminada).

[10] O *timeout* relacionado com o primeiro *round* é iniciado quando um valor é aceito fracamente como decisão do consenso, i.e, quando a proposta é aceita ou quando $f + 1$ mensagens de WEAK para o mesmo valor forem recebidas (linha 14 do algoritmo 7). Já, os *timeouts* relacionados com *rounds* subseqüentes ao primeiro são iniciados quando a mensagem COLLECT é enviada ao líder do novo *round* (linha 15 do algoritmo 2).

Algoritmo 7 Recebimento de mensagens WEAK no aceitante *"myself"*

Require: $receive(\text{WEAK}, i, r, a, v)$
1: **if** $i > lastExecuted + 1$ **then**
2: $outOfContext \leftarrow outOfContext \cup \{\langle \text{WEAK}, i, r, a, v\rangle\}$
3: **else**
4: $consenso \leftarrow getConsenso(i)$
5: **if** $consenso = \perp$ **then**
6: $consenso \leftarrow createConsenso(i)$
7: $checkOutOfContext(i)$
8: **end if**
9: $consenso.w[r][a] \leftarrow v$
10: **if** $consenso.w[r][myself] = \perp \wedge \#_v\ consenso.w[r] \geq f + 1$ **then**
11: **if** $i = lastExecution + 1$ **then**
12: $inExecution \leftarrow i$
13: **end if**
14: $consenso.w[r][myself] \leftarrow v$
15: $send(\text{WEAK}, i, r, myself, v)\ to\ A'$
16: **end if**
17: **if** $\#consenso.w[r] \geq 2f + 1$ **then**
18: $lastStable \leftarrow i - 1$
19: **end if**
20: {Execução normal do WEAK do PAW (ver algoritmo 1).}
21: **end if**

Quando uma mensagem de WEAK (algoritmo 7) é recebida por algum servidor, primeiramente é verificado se a mesma está relacionada com consensos futuros que não podem ser iniciados. Caso afirmativo, a mesma é armazenada em $outOfContext$ para processamento posterior (quando o consenso for inicializado - linha 7), no contrário determina-se o consenso em questão (linhas 4 - 8) e caso ainda não tenha aceito fracamente este valor mas algum servidor correto aceitou (recebeu mais de f mensagens deste tipo), o servidor em questão aceita fracamente tal valor, que é autêntico. Por fim, é realizada a coleta de lixo através da definição dos consensos estáveis, conforme discutido anteriormente, e segue-se para a execução normal do PAW.

As mensagens de STRONG e DECIDE são processadas na forma normal do PAW (ver algoritmo 1), apenas é necessário verificar se o identificador do consenso está dentro dos limites, se o consenso está em execução (caso contrário armazena-se em $outOfContext$) e a existência de consensos estáveis.

Algoritmo 8 Notificação da decisão de um consenso no processo "myself"

procedure $decide(i, v)$
1: $unordered \leftarrow unordered \setminus \{v\}$
2: $TO - deliver(v)$ {Entrega para camada superior}
3: $lastExecuted \leftarrow i$ {i = lastExecuted+1}
4: $inExecution \leftarrow -1$
5: **if** $\exists \langle i+1, r, p_r, v', proofs\rangle \in pendentProp$ **then**
6: $pendentProp \leftarrow pendentProp \setminus \{\langle i+1, r, p_r, v', proofs\rangle\}$
7: $executePropose(i+1, r, pr, v', proofs)$
8: **end if**
9: $checkOutOfContext(i+1)$
10: **if** $(inExecution = -1) \wedge (leader(i+1, 0) = myself) \wedge (unordered \neq \emptyset)$ **then**
11: $inExecution \leftarrow lastExecuted + 1$
12: $send(\text{PROPOSE}, inExecution, 0, myself, unordered.getOlder(), \perp)\ to\ A$
13: **end if**

As notificações sobre consensos decididos são recebidas através do algoritmo 8, que tem como parâmetro o identificador do consenso decidido e o valor da decisão. Note que esta função é acionada pelo algoritmo PAW (linhas 18, 25 e 31 do algoritmo 1). Assim, a mensagem é entregue, retirada de *unordered* e algumas variáveis são atualizadas (linhas 1- 4). Além disso, é verificado se existe alguma proposta pendente e/ou mensagens recebidas fora do contexto, relacionadas com a próxima instância do PAW, que agora possam ser processadas (linhas 5 - 9). Por fim, caso (*i.*) a próxima instância do PAW não teve início (i.e., nenhuma instância está em execução), (*ii.*) o servidor em questão é o líder da próxima instância do PAW e (*iii.*) existe alguma mensagem para ser ordenada, a próxima instância do PAW é iniciada para ordenar a mensagem mais antiga recebida (linhas 10 - 13). O item (*i.*) garante que instâncias do PAW não sejam executadas concorrentemente, o (*ii.*) é derivado do algoritmo geral do PAW (ver seção 4.2) e o (*iii.*) é necessário por motivos óbvios.

4.3.4 Congelamento e Troca de *Rounds*

As execuções de instâncias do PAW estão relacionadas entre si, pois existe uma relação de ordem entre as mesmas. Este fato faz com que algum processamento adicional seja necessário durante a operação de congelamento e troca de *rounds*. A principal alteração diz respeito ao conteúdo das mensagens COLLECT, que além dos dados normais do PAW (valores que foram fracamente e fortemente aceitos), também contém uma variável booleana indicando se o servidor recebeu do emissor a mensagem que está sendo ordenada ($consenso.value \in unordered$). Além disso, é possível que outros consensos e *rounds* também sejam congelados (localmente), como segue, para o congelamento do round r do consenso i:

1. *Caso i ainda não foi iniciado*: Então as mensagens relacionadas com este pedido de congelamento (FREEZE) apenas serão processadas quando este servidor atingir i (são armazenadas em $outOfContext$).

2. *Caso i está em execução*: Este caso reflete a forma normal de operação do PAW. Diferentes ações poderão ser tomadas, dependendo do *round* que está sendo executado[11]. Supondo que este *round* seja r', temos:

   (a) Caso $r' < r$: As mensagens relacionadas com este pedido de congelamento (FREEZE) são processadas. Caso r seja congelado, envia-se os dados referentes a r (caso existam) e aos *rounds* anteriores a r' (incluindo r') na mensagem COLLECT. É difícil de ocorrer esta situação, pois é grande a probabilidade deste servidor receber algumas mensagens dos processos corretos que estão congelando r, evoluindo assim para r.

   (b) Caso $r' = r$: As mensagens relacionadas com este pedido de congelamento (FREEZE) são processadas. Caso r seja congelado, os dados referentes aos *rounds* anteriores a r (incluindo r) são enviados na mensagem COLLECT.

[11] Um servidor está executando um *round* r caso já tenha recebido a proposta para r ou $f+1$ mensagens WEAK (já aceitou fracamente o valor proposto em r).

(c) Caso $r' > r$: Ignora-se as mensagens relacionadas com o pedido de congelamento de r (FREEZE). Como um *round* não pode começar sem que seu anterior seja congelado, r já foi congelado. Isto pode acontecer, por exemplo, caso um servidor correto esteja atrasado e solicitar o congelamento de r juntamente com f servidores faltosos. O fato de um *round* avançado (em relação a r) estar em execução, indica que não foi possível decidir i em r e todos os servidores corretos acabarão por receber a proposta para r' (ou para *rounds* subseqüentes à r').

3. *Caso i já foi decidido*: Diferente ações poderão ser tomadas, dependendo do *round* em que esta decisão foi estabelecida. Supondo que este *round* seja r', temos:

   (a) Caso $r' < r$: As mensagens relacionadas com este pedido de congelamento (FREEZE) são processadas e procede-se como descrito no item *2a*.

   (b) Caso $r' = r$: As mensagens relacionadas com este pedido de congelamento (FREEZE) são processadas e procede-se como descrito no item *2b*.

   (c) Caso $r' > r$: Ignora-se as mensagens relacionadas com o pedido de congelamento de r (FREEZE), pois r já foi congelado (semelhante ao caso *2c*.).

   Além disso, para os casos *3a* e *3b*, se no momento em que a mensagem COLLECT é enviada o servidor está processando um consenso j, sendo $j > i$ (a única possibilidade é $j = i + 1$ – senão i seria estável), congela-se localmente o *round* do consenso j em execução (adicionando as informações referentes a j na mensagem).

4. *Caso i é estável*: Ignora-se as mensagens relacionadas com o pedido de congelamento de r (FREEZE), pois neste caso todos os servidores corretos acabarão por decidir este consenso (ver seção 4.3.1). Estas mensagens nem são processadas, pois estão fora dos limites.

O novo líder[12] espera por $n - f$ mensagens COLLECT relacionadas com o *round* que está sendo congelado para então encontrar um valor bom e iniciar o próximo *round* propondo este valor (envia o PROPOSE). Como visto anteriormente, as mensagens COLLECT podem conter informações sobre *rounds* da instância do consenso seguinte à congelada. Sendo assim, após decidir o consenso relacionado com o *round* que foi congelado, caso uma valor bom para o consenso seguinte possa ser obtido (das mensagens COLLECT recebidas), tal consenso é executado. Caso contrário, as informações referentes a este consenso são eliminadas, este descarte é determinado através das provas enviadas junto com a proposta (função *checkConsensusDiscards* do algoritmo 6). Note que pode não existir um valor bom até mesmo para o consenso relacionado com o *round* inicialmente congelado, para isso basta que a mensagem que esta instância do consenso estava ordenando não seja autêntica.

O valor de uma proposta será considerado bom se, além das condições normais do PAW, a autenticidade da mensagem que está sendo ordenada for comprovada. Então, primeiramente é verificado se tal valor foi aceito (fracamente) por $f + 1$ processos (condição suficiente mas não necessária).

[12] A definição do novo líder é baseada na identidade do líder do *round* que foi congelado, como sendo: $(lider\_round\_congelado + 1)\ mod\ n$.

Caso a condição anterior não seja verificada, a autenticidade da mensagem é determinada através dos dados reportados pelos servidores nas mensagens COLLECT (uma variável indicando se a mensagem que está sendo ordenada foi recebida). Para isso, é necessário que no mínimo $f+1$ servidores reportem o recebimento de tal mensagem. Este é o processamento relacionado com a função *checkMessageAuthentication* do algoritmo 6.

4.3.5 Esgotamento do Tempo para uma Mensagem ser Ordenada

Conforme já discutido, cada servidor associa um *timeout* com cada mensagem recebida para ser ordenada. Esta seção aborda os procedimentos que devem ser realizados quando este tempo se esgotar sem que a mensagem seja ordenada (entregue), o que pode acontecer por dois motivos: *(1)* o servidor líder é faltoso e não propõe uma ordem para esta mensagem; ou *(2)* o emissor é faltoso e não envia tal mensagem para o servidor líder.

Nestas situações, o algoritmo 9 é acionado, onde é determinado se a mensagem ainda não foi ordenada (linha 1). Caso isto seja verificado, é definido se esta mensagem é oriunda de uma difusão incompleta e pode ser descartada (linhas 2 - 3, ver seção 4.3.6). No entanto, se a mensagem não for descartada e o servidor em questão ainda não executou este *timeout*, uma mensagem (TO-FREEZE) é enviada aos outros servidores informando a ocorrência deste *timeout*, além disso um novo *timeout* é associado a esta mensagem que está esperando para ser ordenada, através da função *startTimer* (linhas 5 - 9).

Algoritmo 9 *Timeout* para uma mensagem ser ordenada

procedure $timerExpited(\langle message, sender, id\rangle)$

1: **if** $\langle message, sender, id\rangle \in unordered$ **then**
2: **if** $countTimeouts(\langle message, sender, id\rangle) \geq f+2$ **then**
3: $unordered \leftarrow unordered \setminus \{\langle message, sender, id\rangle\}$
4: **else if** $timeout^{id}_{sender}[myself] = \perp$ **then**
5: $startTimer(\langle message, sender, id\rangle)$
6: $timeout^{id}_{sender}[myself] \leftarrow \langle message, sender, id\rangle$
7: $send(\text{TO-FREEZE}, myself, \langle message, sender, id\rangle)$ *to* A'
8: **end if**
9: **end if**

Todas as informações de *timeouts*, relacionadas com uma mensagem, devem ser armazenadas separadas para cada *timeout* (primeiro, segundo,...) e todas as contagens envolvendo o número de servidores que reportaram a ocorrência dos mesmos devem considerar este fator. Sendo assim, cada *timeout* é independente dos outros, a não ser pelo fato do segundo *timeout* ser maior do que o primeiro, o terceiro maior do que o segundo e assim por diante[13]. Esta abordagem visa diminuir os efeitos do assincronismo do sistema sobre este mecanismo, o qual não fere nenhuma propriedade do sistema (segurança ou vivacidade) mesmo que, por exemplo, um líder não faltoso seja trocado. Todas estas informações, relacionadas com os *timeouts*, são eliminadas quando a mensagem for ou ordenada (entregue) ou descartada.

[13] Para simplificar a apresentação deste protocolo, o armazenamento separado dos dados referentes a *timeouts*, bem como a relação entre os mesmos (o fato do *timeout* sucessor ser maior do que o anterior), não são representados nos algoritmos.

As notificações de *timeouts* são processadas através do algoritmo 10. Neste processamento, caso o *timeout* tenha ocorrido em no mínimo $f+1$ servidores, o mesmo também é executado no servidor em questão (caso ainda não o tenha sido, linhas 2 - 6), pois é garantida a presença de no mínimo um servidor correto nas notificações. Isto impede que servidores maliciosos forcem a troca de líder informando *timeouts* que não ocorreram.

Algoritmo 10 Notificações de *timeouts* para uma mensagem ser ordenada

Require: $receive(\text{TO-FREEZE}, a, \langle message, sender, id\rangle)$
1: $timeout^{id}_{sender}[a] \leftarrow \langle message, sender, id\rangle$
2: **if** $\#timeout^{id}_{sender} \geq f+1 \wedge timeout^{id}_{sender}[myself] = \perp$ **then**
3: $startTimer(\langle message, sender, id\rangle)$
4: $timeout^{id}_{sender}[myself] \leftarrow \langle message, sender, id\rangle$
5: $send(\text{TO-FREEZE}, myself, \langle message, sender, id\rangle)$ *to* A'
6: **end if**
7: **if** $\#timeout^{id}_{sender} \geq 2f+1$ **then**
8: $stopProtocol()$ {Pára os *timers* e o processamento de mensagens relacionadas com o PAW.}
9: **if** $inExecution \neq -1$ **then**
10: $lastExec \leftarrow inExecution$
11: **else**
12: $lastExec \leftarrow lastExecuted$
13: **end if**
14: $l \leftarrow (sender + id + timeoutNum(\langle message, sender, id\rangle) - 1) mod\ n$
15: $waitingLeader^{id}_{sender} \leftarrow l$
16: $send(\text{TO-COLLECT}, myself, \langle message, sender, id\rangle, l, lastExec)_{\sigma\_myself}$ *to* l
17: **end if**

Ainda neste algoritmo, caso o *timeout* tenha ocorrido em pelo menos $2f+1$ servidores, o protocolo é paralisado (os *timers* associados às mensagens e aos *rounds* são parados, juntamente com o processamento de mensagens relacionadas com o algoritmo de consenso - linha 8) e uma mensagem é enviada ao novo líder, informando em que ponto se encontra o processamento de consensos neste servidor ($lastExecuted$ ou $inExecution$). As mensagens relacionadas com o protocolo de consenso, recebidas enquanto tal protocolo está parado, são armazenadas e processadas quando este protocolo é disparado (linha 11 do algoritmo 11). Além disso, este mensagem (TO-COLLECT) deve ser assinada, pois será usado pelo novo líder para provar que: *(1)* realmente é o novo líder escolhido e *(2)* o ponto de início de liderança informado é correto. Note que, se um servidor correto executar este processamento, é garantido que todos os servidores corretos também o farão.

Diferente do congelamento de *rounds* onde o novo líder é escolhido baseando-se apenas no líder do *round* congelado, por motivos de sincronização, a escolha do novo líder considera o identificador da mensagem relacionada com os *timeouts* ($sender + id$) e o número do *timeout* referente a este congelamento (função $timeoutNum$ – lembrando que os dados são armazenados separados para cada *timeout*). Isto permite que todos os servidores escolham o mesmo líder, independente do ponto de processamento em que se encontram, além de impedir que um servidor seja escolhido mais de uma vez como líder, devido a ocorrência de *timeouts* para a ordenação de uma mesma mensagem.

O novo líder espera por $2f+1$ ($n-f$) mensagens de TO-COLLECT (armazenando-as no vetor TOC, linha 2 do algoritmo 11), para então definir a partir de que instância do consenso será o líder (linha 4), a qual é determinada como sendo a seguinte a maior instância reportada como executada (ou em execução) por no mínimo $f+1$ servidores, o que exige a presença de no mínimo um servidor

correto. Sendo assim, envia uma mensagem de troca de líder (LEADER-CHANGE) para todos os servidores, contendo o seu identificador e o ponto a partir do qual será o líder, além das provas assinadas (TOC). As mensagens TO-FREEZE e TO-COLLECT são equivalentes às mensagens FREEZE e COLLECT do algoritmo PAW (ver seção 4.2), no entanto consideram as várias instâncias do PAW em execução.

Algoritmo 11 Mensagens de troca de líder

Require: $receive(\text{TO-COLLECT}, a, \langle message, sender, id\rangle, newLeader, lastExec)_{\sigma\_a}$
1: **if** $newLeader = myself$ **then**
2: $TOC^{id}_{sender}[a] \leftarrow \langle newLeader, lastExec\rangle$
3: **if** $\#TOC^{id}_{sender} \geq 2f + 1$ **then**
4: $start \leftarrow getStart(TOC)$
5: $send(\text{LEADER-CHANGE}, \langle message, sender, id\rangle, \langle newLeader, start\rangle, TOC)$ *to A*
6: **end if**
7: **end if**
Require: $receive(\text{LEADER-CHANGE}, \langle message, sender, id\rangle, \langle newLeader, start\rangle, proofs)$
8: **if** $goodLeaderStart(newLeader, start, proofs) \wedge waitingLeader^{id}_{sender} = newLeader$ **then**
9: $waitingLeader^{id}_{sender} \leftarrow \perp$
10: $setLeader(newLeader, start)$
11: $resumeProtocol(start)$ {Reinicia os *timers* e o processamento de mensagens relacionadas com o PAW.}
12: $send(\text{LEADER-CHANGE}, \langle message, sender, id\rangle, \langle newLeader, start\rangle, proofs)$ *to* A'
13: **end if**

Quando uma mensagem de troca de líder é recebida, primeiramente é verificado se o ponto de início de liderança e o identificador do líder são válidos. Além disso, é determinado se o servidor em questão estava aguardando esta mensagem de troca de líder, relacionada com a mensagem que está aguardando sua ordenação[14]. Caso estas verificações sejam confirmadas, as informações sobre o novo líder são salvas e o protocolo é disparado (*timers* são reiniciados e o processamento de mensagens relacionadas com o protocolo de consenso é retomado – linha 11). Neste mesmo processamento, considerando que o ponto de início de liderança seja a instância s do PAW, o servidor em questão inicia o *timeout* relacionado com o primeiro *round* da instância $s-1$ (caso ainda não tenha iniciado)[15], pois é garantido que pelo menos um servidor correto está executando esta instância. Deste modo, um valor não tinha sido aceito fracamente como decisão da instância $s-1$ (i.e., o *timeout* não tinha sido inicializado) por causa do não recebimento da proposta ou de $f+1$ mensagens WEAK (conforme já discutido), o que pode estar relacionado com um líder faltoso. Por fim, esta mensagem ($LEADER-CHANGE$) é enviada aos outros servidores, realizando uma espécie de "eco" (linha 12).

A necessidade deste "eco" está relacionada com o fato do protocolo ser parado durante o processamento do envio da mensagem TO-COLLECT (algoritmo 10) e apenas ser disparado quando a mensagem de troca de líder é recebida. Deste modo, mesmo que o líder escolhido seja faltoso e enviar a mensagem LEADER-CHANGE para apenas alguns servidores corretos, todos os servidores corretos acabarão por receber tal mensagem e processar a troca de líder.

No entanto, este novo líder faltoso poderá não enviar esta mensagem de troca de líder para algum servidor correto (enviando apenas para servidores faltosos ou mesmo não enviando para servidor al-

[14]Isto impede que um servidor faltoso utilize mensagens TO-COLLECT antigas para forçar uma troca de líder.

[15]Caso o servidor em questão esteja executando alguma instância anterior a $s-1$, este *timeout* é iniciado assim que tal servidor atingir esta instância.

gum), fazendo com que o protocolo entre em *deadlock*. Este problema é sanado através da associação de um *timeout* com esta fase de troca de líder, o que permitirá a escolha de outro líder caso necessário. Para que esta fase de escolha de outro líder seja iniciada é necessário que este *timeout* ocorra em pelo menos um servidor correto, o que impede a ação de servidores maliciosos. Este processamento é idêntico ao esquema de congelamento de *rounds* (processamento do mensagens FREEZE do algoritmo 2) e ao mecanismo de *timeouts* para a ordenação de uma mensagem (processamento de mensagens TO-FREEZE do algoritmo 10). Sendo assim, caso este *timeout* para troca de líder ocorra em pelo menos $f+1$ servidores corretos, todos os servidores corretos entrarão na fase de escolha de novo líder. No contrário, o processamento de troca de líder é executado por pelo menos $f+1$ servidores corretos e todos os servidores corretos acabarão por executá-lo (devido ao "eco" das mensagens LEADER-CHANGE).

É bom reforçar que o líder escolhido devido a ocorrência de um *timeout* para a ordenação de uma mensagem (ex. segundo *timeout*) deve ser diferente dos servidores que foram líderes em outros *timeouts* relacionados com esta mensagem (ex. primeiro *timeout*). Isto garante que para cada *timeout* um servidor diferente será líder, o que possibilita o descarte de mensagens oriundas de difusões incompletas (ver seção 4.3.6). Isto também deve ser considerado quando o líder escolhido é faltoso e não envia a mensagem LEADER-CHANGE, tornado necessária a determinação de outro líder.

4.3.6 Eliminando Mensagens Oriundas de Difusões Incompletas

Como já abordado, um emissor faltoso (qualquer tipo de falta) pode enviar uma mensagem para apenas alguns servidores. Estas mensagens, oriundas de difusões incompletas, devem ser descartadas para que os servidores não ocupem espaço de armazenamento com dados irrelevantes (lixo). Note que, deve-se garantir que uma mensagem será entregue (pelos servidores corretos) apenas se a mesma for enviada para todos os servidores corretos ($2f+1$), independente do líder estar entre tais servidores.

Nesta seção, analisaremos apenas quando o emissor é faltoso e envia a mensagem para no máximo f servidores corretos, não enviando para o líder. Outras situações, onde a mensagem é enviada para um número insuficiente de servidores, incluindo o líder, o algoritmo apresentado nas seções anteriores se encarregará pelo descarte de tal mensagem, que não terá sua autenticidade comprovada.

Sendo assim, quando apenas f servidores (que não seja o líder) recebem uma mensagem, não será proposta uma ordem para a mesma e ocorrerá o *timeout* para sua ordenação nestes f servidores, o que não será suficiente para dar início a fase de troca de líder. Assim, após a ocorrência de $f+2$ *timeouts* (relacionados com esta mensagem) em um mesmo servidor, tal mensagem é eliminada (algoritmo 9), pois é garantido que a mesma não foi enviada para um número suficiente de servidores.

A possibilidade de uma mensagem ser eliminada depois que $f+2$ *timeouts* ocorram em um servidor está relacionada com o fato de que, se em cada *timeout* o líder é trocado, então é garantido que $f+1$ servidores diferentes foram líderes[16]. Isto é, no mínimo um servidor correto foi líder e caso tal servidor tivesse recebido esta mensagem, uma ordem seria proposta para a mesma. Então

[16]Lembrando que, nestes casos, um mesmo servidor não é escolhido mais de uma vez como líder.

conclui-se que o emissor é faltoso e realizou uma difusão incompleta (não enviou a mensagem para todos os servidores corretos).

Note que, um servidor faltoso pode se unir aos f servidores corretos, que receberam a mensagem, para forçar a ocorrência de uma trocar de líder e até mesmo fazer com que esta mensagem seja ordenada. No entanto, para uma mensagem ser ordenada (entregue) é necessário que a mesma seja enviada para pelo menos um servidor correto (os servidores faltosos não podem inventar mensagens). Além disso, caso esta mensagem seja enviada para todos os servidores corretos ($2f+1$), sua ordenação é garantida.

É importante perceber que, caso o emissor envie uma mensagem para apenas f servidores corretos, não é garantido que ocorrerá uma troca de líder, mas mesmo assim acontecerão os $f+2$ *timeouts* em cada servidor que receber esta mensagem e a mesma será eliminada. Adicionalmente, este emissor faltoso pode ser colocado em uma *"black list"* para que suas difusões futuras sejam ignoradas.

4.3.7 Otimizações no Algoritmo

Com o intuito de melhorar o desempenho do protocolo de difusão atômica, duas otimizações foram incorporadas aos algoritmos apresentados nas seções anteriores. Estas otimizações preservam as propriedades de segurança e vivacidade do sistema.

Execução em lotes. Com esta otimização, cada instância do consenso não ordena apenas uma, mas um lote de mensagens. Isto tende a melhorar a escalabilidade do sistema, permitindo um bom desempenho mesmo em condições elevadas de carga [18, 32]. Nestas execuções, todo o processamento envolvido na ordenação de uma mensagem, como por exemplo a determinação da autenticidade da mesma, deve ser estendido as demais mensagens do lote. Sendo assim, se apenas uma mensagem do lote não for autêntica, a ordenação de todo o lote estará comprometida. Após o lote ser ordenado, as mensagens devem ser entregues de forma determinista, i.e, de acordo com a ordem dentro do lote.

Resumos criptográficos. O objetivo desta otimização é reduzir o consumo de largura de banda e os gastos com processamentos, principalmente quando o tamanho das mensagens for grande. Deste modo, todas as mensagens relacionadas com o algoritmo PAW possuem apenas os resumos criptográficos das mensagens que estão sendo ordenadas, o que diminui consideravelmente o tamanho das mesmas.

No entanto, o uso de resumos criptográficos possibilita que uma mensagens m seja ordenada antes mesmo de ser recebida em algum servidor correto s, pois é o resumo criptográfico ($H(m) = d$) de m que está sendo ordenado. Além disso, um emissor faltoso pode não enviar m para s e ainda assim m ser ordenada e entregue, como discutido na seção 4.3.6. Deste modo, como s conhece d, poderá recuperar m ($H(m) = d$) de outro servidor.

Para garantir que uma mensagem seja recuperada, os servidores não poderão eliminá-la assim que for entregue. Deste modo, o descarte de uma mensagem é realizado apenas quando todos os servidores entregaram tal mensagem. Sendo assim, as mensagens ordenadas pela instância i do PAW

são eliminadas apenas quando todos os servidores enviarem alguma mensagem relacionada com a instância $i+1$, isto garante que todos os servidores corretos já entregaram as mensagens ordenadas em i. Esta abordagem causa outro problema, pois basta que um servidor seja faltoso e não envie mensagens relacionadas com o protocolo PAW para que os servidores fiquem impossibilitados de eliminar as mensagens já entregues, fazendo com que o *buffer*, que contém tais mensagens, cresça indefinidamente.

A solução deste problema consiste em fazer com que, quando o tamanho deste *buffer* ultrapassar um limite, as mensagens nele armazenadas sejam encaminhadas aos servidores que ainda não as entregaram (ainda não foi possível determinar se entregaram), descartando-as. Como utilizamos canais ponto a ponto confiáveis, estas mensagens acabarão por serem recebidas nestes servidores. Note que, tais mensagens serão encaminhadas para no máximo f servidores. Isto é determinado pelo fato de que os servidores executam apenas uma instância do consenso (PAW) por vez, não iniciando a instância $i+1$ antes de decidir i. Deste modo, sempre teremos $2f+1$ servidores executando ou a instância i ou a $i+1$ (ver seção 4.3.1).

4.4 Alguns Cenários de Falhas

Nesta seção, alguns cenários de falhas são analisados. O objetivo é mostrar o comportamento do protocolo em determinadas situações de falhas, visando a garantia das propriedades do sistema. A seguir, quatro casos são abordados:

- *Líder faltoso que não propõe a ordem para uma mensagem*: Neste caso, acontecerá o *timeout* para a ordenação desta mensagem e o líder será trocado conforme discutido na seção 4.3.5.
- *Líder faltoso que propõe diferentes ordens para algumas mensagens dentro de um lote*: Neste caso, o comportamento é como se o valor proposto não fosse o mesmo entre os diferentes servidores. Se cada servidor receber uma proposta diferente (com as ordens diferentes dentro do lote) o consenso não terá progresso, então o *round* será congelado e um novo líder será escolhido. No entanto, se a mesma proposta for enviada para no mínimo $2f+1$ servidores, a mesma será a decisão do consenso e as mensagens serão entregues, na ordem definida nesta proposta, por todos os servidores corretos.
- *Emissor faltoso que não envia a mensagem para todos os servidores*: Este caso ilustra uma difusão incompleta. Sendo assim, esta mensagem poderá ou não ser entregue, conforme discutido na seção 4.3.6.
- *Emissor faltoso que envia diferentes mensagens para diferentes servidores*: Neste caso, cada mensagem diferente será tratada como uma mensagem oriunda de uma difusão incompleta (ver seção 4.3.6).

4.5 Trabalhos Relacionados

Existem muitos estudos que envolvem o algoritmo PAXOS BIZANTINO, tanto relacionados com o aprimoramento do algoritmo em si (buscando o aperfeiçoamento de suas características) quanto no desenvolvimento de outros protocolos a partir deste algoritmo (normalmente protocolos de difusão atômica). A abordagem mais completa sobre o algoritmo PAXOS BIZANTINO é apresentada em [35], onde este algoritmo é parametrizado e pode ser usado em ambientes sujeitos tanto a faltas de parada quanto a faltas bizantinas. No entanto, este trabalho não fornece a sua transformação para um protocolo de difusão atômica.

Outra variante deste algoritmo é apresentada em [58], onde ocorre uma distinção das classes de processos definidas para o PAXOS (proponentes, aceitantes e aprendizes). Esta abordagem é capaz de decidir o consenso em dois passos de comunicação até mesmo na presença de faltas, desde que estas não ocorram no líder. Para isso, é requerido um número maior de servidores: $n \geq 3f + 2t + 1$, onde n é o número total de servidores, f é o número de faltas toleradas para garantir segurança (*safety*) e t é o número de faltas toleradas para terminar o consenso em apenas dois passos de comunicação, sendo que $t \leq f$. Neste mesmo trabalho, é provado que qualquer algoritmo capaz de resolver o consenso em apenas dois passos de comunicação (mesmo na presença de faltas) requer no mínimo $n \geq 5f + 1$ servidores.

Dentre as variações do protocolo PAXOS BIZANTINO encontradas na literatura, as abordagens estudadas em [26, 83] são as mais semelhantes com a apresentada neste texto. No entanto, suas transformações para protocolos de difusão atômica possuem diferentes características (destacadas a seguir) das obtidas neste trabalho.

O mecanismo de difusão atômica proposto em [83] é baseado na primitiva de consenso vetorial [34], onde cada processo (servidor) propõe um valor para a sua posição no vetor e todos os processos precisam entram em acordo sobre este valor. Deste modo, existe a necessidade da execução de uma instância do algoritmo de consenso para cada servidor do sistema, apresentando uma complexidade de troca de mensagens na ordem de n^3. A abordagem empregada neste texto executa apenas uma instância do consenso para cada mensagem a ser ordenada (não considerando execuções em lote), tendo complexidade de troca de mensagens na ordem de n^2. Esta é uma característica muito importante, principalmente se considerarmos ambientes com pouca largura de banda, número grande de réplicas (servidores) e grande variabilidade na latência das comunicações.

O primeiro protocolo de difusão atômica baseado no algoritmo PAXOS BIZANTINO, que é apresentado em [26], foi projetado para ambientes assíncronos (sujeitos a faltas bizantinas). Neste trabalho foi introduzido o algoritmo PAXOS BIZANTINO, através do desenvolvimento de uma biblioteca de replicação Máquina de Estados [68]. O protocolo de difusão atômica abordado em [26] não considera as alterações para terminação rápida apresentadas em [58, 83], as quais são contempladas no protocolo descrito neste texto. Além disso, o mecanismo desenvolvido em [26] utiliza *checkpoints*, os quais são necessários em vários pontos do protocolo: na coleta de lixo, na recuperação (atualização) do estado das réplicas (servidores), na troca de líder. Estes *checkpoints* contém o identificador da última requisição cuja execução está refletida no estado representado pelo mesmo. Dependendo do

número (período) de *checkpoints*, o espaço necessário para armazenar tais *checkpoints* pode não ser desprezível, bem como as trocas de mensagens contendo *checkpoints*. Além disso, os *checkpoints* devem ser assinados, pois os mesmos são usados como provas pelos processos. Estas assinaturas são realizadas através de um esquema de vetor de MACs (*message authentication code*), o que diminui o impacto causado por este mecanismo no desempenho deste protocolo. Outra diferença significativa é que este protocolo não assume canais confiáveis (as trocas de mensagens são realizadas através de UDP/IP e UDP/*Multicast* IP), enquanto que a abordagem apresentada neste texto utiliza canais ponto a ponto confiáveis (TCP/IP). Versões mais novas deste sistema já se utilizam de canais confiáveis (TCP/IP) [32].

4.6 Considerações Finais

Neste capitulo foi definido um protocolo de difusão atômica baseado no algoritmo PAW, que é uma variante do protocolo PAXOS para sistema sujeitos a faltas bizantinas. A primeira parte do capítulo detalha o algoritmo PAW, discutindo os principais pontos envolvidos na preservação das propriedades do consenso (ver seção 2.2.1).

O estudo do algoritmo PAW é importante para facilitar o entendimento da segunda parte do capítulo, que apresenta o protocolo de difusão atômica elaborado sobre este algoritmo, o qual sofreu algumas alterações. Além disso, diversos mecanismo foram desenvolvidos para garantir as propriedades exigidas na difusão atômica (ver seção 2.2.2), mostrando que apesar da idéia geral sobre o protocolo ser simples, a concretização do mesmo não é trivial.

Capítulo 5

Infra-Estrutura com Segurança de Funcionamento para Coordenação de Serviços *Web* Cooperantes

Este capítulo apresenta o WS-DEPENDABLESPACE, que é uma infra-estrutura com segurança de funcionamento usada na coordenação de serviços *web*. Além disso, esta mesma infra-estrutura também pode ser usada, como um repositório de dados seguro e confiável, por serviços *web* cooperantes. Primeiramente, alguns conceitos relacionados com a arquitetura orientada a serviços e com a arquitetura dos serviços *web* são abordados. Após isso, a arquitetura do WS-DEPENDABLESPACE é discutida e os vários mecanismos desenvolvidos para o funcionamento deste sistema são apresentados. Além disso, algumas aplicações que podem utilizar este modelo para a coordenação e/ou cooperação de serviços *web* são apresentadas. Este capítulo ainda discute a relação desta infra-estrutura com algumas especificações para serviços *web* encontradas na literatura.

5.1 Contexto e Motivação

A tecnologia de *Web Services*, ou serviços *web*, tem se consolidado cada vez mais como um padrão *de facto* quando se consideram sistemas distribuídos na Internet. Esta tecnologia concretiza o modelo de computação orientada a serviços [19] sobre padrões largamente utilizados na *web*. A adoção desses padrões proporciona aos serviços *web* grande facilidade de uso e baixo custo de implantação. Isto se reflete no suporte maciço da indústria a esta tecnologia.

O grande atrativo dos serviços *web* é sua interoperabilidade e simplicidade devido ao uso de um modelo conhecido (invocação remota de operações) sobre tecnologias largamente utilizadas (notadamente, HTTP e XML). Sendo a interoperabilidade um dos pontos fundamentais dos serviços *web*, não tardou para que surgissem esforços buscando definir formas adequadas de se combinar serviços, visando a cooperação na execução de tarefas envolvendo várias organizações [19, 65]. Iniciativas como

o WS-ORCHESTRATION [4] e o WS-CHOREOGRAPHY [22] atacam justamente esse problema, propondo mecanismos para a definição e execução de tarefas complexas que envolvem várias subtarefas encapsuladas em serviços *web*.

Iniciativas como as listadas acima se concentram na integração dos serviços *web* através da especificação de um fluxo de troca de mensagens entre os mesmos (coordenação orientada a controle [63]). Uma abordagem complementar é a coordenação dos serviços usando um repositório de dados compartilhado (coordenação orientada a dados [63]). Nessa abordagem, os serviços cooperantes se comunicam através do uso de um repositório de dados que pode ser usado tanto para o armazenamento de dados compartilhados quanto como um mediador, oferecendo comunicação desacoplada.

Neste contexto, um modelo de coordenação orientada a dados bastante popular é o baseado em *espaço de tuplas* (ver seção 2.3). Os benefícios do uso deste modelo estão relacionados com suas características de desacoplamento no tempo, no espaço e ao seu poder de sincronização (ex. controle de concorrência). Diversos trabalhos têm explorado o espaço de tuplas como infra-estrutura de coordenação para serviços *web* (ex. [11, 21, 56, 57]), sendo que o desacoplamento entre os serviços cooperantes (nem as interfaces precisam ser conhecidas) é a principal vantagem desta abordagem.

Além disso, devido as características do ambiente para o qual os serviços *web* são endereçados (Internet), é grande a necessidade de mecanismos que possibilitem uma comunicação e coordenação (entre estes serviços) de forma segura e confiável. Muitas vezes, estes serviços estão relacionados com a realização de negócios coorporativos, onde circulam informações sigilosas, sendo necessário garantir a segurança destas informações. Neste sentido, a nossa proposta segue a mesma linha dos trabalhos que exploram as facilidades relacionadas com o modelo de coordenação através de espaços de tuplas, mas propõem um suporte de cooperação para serviços *web* que também é seguro e confiável.

A infra-estrutura aqui proposta, chamada WS-DEPENDABLESPACE [3], doravante apenas chamada de WSDS, tem como base o espaço de tuplas com segurança de funcionamento descrito no capítulo 3 e incorpora novos componentes, que proporcionam a integração do modelo ao mundo dos serviços *web*. A arquitetura do WSDS faz uso de *gateways* "sem estado" que agem como clientes de um espaço de tuplas confiável, repassando as requisições advindas de clientes do serviço de coordenação. Diversos mecanismos foram introduzidos nesta arquitetura para permitir que o sistema tolere faltas acidentais (paradas e *bugs*) e maliciosas (ataques e intrusões) em uma parte dos componentes do sistema. Além disso, o WSDS mantém todas as propriedades de segurança do espaço de tuplas confiável (DEPSPACE) sem ferir nenhuma especificação para serviços *web*.

5.2 Arquitetura Orientada a Serviços

A arquitetura orientada a serviços (*Service Oriented Architecture* – SOA) utiliza o conceito de serviços como elemento fundamental na concepção de aplicações. Este paradigma de programação – e mais especificamente os serviços *web* – é uma evolução natural de conceitos como RPC e tecnologias como o CORBA [62]. De forma semelhante a estas abordagens, os serviços providos por uma

organização devem ser descritos numa interface que pode ser entendida e invocada por outras partes em um sistema distribuído, por meio de um formato padrão.

Para isso, esta arquitetura é constituída de três participantes [64]:

- Agência de Registro de Serviços: esta agência nada mais é do que um repositório usado para publicar e localizar as interfaces dos serviços. Estas interfaces são auto-descritivas e baseadas em padrões abertos, i.e., definem os métodos públicos, juntamente com seus parâmetros, valores de retorno e meios de tratar exceções (não fornecendo a implementação).
- Provedor de Serviços: o provedor de serviços é responsável por publicar as interfaces dos seus serviços (em um formato padrão) e atender às requisições dos clientes.
- Cliente: o cliente pode ser uma aplicação ou um outro serviço que efetua requisições a um serviço. As funcionalidades sobre o serviço são conhecidas através da descrição disponibilizada pelo provedor de serviços, recuperada por meio de uma pesquisa na agência de registro de serviços.

As interações entre estes elementos podem ser observadas na figura 5.1. Em uma seqüência normal de interações, o provedor de serviços define a descrição dos serviços oferecidos e a publica na agência de registro de serviços. Após isso, o cliente recupera a descrição do serviço no repositório de interfaces, podendo especificar as características desejadas. Deste modo, o cliente receberá a interface e a localização do serviço, podendo então interagir com o provedor de serviços.

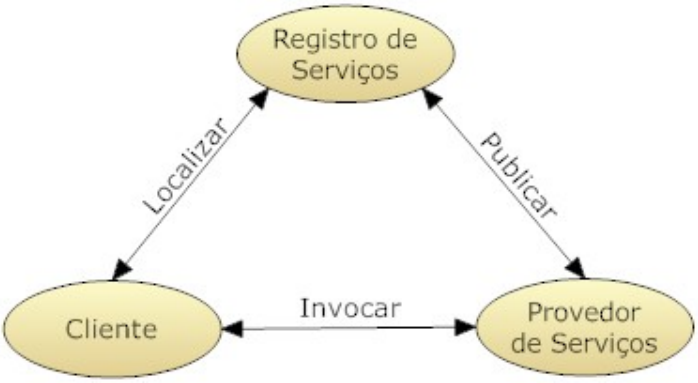


Figura 5.1: Interação entre os elementos da SOA.

Através deste mecanismo, um provedor de serviços pode fornecer serviços tanto para aplicações (de usuários finais) como para outros serviços distribuídos pela Internet, possibilitando a composição de serviços.

Como o fornecimento destes serviços está baseado na troca de mensagens entre os provedores dos mesmos e os clientes, as mensagens devem seguir um formato padrão. Além disso, o fornecimento de serviços deve ser [64]:*(i.)* independente de implementações, os serviços devem ser acessados através de tecnologias padrões; *(ii.)* fracamente acoplados, os clientes e provedores não devem precisar conhecer a estrutura interna um do outro, *(iii.)* transparente em relação a localização, os serviços devem

ter sua localização armazenada em um repositório e acessado pelos clientes independentemente de sua localização.

5.3 Arquitetura dos Serviços *Web*

Os serviços *web* são aplicações identificadas por meio de uma URI[1] (*Uniform Resource Identifier*), que possuem interfaces definidas e descritas em XML (*eXtensible Markup Language*). As interações com outras aplicações são concretizadas através de trocas de mensagens XML utilizando protocolos padrões da Internet [78].

Em geral, os serviços *web* (e a arquitetura orientada a serviços como um todo) são mais apropriados para aplicações onde [78]:

- É necessário operar sobre a Internet, onde velocidade e confiabilidade não pode ser garantida.
- Não é possível administrar *upgrade* de versões, então a estrutura dos clientes e fornecedores de serviços sofre alterações freqüentemente.
- Os componentes do sistema distribuído são executados em diferentes plataformas, com produtos fornecidos por diferentes empresas.
- Uma aplicação existente precisa ser exportada para uso na Internet e pode ser empacotada como um serviço *web*.

Os padrões fundamentais usados pelos serviços *web*, todos baseados em XML, são: o SOAP (*Simple Object Access Protocol*) – protocolo para troca de mensagens entre clientes e serviços, operando sobre diversos protocolos de comunicação; o WSDL (*Web Service Description Language*) – linguagem para descrição de serviços *web*; e o UDDI (*Universal Description, Discovery and Integration*) – repositório onde os serviços *web* são registrados para que possam ser descobertos por clientes. Por usar apenas elementos baseados em padrões, os serviços *web* facilitam a comunicação entre aplicações que residem em múltiplas plataformas e que utilizam diferentes modelos de objetos baseados em linguagens diferentes.

A figura 5.2 apresenta um cenário típico de interações entre os elementos da arquitetura dos serviços *web*. Inicialmente, o processo que deseja tornar um serviço *web* disponível deve descrever a interface do serviço, utilizando a WSDL, e publicá-la em um serviço de busca público, como por exemplo o UDDI (passo 1). A partir de então, o cliente que desejar acessar este serviço deve localizá-lo no repositório e obter a sua WSDL (passos 2 e 3). Por fim, o cliente obtém acesso ao serviço e comunica-se com o provedor através de trocas de mensagens XML, encapsuladas dentro de envelopes SOAP (passo 4). As várias tecnologias empregadas na arquitetura dos serviços *web* são brevemente descritas a seguir.

[1] Uma URI é o formato para identificar um recurso abstrato ou concreto, que pode ser basicamente qualquer coisa com uma identidade [79].

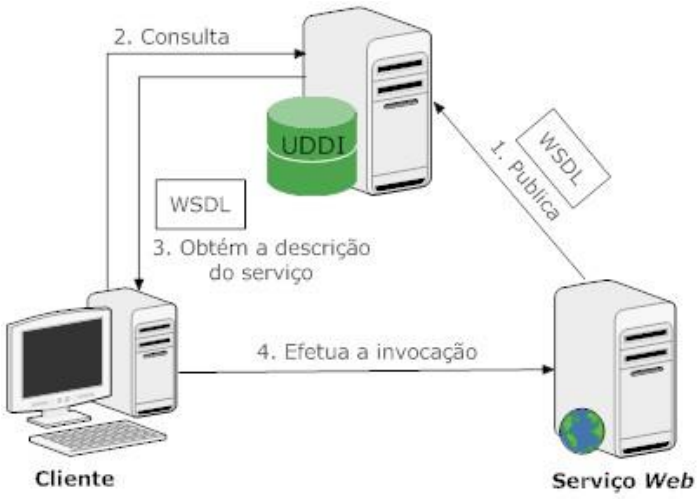


Figura 5.2: Arquitetura dos serviços *web*.

5.3.1 O padrão XML

O padrão XML (*eXtensible Markup Language*) não é exatamente uma linguagem, mas uma metalinguagem, usada na definição de novas linguagens. Este padrão é independente de plataforma e pode ser usado para representar o conteúdo de muitas linguagens para transferência de dados. Devido a estes fatores e ao grande suporte a este padrão oferecido pelas empresas, o XML se tornou de fato um formato para troca de dados entre aplicações na Internet [79].

Pode-se definir novas linguagens a partir do XML, através de alguns conceitos definidos neste padrão como: elementos, atributos, comentários, texto literal e documento [79]. As várias tecnologias utilizadas na arquitetura dos serviços *web* podem ser entendidas como linguagens XML [79].

Além dos aspectos discutidos neste texto, existem vários outros componentes envolvidos com este padrão como o *XML Schema* e o *XML Namespaces* [79], que são usados na definição destas linguagens de transferência.

5.3.2 Linguagem de Descrição de Serviços *Web* - WSDL

Esta linguagem é uma gramática em XML, usada na especificação das interfaces dos serviços *web*, onde é definido quais são os serviços oferecidos e como os clientes e servidores irão processar as requisições. Através desta linguagem é realizada a descrição das características funcionais dos serviços *web*, i.e., quais as ações são realizadas pelo serviço em relação as mensagens recebidas e enviadas. Além disso, especifica o formato em que as informações devem ser enviadas e como o cliente pode se comunicar com o provedor do serviço.

Um documento WSDL é composto por duas partes [79]: uma de definições abstratas e outra de definições concretas. A parte abstrata descreve as ações do serviço *web* em relação as mensagens SOAP de forma independente de plataforma e linguagem de programação. Já a parte concreta está relacionada com a implementação e define como e onde os serviços são oferecidos.

5.3.3 Serviço de Registro e Divulgação de Serviços *Web* - UDDI

O serviço UDDI (*Universal Description, Discovery and Integration*) [59] especifica como é realizada a publicação e descoberta de serviços (como funciona o serviço de registro). Os clientes podem consultar um serviço UDDI, cuja localização é conhecida, para localizar serviços compatíveis com seus requerimentos. Além disso, estes clientes podem enviar requisições a este serviço com o objetivo de obter informações detalhadas sobre os serviços *web*.

Os mecanismos que possibilitam a descoberta de serviços são fundamentais dentro da arquitetura orientada a serviços, sendo que estes mecanismos podem ser especificados de diferentes formas. No entanto, no contexto dos serviços *web*, o serviço UDDI apresenta-se como um padrão flexível e funcional para a descoberta de serviços.

5.3.4 O protocolo SOAP

O protocolo SOAP (*Simple Object Access Protocol*) fornece uma estrutura padrão e extensível, utilizada para o empacotamento e trocas de mensagens XML. Com relação a esta arquitetura, o SOAP também fornece mecanismos convenientes para especificar as capacidades dos comunicantes (tipicamente através do uso de cabeçalhos).

O SOAP pode operar sobre diversos protocolos já consolidados, como o HTTP, o SMTP, o FTP, etc. No entanto, o configuração mais comum nas implementações de serviços *web* é o SOAP sobre o HTTP. Isto se deve as facilidades fornecidas pelo HTTP, como toda a infra-estrutura de servidores já existente e a capacidade de atravessar os limites de segurança impostos por *firewalls*.

5.4 Vantagens da Coordenação de Serviços *Web* através de Espaços de Tuplas

A principal vantagem de utilizar um espaço de tuplas na coordenação de serviços *web* é o desacoplamento. Nas aplicações *web* padrões, temos um forte acoplamento em pelo menos dois sentidos [23]: *(1)* para completar uma iteração, os serviços cooperantes precisam estar ativos e executando ao mesmo tempo; e *(2)* as interações são programadas com base nas interfaces dos serviços, caso alguma interface sofra alterações é necessário reprogramar estas iterações.

Deste modo, com o auxilio de um serviço de coordenação por espaço de tuplas, um serviço qualquer (cliente de outro(s) serviço(s)) pode enviar uma requisição e terminar. Após algum tempo, este serviço pode ser reativado para consumir a resposta de sua requisição, que foi produzida enquanto tal serviço estava desativado. Isto quer dizer que os serviços não precisam estar ativos ao mesmo tempo, além de não precisarem conhecer suas localizações, isto é conhecido como desacoplamento no tempo e no espaço, respectivamente.

Além disso, como as interações ocorrem através de um mediador (espaço de tuplas), não é necessário que os serviços conheçam as interfaces uns dos outros. Sendo assim, quando algum serviço sofrer alterações em sua interface não será necessário reprogramar as interações envolvendo tal serviço.

Outro ponto a destacar é que este tipo de coordenação pode ser usado como mecanismo de sincronização entre vários serviços *web* cooperantes (ex., no controle da concorrência entre estes serviços). Como já comentado, o WSDS implementa uma infra-estrutura de coordenação que fornece todas estas vantagens de forma segura e confiável.

5.5 WS-DEPENDABLESPACE

Esta seção descreve o WSDS. Primeiramente, algumas premissas e garantias são apresentadas. Após isso, a arquitetura e os princípios básicos de funcionamento do sistema são abordados, para então discutirmos os mecanismos específicos integrados à arquitetura, os quais visam a solução de alguns problemas de segurança que podem ocorrer.

5.5.1 Premissas e Garantias

Esta seção reforça as premissas assumidas no modelo de sistema apresentado na seção 2.1, destacando e/ou incluindo as premissas relacionadas com o WSDS.

Sendo assim, os processos do sistema são divididos em três conjuntos: n_r servidores DEPSPACE $U = \{s_1, ..., s_{n_r}\}$, n_g *gateways* de acesso $G = \{g_1, ..., g_{n_g}\}$ e um conjunto ilimitado de clientes $\Pi = \{c_1, c_2, ...\}$. Os *gateways* são os únicos processos do sistema que necessariamente exportam interfaces WSDL, sendo portanto, serviços *web*. Os clientes se comunicam com os *gateways* através de mensagens SOAP, que operam sobre o protocolo HTTP, garantindo que esta comunicação seja confiável. Os *gateways* se comunicam com os servidores através de canais confiáveis e autenticados, que também são utilizados na comunicação entre os servidores. Estes canais confiáveis e autenticados estão fora do escopo dos padrões de serviços *web*.

Assumimos também que cada processo do sistema tem um par de chaves pública-privada, sendo a chave privada conhecida apenas pelo próprio processo. As chaves públicas de todos os processos do sistema são conhecidas por todos os outros processos (através de certificados). Essas chaves são usadas na produção e verificação de assinaturas digitais, nos mesmos moldes do RSA [67].

Em termos de garantias, o WSDS permanece correto (provendo um espaço de tuplas que satisfaz os atributos de segurança de funcionamento) enquanto no máximo $f_r \leq \lfloor \frac{n_r+1}{3} \rfloor$ servidores DEPSPACE (menos de um terço, o ótimo para esse tipo de replicação [26]), $f_g \leq n_g - 1$ *gateways* (no mínimo um correto) e um número ilimitado de clientes falhem.

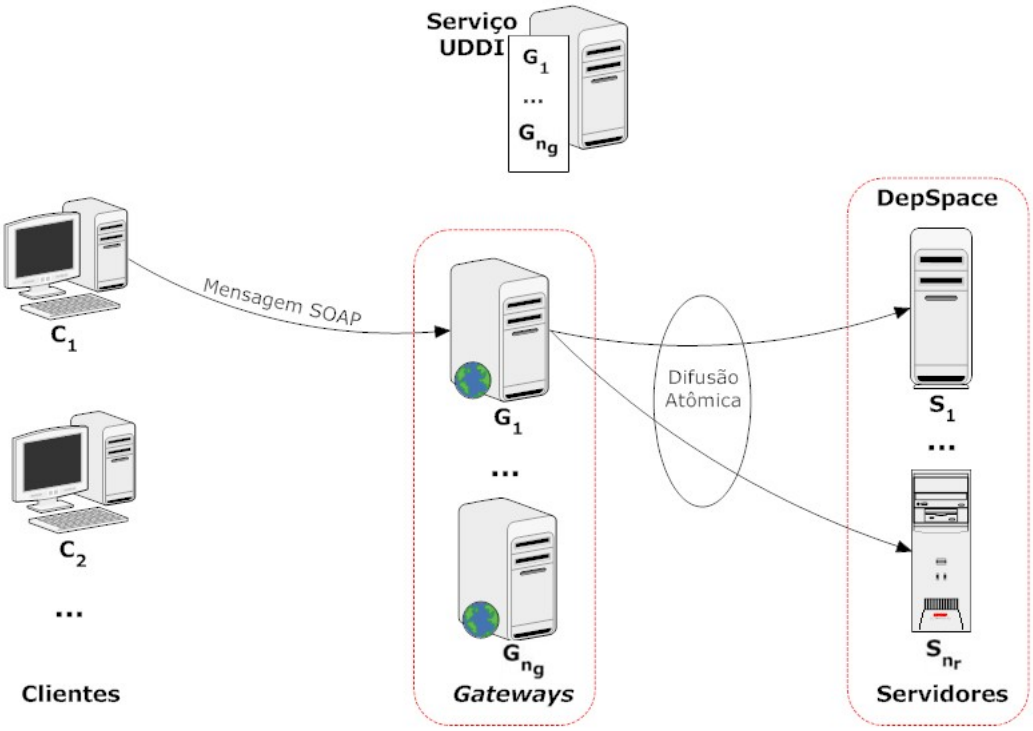


Figura 5.3: Arquitetura e príncipio de funcionamento do WSDS.

5.5.2 Arquitetura e Princípio de Funcionamento

A figura 5.3 apresenta a arquitetura do WSDS, onde é possível observar que os *gateways* ligam os clientes aos servidores do DEPSPACE. Para isso, disponibilizam a interface de seu serviço em um repositório UDDI para que os clientes possam acessá-los. Conforme já discutido, os clientes e os servidores têm pares de chaves públicas e privadas. As chaves públicas de todos os processos são disponibilizadas através de certificados facilmente verificáveis.

O principal componente introduzido nesta arquitetura é o *gateway web service* (WSG), doravante chamado apenas *gateway*, o qual é um serviço *web* que funciona como "ponte" entre os clientes (deste serviço) e o espaço de tuplas replicado (DEPSPACE), recebendo as mensagens SOAP vindas destes clientes e transformando-as em invocações ao DEPSPACE.

Deste modo, quando um cliente deste serviço for acessar o espaço de tuplas, primeiramente deve consultar um serviço UDDI (disponível na Internet) visando obter um ou mais endereços de *gateway*[2]. A partir daí, o cliente envia sua requisição para um dos *gateways*, o qual, por sua vez, a encaminha ao DEPSPACE usando um protocolo de difusão atômica (ver seção 4.3). Após o processamento da requisição, os servidores respondem ao *gateway* que espera por $2f_r + 1$ ($n_r - f_r$) respostas para então encaminhá-las ao cliente. O cliente obtém a resposta da requisição verificando (na resposta do *gateway*) qual resposta foi enviada por $f_r + 1$ servidores.

O *gateway* não realiza nenhum processamento com o conteúdo das requisições ou das respostas,

[2]Para melhorar a disponibilidade do sistema, os *gateways* podem ser registrados em mais de um serviço UDDI e/ou em um serviço que seja tolerante a faltas bizantinas.

a não ser as transformações entre os dois "mundos" (SOAP e Java) descrita anteriormente e de reunir as $n_r - f_r$ respostas à requisição antes de enviá-las ao cliente. Além disso, este serviço não possui estado (*stateless*), sendo portanto desnecessário realizar qualquer sincronização entre os n_g *gateways* do sistema.

5.5.3 Lidando com *Gateways* Falhos

Apesar da aparente simplicidade da arquitetura, com a utilização deste elemento "ponte" surgem alguns problemas quando consideramos um sistema tolerante a faltas e a intrusões. Estes problemas são descritos nesta seção, juntamente com a solução empregada em suas resoluções.

5.5.3.1 Autenticidade e Integridade das Requisições e das Respostas

Um primeiro problema que surge na arquitetura da figura 5.3 reside no fato de que um *gateway* faltoso pode solicitar a execução de falsas requisições (que não foram enviadas por algum cliente). Além disso, poderá alterar requisições, modificando alguns de seus dados (parâmetros, tipo de operação, etc...). Da mesma maneira, poderá forjar ou modificar as respostas dos servidores. A questão principal aqui é garantir que somente requisições (inalteradas) decorrentes de invocações de clientes e respostas (inalteradas) produzidas por servidores sejam processadas.

Outro ponto a destacar, é que o cliente precisa enviar suas credenciais junto com as requisições, pois as mesmas são necessárias para que os servidores verifiquem se o mesmo possui permissão para acesso ao espaço e/ou à tupla acessada. Estas credenciais são enviadas aos servidores através de um certificado digital relacionado com o cliente, o qual também é utilizado para provar a autenticidade das requisições. Deste modo, cada cliente deverá obter um certificado digital (junto a uma autoridade certificadora reconhecida) e assinar suas requisições. Os servidores apenas executarão determinada operação se esta assinatura for válida de acordo com o certificado correspondente (enviado pelo cliente em sua primeira mensagem para o servidor).

Com o intuito de provar que uma resposta é autêntica, a mesma deve ser assinada pelos servidores com sua chave privada. Assim, os clientes poderão verificar a autenticidade das respostas utilizando as chaves públicas dos servidores, também contidas em certificados válidos que acompanham as respostas. As assinaturas das requisições e das respostas garantem a autenticidade e a integridade na comunicação fim-a-fim, i.e., nada pode ser alterado pelo *gateway* sem ser detectado pelos servidores do DEPSPACE ou clientes. As verificações dos certificados que acompanham tanto as requisições como as respostas podem ser feitas com procedimentos padrões (ex. PKI X.509).

5.5.3.2 Atendimento Incompleto de Requisições

Nem sempre um cliente consegue que sua requisição seja corretamente executada (atendida) caso esteja usando um *gateway* faltoso. Para que isso ocorra, basta que este *gateway* não envie as respostas

ao cliente, fazendo com que o mesmo fique bloqueado indefinidamente à espera destas. Além disso, na execução de operações de remoção de tuplas, o *gateway* acessado poderá remover as tuplas do DEPSPACE e não enviar as respostas ao cliente, fazendo com que a tupla "desapareça" do espaço sem ser consumida por cliente algum. A solução completa deste problema de desaparecimento de tuplas envolve a combinação do mecanismo descrito nesta seção com o de eliminação de requisições duplicadas (seção 5.5.3.3).

Para resolver estes problemas, um *timeout* é associado a cada envio de requisição no cliente. Caso acontecer o *timeout* de uma requisição e a resposta ainda não foi obtida, o cliente solicita a execução desta requisição a outros f_g *gateways*, garantindo que acessou pelo menos um *gateway* correto. Deste modo, a execução desta requisição estará completa quando o cliente receber o primeiro conjunto de respostas válidas de um *gateway* e conseguir determinar a resposta. Este mecanismo deverá ser acionado sempre que uma resposta não possa ser determinada, seja pelo *timeout* ou por falhas nas verificações de assinaturas dos servidores.

No caso das operações bloqueantes (*rd* e *in*) não é possível determinar se o cliente ainda não recebeu as respostas pelo fato do *gateway* acessado ter falhado ou não existir uma tupla no espaço que combine com o molde usado na operação, o que impossibilita o uso de *timeouts* para estas operações. A solução para este problema é fazer com que o cliente envie estas solicitações para $f_g + 1$ *gateways* diferentes e determine a resposta como descrito anteriormente (casos onde ocorre o *timeout* para a execução de uma requisição). O funcionamento correto destas operações depende, é claro, de que se assuma comunicações confiáveis entre clientes e *gateways* (ver seção 5.5.1).

5.5.3.3 Requisições Duplicadas

Com o mecanismo descrito na seção anterior é possível perceber que, como a requisição poderá ser enviada a mais de um *gateway*, possivelmente será enviada mais de uma vez aos servidores DEPSPACE. Além disso, um *gateway* faltoso poderá solicitar a execução de uma operação já executada pelo DEPSPACE (ataque de *replay*). Nestes casos, é necessário que estas requisições duplicadas sejam eliminadas para manter a consistência do estado do espaço de tuplas. Para que isto seja possível, o cliente deve identificar cada requisição de forma única, através de sua identidade e de um número de seqüência, assim os servidores poderão verificar se tal requisição pode ser executada.

Para este fim, cada servidor contém um *buffer* onde são armazenadas as respostas correspondentes a última requisição de cada cliente. Assim, uma requisição é executada por um servidor apenas se ela tem um número de seqüência uma unidade maior que a invocação cuja resposta está no *buffer*. Caso a requisição recebida pelos servidores tenha o mesmo identificador da última executada para este cliente, apenas a resposta armazenada no *buffer* é enviada ao *gateway* solicitante. Em qualquer outro caso a requisição é descartada. Deste modo, um cliente não pode solicitar a execução de uma requisição sem que a anterior esteja completamente atendida[3]. Este mecanismo, além de evitar que

[3]Note que esta limitação pode ser relaxada para k requisições se o servidor armazenar as últimas k respostas a cada cliente.

uma requisição seja executada mais de uma vez, resolve o problema do desaparecimento de tuplas devido ao atendimento incompleto de requisições, pois quando o cliente solicitar a reexecução de uma operação através de outro *gateway*, a mesma resposta (com a mesma tupla) estará nos *buffers* dos servidores pronta para ser enviada.

5.5.3.4 Confidencialidade

Esta propriedade é necessária para que o sistema tenha segurança de funcionamento (ver seção 3.3), sendo provida pelo DEPSPACE (ver seção 3.3.3) e considerada no projeto desta arquitetura. Para garantir que esta propriedade seja cumprida, devemos impedir que alguns componentes do sistema tenham acesso aos *shares* das tuplas, especialmente os *gateways* e clientes não autorizados. Além disso, como no DEPSPACE, um servidor apenas consegue acessar o *share* a ele endereçado.

Nas operações de escrita, os *shares* são enviados cifrados aos servidores, sendo estes os únicos elementos do sistema que poderão acessá-los (ver seção 3.3.3). Então, os cuidados com a confidencialidade estão relacionados com as respostas das operações de leitura, pois caso os servidores respondessem a estas operações com os *shares* "limpos", um *gateway* faltoso poderia acessar estes *shares* e recuperar a tupla (podendo revelá-la a partes não autorizadas). Deste modo, nas respostas das operações de leitura, os *shares* das tuplas devem ser cifrados para que apenas o cliente solicitante possa acessá-los. Estas cifragens são realizadas através de chaves simétricas obtidas a partir de segredos compartilhados entre o cliente e cada servidor.

Note que, o processamento descrito nesta seção também é necessário no DEPSPACE para evitar *eavesdropping* das respostas. Neste sentido, esta discussão também é útil para entendermos melhor o mecanismo de confidencialidade descrito na seção 3.3.3.

5.6 Implementação

Todas as implementações foram realizadas utilizando a linguagem de programação Java e a implementação do DEPSPACE apresentada no capítulo 3. Deste modo, as operações criptográficas utilizadas neste sistema são as mesmas usadas pelo DEPSPACE, as quais são apresentadas na seção 3.4.

Além disso, os *gateways* foram concretizados sobre o *Axis* 1.3 [6], uma implementação livremente disponível do protocolo SOAP, que fornece um conjunto de APIs que facilitam o desenvolvimento de aplicações clientes e de serviços *web*. Estes *gateways* são instalados no container J2EE *Tomcat* 5.5.20 [5], que é um servidor de HTTP robusto e completo, também possuindo licença de código aberto.

Outro ponto a destacar é que todos os mecanismos e processamentos adicionais relacionados ao WSDS (descritos na seção 5.5.3) são integrados aos servidores DEPSPACE através da instalação de interceptadores (ver seção 3.4.7), o que permite a mudança no comportamento dos servidores sem alterações em sua estrutura. Os dados adicionais, necessários na execução de uma requisição (ex.

dados relacionados com assinaturas), são enviados através de uma abstração (`WSMessageContext`) que acompanha as mensagens, como pode ser observado na figura 5.4 que representa a interface dos *gateways*. Ainda nesta interface, a abstração `WSResponse` merece destaque por representar o conjunto de respostas a serem enviadas ao cliente. A forma de utilização deste sistema é discutida no apêndice A.

```
public interface WSDepSpace extends java.rmi.Remote{
  WSResponse createSpace(Properties props, WSMessageContext messageContext)
                          throws WSDepSpaceException,java.rmi.RemoteException;
  WSResponse deleteSpace(String name, WSMessageContext messageContext) throws...
  WSResponse cas(WSDepTuple template, WSDepTuple tuple,WSMessageContext messageContext)throws...
  WSResponse clean(WSDepTuple proof, WSMessageContext messageContext) throws...
  WSResponse in(WSDepTuple template, WSMessageContext messageContext) throws...
  WSResponse inp(WSDepTuple template, WSMessageContext messageContext) throws...
  WSResponse out(WSDepTuple tuple, WSMessageContext messageContext) throws...
  WSResponse rd(WSDepTuple template, WSMessageContext messageContext) throws...
  WSResponse rdp(WSDepTuple template, WSMessageContext messageContext) throws...
}
```

Figura 5.4: Interface dos *gateways*.

5.7 Aplicações do WSDS

Esta seção apresenta alguns exemplos de uso do WSDS, com o objetivo de não apenas mostrar que o WSDS pode ser usado para resolver problemas tão dispares de forma tão boa quanto sistemas desenvolvidos exclusivamente para solucionar tais problemas, mas também mostrar que o mesmo é genérico suficiente para ser utilizado em um grande número de aplicações.

5.7.1 Licitações Seguras

Um tipo de aplicação que pode ser facilmente implementada usando o WSDS é um sistema de gerenciamento de licitações. Utilizando o WSDS, um licitador interessado em determinado serviço escreve uma tupla no espaço descrevendo as características do serviço a ser contratado (ou produto a ser comprado). A partir daí, os prestadores de serviço interessados (que podem ter sido previamente cadastrados no WSDS) também inserem uma tupla no WSDS descrevendo sua proposta para prestação do serviço. Por fim, depois da data limite para se efetuar as propostas, o cliente solicitante lê todas as tuplas com propostas relacionadas ao seu serviço e faz uma de três coisas: *(i.)* escolhe o fornecedor com a melhor proposta, pondo fim a licitação; *(ii.)* não escolhe nenhuma proposta pois decide que todas são inadequadas no que diz respeito à qualidade e ao preço estipulado na descrição da licitação; ou *(iii.)* faz um resumo das propostas obtidas e as publica no espaço, dando início a uma outra rodada de licitações (em uma espécie de leilão, onde os prestadores poderiam fazer novas propostas melhorando seu preço e ou ofertando melhorias no serviço). A figura 5.5 ilustra essa aplicação e os diversos tipos de tuplas que podem aparecer no WSDS.

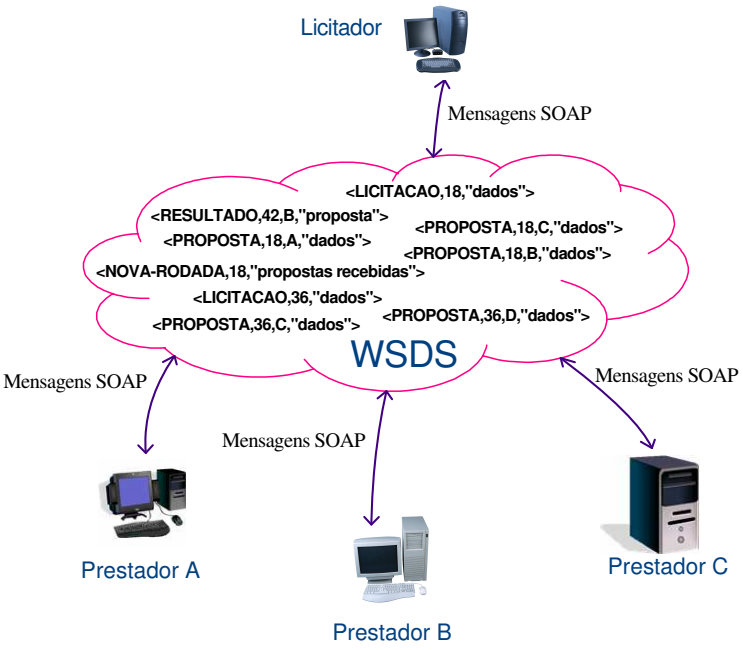


Figura 5.5: Licitações seguras.

Neste sistema existem vários requisitos de segurança: *(i.)* um prestador de serviço não deve ter acesso a proposta de outro prestador; *(ii.)* um prestador não pode realizar mais de uma proposta; *(iii.)* se um cadastramento prévio for exigido, um fornecedor não cadastrado não deve ser capaz de fazer propostas; entre outros específicos de cada licitação. Usando as capacidades do WSDS, em particular seus mecanismos de controle de acesso (ver seção 3.3.2), é possível garantir que todos esses requisitos sejam respeitados: o requisito *(i.)* pode ser implementado através da inclusão de tuplas contendo propostas que requerem credenciais de licitador para leitura e os requisitos *(ii.)* e *(iii.)* podem ser implementados através de políticas de granularidade fina que estipulam regras do tipo *"se houver uma proposta do prestador A para licitação X no espaço, não é permitida a inclusão de uma nova proposta de A para X"* e *"o prestador A só pode inserir uma proposta se houver uma tupla do tipo* $\langle PRESTADOR, A\rangle$ *no espaço"*[4], respectivamente[5].

Outros exemplos onde o espaço de tuplas é usado como mecanismo de comunicação desacoplada: o sistema I3 [72] oferece *multicast*, *anycast* e suporte a comunicação móvel na Internet através de uma abstração bastante semelhante a um espaço de tuplas; o padrão *master-worker* (onde um processo distribui tarefas a outros e depois agrupa os resultados), muito usado para distribuição de tarefas no processamento em *clusters*, pode facilmente ser implementado usando um espaço de tuplas [25] e, uma vez que este espaço esteja disponível na Internet, este mesmo modelo de programação pode ser usado na distribuição de tarefas em um *grid* computacional [37]. A alta disponibilidade e as políticas suportadas pelo WSDS podem garantir que estes serviços de comunicação e distribuição mantenham suas propriedades mesmo na presença de faltas e intrusões. Além disso, a característica de desacoplamento da coordenação por espaço de tuplas permite que as partes comunicantes interajam mesmo sem conhecerem o endereço uma das outras e sem estarem ativas ao mesmo tempo (ex. devido

[4]Esta regra deve ser complementada com outra que diz que esse tipo de tupla só pode ser inserida por um administrador.
[5]Lembrar que o WSDS também suporta confidencialidade de tuplas (ver seção 5.5.3.4).

a desconexões temporárias).

5.7.2 Compartilhamento de Dados entre Serviços Cooperantes

Um requisito recorrente em aplicações que envolvem a invocação de diversos serviços *web* para execução de uma determinada tarefa é o compartilhamento de informações entre estes serviços. Esse tipo de aplicação faz uso de uma *base de dados compartilhada* (cujos diversos serviços têm acesso) ou requer que todas as *informações de sessão* da tarefa (pertinentes aos diversos serviços) sejam anexadas a todas as mensagens trocadas durante a execução da tarefa.

Quando consideramos que os serviços *web* não estão instalados no mesmo domínio administrativo, a existência de uma base de dados aberta para acesso direto a partir da Internet pode incorrer em sérios problemas de segurança e torna os serviços cooperantes dependentes de um ponto único de falha. Já a passagem de informações de sessão em todas as mensagens pode exigir que um volume não desprezível de informações tenha que ser enviado de um serviço para outro, o que pode prejudicar muito o desempenho do sistema. Se um serviço como o WSDS estiver disponível, os serviços cooperantes podem armazenar todas as informações de sessão em um espaço de tuplas compartilhado. As características do WSDS garantem *(i.)* a confiabilidade deste repositório (replicação tolerante a faltas bizantinas), *(ii.)* controle de acesso ao espaço e as tuplas armazenadas, *(iii.)* confidencialidade dos dados armazenados e *(iv.)* acesso universal (através dos *gateways*, que são serviços *web*).

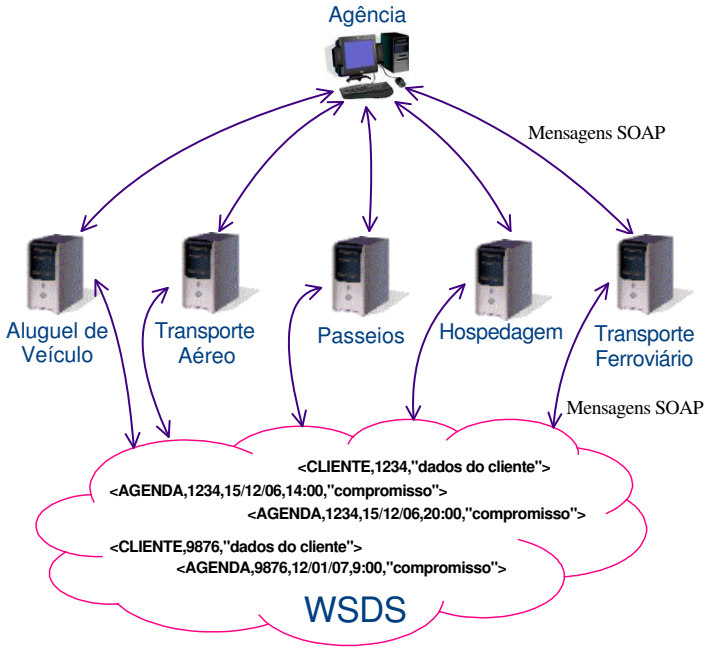


Figura 5.6: Agendamento de viagens.

Como exemplo deste tipo de sistema, considere um conjunto de serviços *web* usados por uma agência de turismo para alocar os recursos requeridos em uma viagem de férias (ex. transporte aéreo, hospedagem, aluguel de carro, agendamento de passeios turísticos, dentre outros). Para que todo esse processamento possa ser feito de forma automatizada, as informações sobre o viajante, suas

preferências e sua agenda devem estar disponíveis a todos estes serviços. Além disso, os serviços podem alterar esses dados (ex. agenda) conforme executam sua parte do planejamento e agendamento da viagem. A figura 5.6 ilustra esse sistema.

Este exemplo é extremante interessante no sentido de necessitar também das capacidades de sincronização do espaço: as tuplas da agenda podem ser usadas de tal forma que os problemas decorrentes de diferentes serviços agendando compromissos concorrentemente, para um mesmo horário, possam ser resolvidos através de operações com poder de sincronização providas pelo espaço, como *cas* e *inp*.

A política de segurança poderia definir que tipo de informação (tupla) pode ser acessada por cada tipo de serviço e que apenas a agência de viagem pode remover um compromisso já agendado ou cancelar o aluguel de um recurso (hospedagem, carro) já alugado. Além disso, as tuplas com os dados do cliente devem ser mantidas tão confidenciais quanto possível, visando manter sua privacidade.

Neste exemplo, a agência poderia contratar cada serviço para o viajante através do mecanismo de licitações seguras apresentado na seção anterior. Os dois modelos de programação podem ser integrados no mesmo espaço de tuplas ou usando diferentes espaços lógicos para cada licitação e para a agenda do cliente.

Outros exemplos de serviços desse tipo já explorados são: integração de sistemas de detecção de intrusões [81] – onde resumos de comportamentos observados por diferentes sistemas são compartilhados no espaço de tuplas para permitir a correlação de atividades suspeitas; e o suporte a empresas virtuais [21, 66] – onde um espaço de dados compartilhado é usado entre os diversos parceiros que formam a empresa virtual, visando compartilhar informações.

5.8 Relações com Especificações para Serviços *Web*

Existe uma série de especificações desenvolvidas (ou em desenvolvimento) pela comunidade para prover integração de serviços *web*. Nesta seção discutimos a relação do WSDS com algumas destas especificações.

WS-Coordination: Este padrão define um modelo genérico através do qual diversos serviços *web* podem se coordenar para a realização de uma determinada tarefa [50]. O ponto central deste modelo é o contexto de coordenação, uma abstração que contém todas as informações sobre quem são os serviços coordenantes e quaisquer outras informações sobre a coordenação. O WS-COORDINATION oferece interfaces que contemplam operações para criação de contextos e registro de participantes em contextos já existentes. A abstração de espaço de tuplas, provida pelo WSDS, pode ser implementada como outra instanciação do WS-COORDINATION onde o contexto de coordenação é o espaço de tuplas lógico e os serviços registrados no contexto podem ser entendidos como os clientes que podem interagir com o espaço.

WS-Orchestration: A orquestração de serviços *web* compreende o uso de diversos serviços coordenados por um "maestro", chamado motor de orquestração. A idéia básica é definir quais serviços *web* serão invocados e em que ordem, usando uma linguagem de coordenação, como a WSBPEL (*Web Services Business Process Execution Language*) [4]. A agência de viagens descrita na seção anterior é um exemplo de tarefa que poderia ser implementada com orquestração. Até o momento não existe um padrão de repositório de dados compartilhados para orquestração de serviços *web*, e toda informação requerida pelos serviços deve ser mantida pelo motor de orquestração e enviada a cada serviço como parâmetro da operação requisitada, podendo ser bastante ineficiente se o volume de dados for grande. Um repositório de dados seguro e confiável pode ser útil nesse sentido, e sua integração com a linguagem WSBPEL pode ser facilmente feita, bastando que os serviços sendo orquestrados usem o WSDS.

WS-Choreography: A coreografia de serviços *web* permite a definição formal de um conjunto de interações entre serviços *web*, a verificação de sua corretude (ex. provando a ausência de *deadlocks*) e a geração do código responsável pelas interações [22]. A coreografia difere da orquestração em pelo menos dois aspectos: é distribuída (enquanto a orquestração tem um coordenador – o motor de orquestração); e as linguagens de coreografia (como o WSCL – *Web Services Choreography Language*) são usadas para *especificação* de interações requeridas durante a realização de tarefas, não sendo *executáveis* (diferindo de linguagens para orquestração, como o WSBPEL) [65]. A introdução da coreografia de serviços *web* visa tornar essas interações menos suscetíveis a erros, através de uma especificação mais rigorosa das relações entre os serviços. Porém, esta tecnologia não oferece nenhum tipo de garantia que as interações vão ocorrer como especificado em tempo de execução. O uso de um mediador como o WSDS preenche esta lacuna. As interações poderiam ser feitas através de tuplas inseridas e lidas do espaço de tuplas. Políticas podem ser geradas, a partir da coreografia especificada, para garantir que o sistema funciona da forma prevista mesmo na presença de falhas e intrusões. Além disso, o uso do WSDS permite a implementação de interações multiparte e a gravação (e posterior auditoria) de todas as interações executadas pelo serviços.

5.9 Trabalhos Relacionados

Atualmente existe um grande esforço da comunidade de sistemas distribuídos no desenvolvimento de mecanismos padronizados que permitam a cooperação e a integração de serviços *web*. Iniciativas como WS-COORDINATION [44], WS-ORCHESTRATION [4] e WS-CHOREOGRAPHY [22] vão exatamente nessa direção (ver seção 5.8). Estas iniciativas consideram basicamente a integração dos serviços *web* através de trocas de mensagens, e não definem nenhum tipo de abstração que suporte o armazenamento de dados relacionados a uma tarefa cooperativa sendo executada pelos serviços. Esse tipo de abstração é crucial em aplicações onde o volume de dados compartilhado é grande ou onde desacoplamento é necessário. O WSDS visa fornecer essa abstração sem no entanto esquecer de aspectos de segurança de funcionamento. Mais especificamente, o serviço de coordenação por ele suportado é seguro e tolerante a falhas e intrusões.

A integração do modelo de coordenação por espaço de tuplas no ambiente dos serviços *web* tem se tornado um tema de pesquisa ativo nos últimos anos (ex. [11, 21, 56, 57]). Esses trabalhos enaltecem as facilidades que um repositório de dados compartilhado, com interfaces bem definidas e alguma capacidade de sincronização, pode proporcionar na colaboração de serviços *web* [57] e na execução de *workflows* [11, 21]. Um aspecto importante que escapa a quase todos esses trabalhos é a segurança de funcionamento [9] requerida nesse serviço. A segurança de funcionamento deve ser provida em dois níveis: *(1)* mecanismos de segurança para controle de acesso ao espaço de tuplas e *(2)* disponibilidade do serviço de coordenação em si. A maioria dos trabalhos na área não implementa nenhum desses níveis, focando-se na integração do modelo de coordenação por espaço de tuplas com os padrões para serviços *web* [11, 21, 57]. Uma notável excessão é o WS-SECSPACES [56], que se preocupa com o nível *(1)*, provendo um modelo de controle de acesso em nível de espaço e tuplas, permitindo a especificação de quem pode inserir, ler e remover determinada(s) tupla(s).

O WSDS, apresentado neste trabalho, suporta um modelo de controle de acesso semelhante ao provido pelo WS-SECSPACES além de suportar a definição de políticas de segurança de granularidade fina, que dão poder de coordenação ao espaço de tuplas mesmo na presença de processos maliciosos [16]. Além disso, o WSDS implementa mecanismos para suprir o nível *(2)* de segurança de funcionamento, através de uma arquitetura simples e eficiente que faz uso de *gateways* para acesso a um serviço de coordenação confiável – DEPSPACE – implementado através de replicação Máquina de Estados tolerante a faltas bizantinas [26]. O sistema resultante tolera faltas acidentais e maliciosas em todos os componentes do sistema, provendo serviço correto enquanto houver no mínimo um *gateway* e mais de um terço das réplicas do espaço de tuplas corretos. Todas essas características são providas sem ferir a adesão aos padrões para serviços *web*.

5.10 Considerações Finais

Este capítulo apresentou o WSDS, um serviço de coordenação e/ou cooperação para serviços *web* que provê elevadas garantias em termos de segurança de funcionamento, como controle de acesso através de políticas de granularidade fina e tolerância a faltas e intrusões. A arquitetura do sistema se baseia no uso de um espaço de tuplas com segurança de funcionamento e em serviços *web* sem estado (*gateways*) para acesso a este espaço. Diversos mecanismos foram empregados para garantir a segurança do sistema mesmo na presença de *gateways* maliciosos.

O sistema foi implementado usando ferramentas de código aberto e uma análise preliminar de seu desempenho, apresentada no capítulo 6, mostra que grande parte dos custos em termos da latência do serviço vêm do processamento das assinaturas digitais requeridas, o que deve ser diluído quando da instalação do sistema em uma rede de larga escala como a Internet.

Capítulo 6

Experimentos, Resultados e Análises

Este capítulo apresenta uma série de experimentos realizados com os protocolos e mecanismos que compõem os sistemas apresentados neste trabalho (DEPSPACE e WS-DEPENDABLESPACE). Mais precisamente, a latência destes sistemas é investigada. Para o sistema DEPSPACE, também é realizada uma análise do *throughput* fornecido pelo mesmo. Além disso, o protocolo de difusão atômica utilizado por estes sistemas é examinado em diversos cenários, incluindo execuções com faltas e com concorrência.

6.1 Ambiente de Execução

O ambiente de testes foi composto por cinco máquinas com a mesma configuração de *hadware* (AMD Athlon XP 1.9GHz, 512MB de RAM, placa *ethernet* de 100MB/s) conectadas por um *switch* de 1GB/s, formando uma rede local. Estas máquinas foram configuradas com o mesmo ambiente de *software* (S.O. GNU/Linux *Kernel* 2.6.12, máquina virtual Java da SUN versão 1.5.0_06 (com o compilador *Just-In-Time* do Java sempre ativado). A topologia da rede utilizada nos experimentos é apresentada pela figura 6.1.

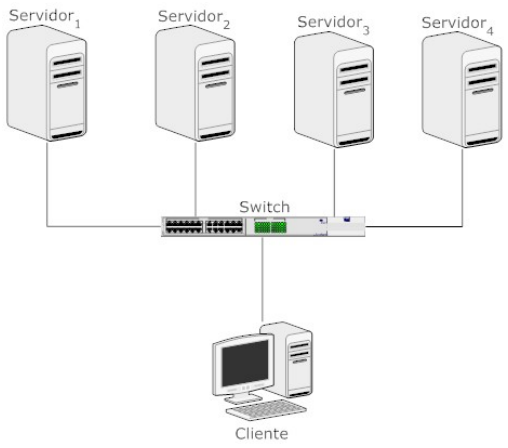


Figura 6.1: Topologia da rede.

6.2 Analisando o Desempenho do Protocolo de Difusão Atômica

Esta seção apresenta algumas análises sobre o desempenho do protocolo de difusão atômica apresentado no capítulo 4, em especial os efeitos da concorrência (sistema submetido a alta carga de mensagens) e das faltas no sistema. Nestes experimentos, utilizou-se um cliente que envia uma mensagem aos servidores e aguarda por $f+1$ *acks* referentes à entrega desta mensagem pelos servidores (lembrando que o número de servidores é $n \geq 3f+1$). Sendo assim, cada servidor envia o *ack* para o cliente apenas quando a mensagem é entregue por este servidor, i.e, depois da mensagem ser ordenada. Desta forma, um cliente inicia uma operação enviando a mensagem para os servidores e termina esta operação após o recebimento de $f+1$ *acks* relacionados com a entrega desta mensagem (o termo *operação* é utilizado nesta seção para referir-se a este processamento). O sistema foi configurado com 4 servidores (réplicas) e utilizou-se uma mensagem de 64 *bytes*.

Todos os valores reportados nesta seção compreendem o tempo médio necessário para a execução de uma operação por este cliente (latência), computado a partir de 1000 repetições da operação e excluindo-se os 5% dos valores com maior desvio. Os valores em tempo referidos no texto a seguir são aproximados.

Figura 6.2: Análise do protocolo de difusão atômica em vários cenários de faltas.

A figura 6.2 apresenta a latência média para execução de operações no sistema sob diversas condições de faltas e sem concorrência. Neste figura, podemos perceber que a diferença de tempo entre a execução rápida do PAXOS (em *rounds* muito favoráveis) e a execução normal (em *rounds* favoráveis) é praticamente inexistente. Isto se deve à diminuta latência para envio de mensagens via o canal autenticado em nosso modelo[1], o que torna o tempo requerido para a execução de um passo de comunicação extra praticamente irrisório.

[1] A implementação de canais autenticados é bastante leve, uma vez que a geração e verificação de HMACs para autenticação de mensagens leva menos de 1 ms.

Como já era esperado, o cenário em que o sistema apresentou a pior latência é quando o proponente do primeiro *round* do PAXOS é o servidor faltoso. Neste caso, a mensagem somente é ordenada no segundo *round* e a fase de troca de líder envolve computações de assinaturas assimétricas (RSA) (ver seção 4.2.3), que têm um processamento custoso, o que acarreta uma latência total de 73 ms.

A figura 6.3 mostra a latência das operações em execuções sem faltas e com concorrência de 0-10 clientes executando operações no sistema, i.e., enviando mensagens para serem ordenadas. Os valores reportados neste gráfico referem-se ao tempo necessário para um cliente executar uma operação (ter sua mensagem ordenada). O objetivo destes experimentos é verificar a escalabilidade do sistema quando exposto a uma alta carga de mensagens a serem ordenadas.

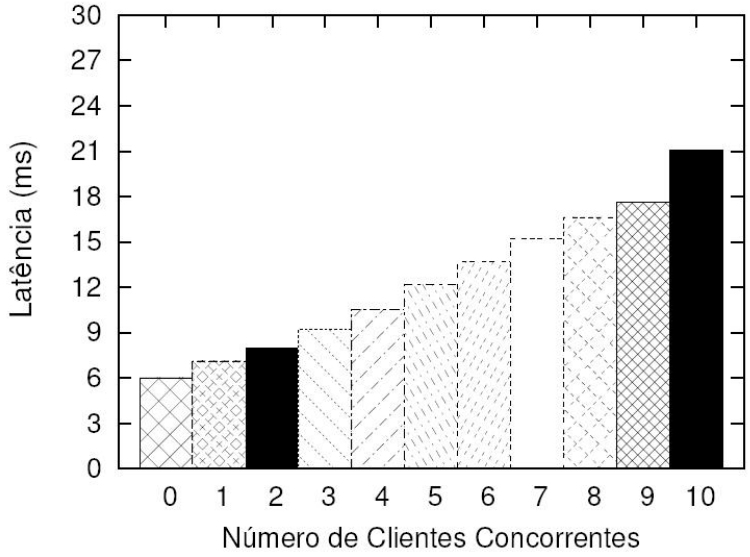


Figura 6.3: Análise do protocolo de difusão atômica em vários cenários de concorrência.

Analisando a figura 6.3 é possível perceber que o sistema apresentou uma escalabilidade aceitável, pois a latência percebida pelo cliente apresentou um crescimento bastante "suave" com o aumento da concorrência no sistema. Como pode ser observado nesta figura, a latência de processamento de uma operação variou entre 6 ms (sem concorrência no sistema) e 21 ms (com 10 clientes concorrentes no sistema). Este comportamento se deve ao fato das mensagens serem ordenadas em lotes, deste modo a execução de uma instância do algoritmo PAXOS ordena um conjunto de mensagens (e não apenas uma – ver seção 4.3.7).

Os resultados destes experimentos mostram que o protocolo de difusão atômica apresentado no capítulo 4 é bastante eficiente em casos sem falta, isto pode ser explicado por dois fatos: *(1)* o não uso de criptografia assimétrica nestes casos; e *(2)* a baixíssima latência e a previsibilidade das comunicações em redes locais, que tornam praticamente impossível a troca de *rounds* em casos sem falta.

6.3 Analisando o Desempenho do DEPSPACE

Esta seção apresenta uma avaliação experimental do DEPSPACE (ver capítulo 3), onde duas variáveis foram consideradas: o número de servidores (réplicas) e o tamanho das tuplas. Os testes foram realizados com $n = 4, 7, 10$ servidores[2] (suportando 1, 2, e 3 servidores faltosos, respectivamente) e tuplas de 64 *bytes*. Com relação a variação do tamanho das tuplas, foram realizados experimentos com tuplas de tamanho igual a 64, 256, e 1024 *bytes* e o sistema foi configurado com 4 servidores.

Todas as tuplas utilizadas nos testes foram configuradas com 4 campos. Nos experimentos com confidencialidade, estes campos foram do tipo comparável, o que faz com que o *hash* dos mesmos também seja enviado com a tupla, além de todas as informações relacionadas com o mecanismo de confidencialidade (ver seção 3.3.3).

Foram analisados a latência e o *throughput* do sistema, que foi configurado de duas formas: com confidencialidade e com a camada de confidencialidade desativada.

6.3.1 Analisando a Latência

O primeiro experimento analisou o tempo (percebido pelo cliente) necessário para realizar cada uma das operações não bloqueantes: *out*, *rdp*, *inp* e *cas*. O cliente foi executado em uma das máquinas e os servidores nas outras quatro. Os valores reportados correspondem a média de 1000 invocações ao sistema, excluindo-se 5% dos valores com maior desvio.

Os resultados (figuras 6.4 e 6.5) mostram que as operações de *out*, *inp* e *cas* apresentaram valores de latência muito próximos quando a camada de confidencialidade é desativada (linhas sólidas). Esta é a latência imposta pelo protocolo de difusão atômica, que varia de aproximadamente 6 ms (4 servidores) até 12 ms (10 servidores). A operação de *rdp*, por outro lado, é muito mais eficiente (aproximadamente 2 ms) devido a otimização apresentada na seção 3.4.1, que evita o uso do protocolo de difusão atômica (figuras 6.4(b) e 6.5(b)).

As linhas pontilhadas nos gráficos mostram a latência dos protocolos quando a camada de confidencialidade é ativada. As operações de *cas* (figuras 6.4(d) e 6.5(d)) têm duas linhas pontilhadas, uma representando os casos onde a tupla é inserida no espaço e a outra os casos onde alguma tupla é lida do espaço.

O custo adicional, causado pela adição de confidencialidade, está relacionado principalmente com o processamento do lado do cliente, onde os *shares* precisam ser gerados, verificados e combinados (dependendo da operação). O único processamento adicional no lado do servidor é a extração do *share* destinado a tal servidor. Este é um fato importante, pois indica a capacidade do sistema escalar em relação a este fator. Outro fator que prejudica o desempenho do sistema quando o mecanismo de confidencialidade é ativado, é a quantidade de dados extra (relacionados com este mecanismo) que

[2]Devido as limitações de recursos, nos experimentos com 7 e 10 servidores, foi necessário executar mais de um servidor por máquina.

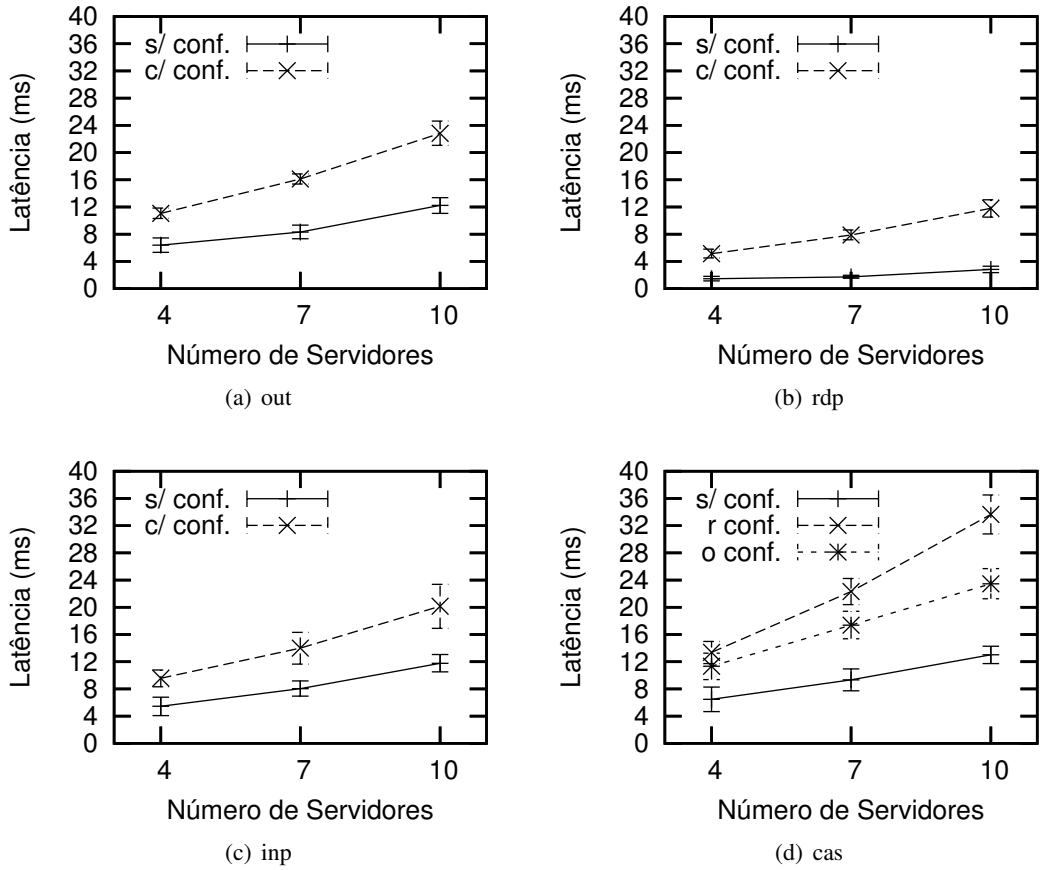
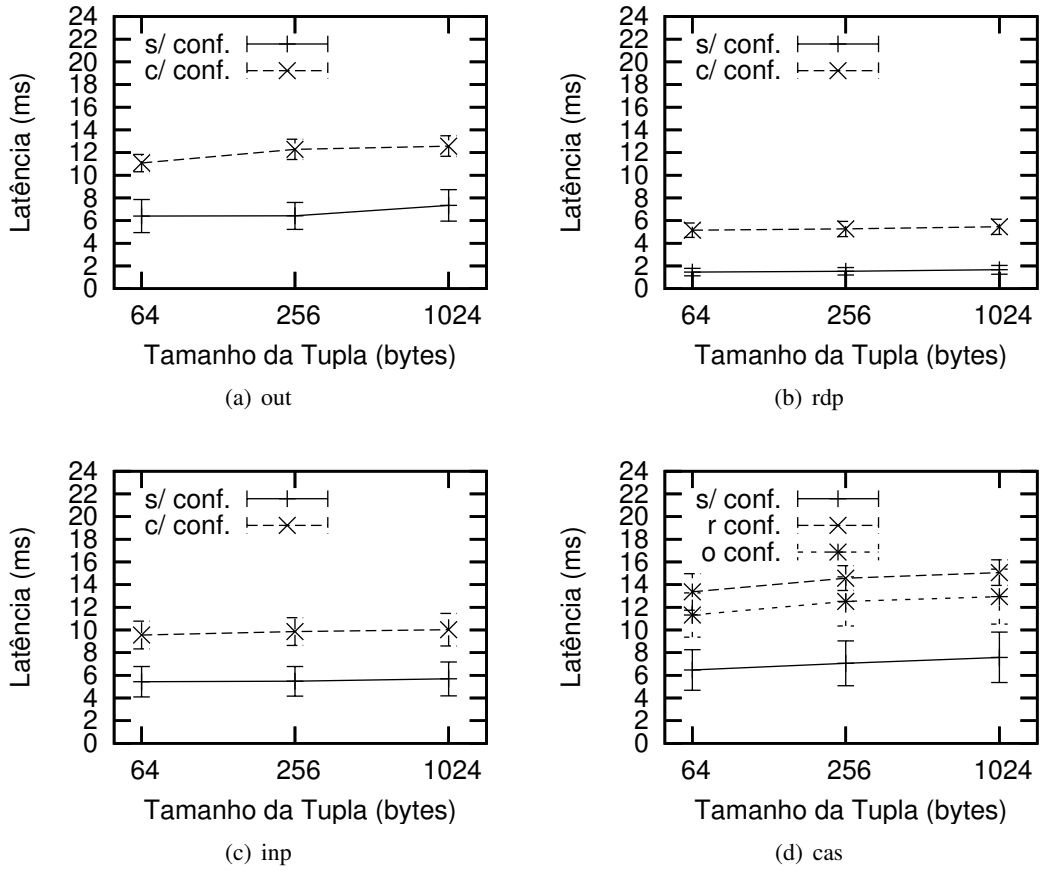


Figura 6.4: Latência das operações do DEPSPACE variando o número de servidores.

precisa ser enviada aos servidores. O esquema de confidencialidade utilizado pelo DEPSPACE (ver seção 3.3.3) foi desenvolvido em [12, 15], onde pode ser encontrada uma análise mais detalhada sobre os custos deste mecanismo.

Nos experimentos onde o tamanho das tuplas sofreram variações (figura 6.5), é facilmente verificado que o tamanho da tupla praticamente não afeta a latência apresentada pelo sistema. Isto deve-se a dois fatores de implementação: *(i.)* uso dos *hashs* das mensagens pelo protocolo de difusão atômica (ver seção 4.3.7); e *(ii.)* o segredo compartilhado entre os servidores não é a tupla, mas a chave simétrica usada para cifrar a tupla (ver seção 3.3.3). O item *(i.)* implica que não é a mensagem inteira que será ordenada, mas o resumo da mesma (*hash*), conseqüentemente o tamanho da mensagem não afeta significantemente o desempenho do protocolo de difusão atômica. Já o item *(ii.)* faz com que as operações criptográficas, utilizadas pelo esquema de confidencialidade, não tenham o desempenho influenciado pelo tamanho da tupla.



(a) out (b) rdp

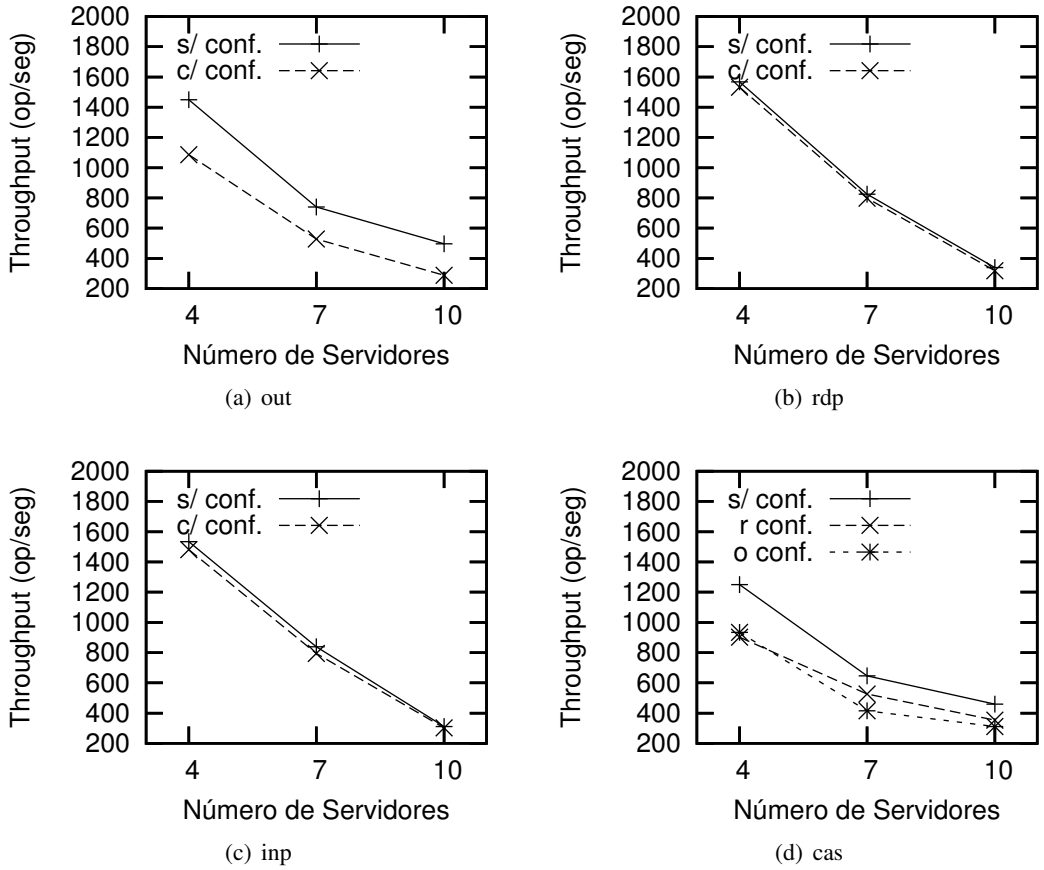
(c) inp (d) cas

Figura 6.5: Latência das operações do DEPSPACE variando o tamanho das tuplas.

6.3.2 Analisando o *Throughput*

O segundo experimento analisou o *throughput* apresentado pelo DEPSPACE para os dois fatores analisados (número de servidores e tamanho das tuplas). Neste experimento foi necessário um cliente modificado, que realizasse o pré-processamento de C requisições para a operação de interesse (executando o processamento do lado do cliente) e enviasse estas requisições para os servidores (uma por uma). Então, foi determinado o tempo T necessário para o servidor líder executar todas estas requisições (tempo entre o recebimento da primeira requisição e o envio da última resposta). O *throughput* foi determinado a partir destes valores como sendo C/T. Os resultados são apresentados nas figuras 6.6 e 6.7, onde pelo menos três observações merecem destaque.

A figura 6.6 mostra um decréscimo dramático no *throughput* do sistema (para todas as operações), quando o número de servidores aumenta. De fato, os resultados mostram que para aumentar apenas uma unidade no número de faltas suportadas pelo sistema (i.e., de $n = 4$ para $n = 7$) o *throughput* cai pela metade (facilmente observado na figura 6.6(b)). Isto acontece por duas razões: *(1)* é sabido que o algoritmo PAXOS BIZANTINO, usado pelo protocolo de difusão atômica (ver capítulo 4), não apresenta uma boa escalabilidade em relação ao número de servidores (n) [1]: este algoritmo apresenta



(a) out (b) rdp

(c) inp (d) cas

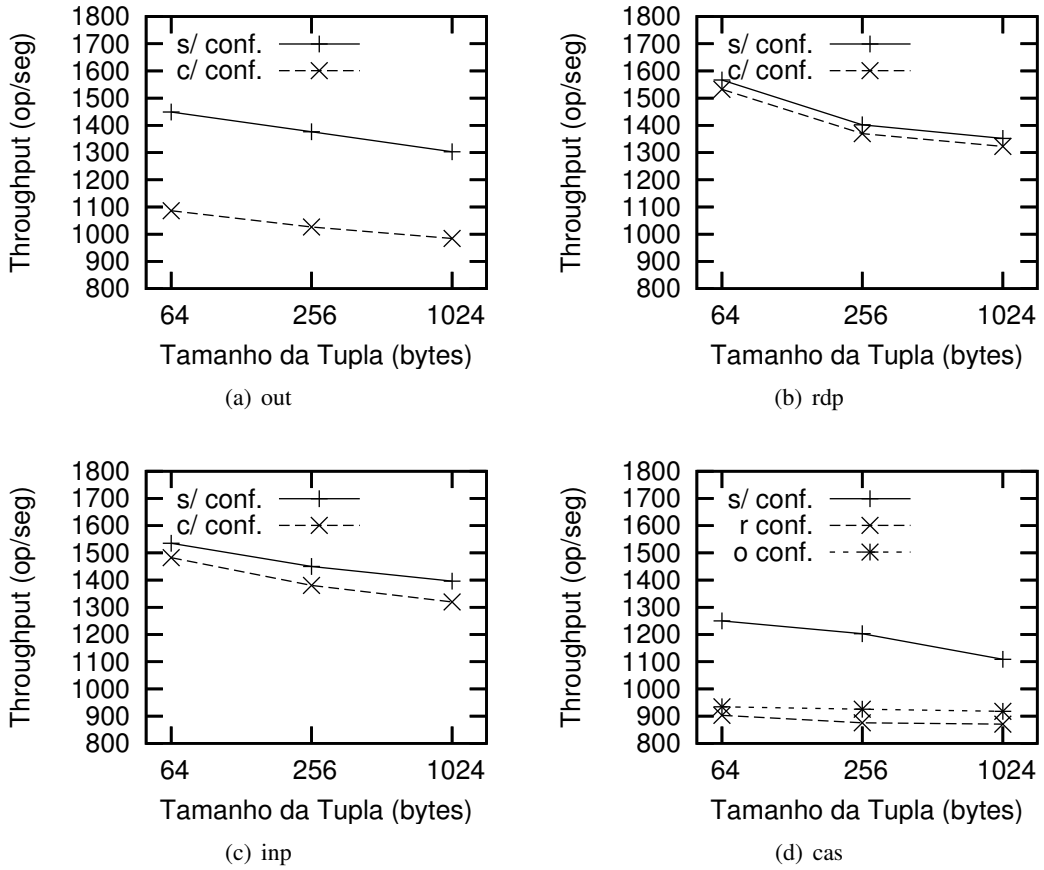
Figura 6.6: *Throughput* das operações do DEPSPACE variando o número de servidores.

complexidade de mensagens quadrática ($O(n^2)$) e a comunicação entre os processos, no DEPSPACE, é baseada no protocolo TCP/IP[3]; e *(2)* nas configurações com $n = 7$ e $n = 10$ foi necessário executar mais de um servidor por máquina, o que significa que algumas máquinas executaram duas ($n = 7$) ou três ($n = 10$) vezes o processamento exigido para um servidor. Mais especificamente, foi observado que o processamento de I/O na rede foi fator determinante nestes resultados.

A figura 6.7 mostra que o sistema fornece um *throughput* alto quando é configurado com poucos servidores[4]. Além disso, com o aumento do tamanho das tuplas o *throughput* apresentou uma queda suave, i.e., aumentado o tamanho da tupla em 16 vezes (64 para 1024 *bytes*) o *throughput* diminui cerca de 10%. O bom desempenho (com relação ao *throughput*) apresentado pelo sistema com confidencialidade, deve-se ao pouco processamento exigido nos servidores e a ordenação de mensagens em lote (ver seção 4.3.7).

[3]O sistema BFT utiliza o protocolo UDP/*multicast* IP [26].

[4]O *throughput* apresentado pelo DEPSPACE (com confidencialidade) atinge 50% a 80% do *throughput* reportado pelo sistema GIGASPACES (em uma configuração de *hardware* mais modesta), que é escalável mas não é replicado e nem tolerante a intrusões [43].



(a) out (b) rdp

(c) inp (d) cas

Figura 6.7: *Throughput* das operações do DEPSPACE variando o tamanho das tuplas.

Outro ponto que merece ser destacado é o decréscimo do *throughput* nas operações que inserem (ou podem inserir) tuplas no espaço quando a mecanismo de confidencialidade é ativado (figuras 6.7(a) e 6.7(d)). Isto acontece pelo fato do tamanho das tuplas sofrer um grande aumento com a inclusão das informações relacionadas com a propriedade de confidencialidade, i.e., uma tupla de 256 *bytes* resulta em uma requisição, para a operação *out*, de 654 *bytes* (sem confidencialidade) ou 1990 *bytes* (com confidencialidade). Esta diferença de tamanho deve-se aos dados adicionais relacionados com o esquema de confidencialidade e ao mecanismo de serialização do Java, que faz com que a serialização da classe `BigInteger` (amplamente usada pelo esquema de confidencialidade) apresente um tamanho maior do que o esperado.

Mesmo com as deficiências acima descritas, os experimentos mostraram que no pior caso observado (operação de *out* com 10 servidores), o DEPSPACE pode executar aproximadamente 300 op/seg com latência de 24 ms, o que é aceitável para a maioria das aplicações para as quais o DEPSPACE foi desenvolvido.

6.4 Analisando o Desempenho do WS-DEPENDABLESPACE

Nesta seção é apresentada uma análise acerca do desempenho do WS-DEPENDABLESPACE (ver capítulo 5), em particular da latência envolvida na execução de uma operação deste serviço, a qual é determinada pela equação:

$$L_{wsds} = L^{c}_{sign} + L^{c \to g}_{comm} + L^{g \to s}_{tom} + L^{s}_{ver} + L^{s}_{op} + L^{s}_{sign} + L^{s \to g}_{comm} + L^{g \to c}_{comm} + (f+1)L^{c}_{ver} \quad (6.1)$$

As latências envolvidas nesta equação são: L^{c}_{sign} - assinatura da requisição; $L^{c \to g}_{comm}$ - envio da requisição ao *gateway*; $L^{g \to s}_{tom}$ - envio da requisição aos servidores e sua ordenação; L^{s}_{ver} - verificação da integridade da requisição; L^{s}_{op} - execução da operação; L^{s}_{sign} - assinatura da resposta; $L^{s \to g}_{comm}$ - envio da resposta ao *gateway*; $L^{g \to c}_{comm}$ - envio das respostas ao cliente, note que $L^{c \leftrightarrow g}_{comm} = L^{c \to g}_{comm} + L^{g \to c}_{comm}$ representa a comunicação entre o cliente e o *gateway*; L^{c}_{ver} - verificação da integridade da resposta.

Assumindo que as operações de assinatura e verificação têm latências semelhantes no cliente e nos servidores e que o custo do processamento local das operações no espaço de tuplas é desprezível ($L^{s}_{op} = 0$), chegamos a latência esperada para os serviços oferecidos pelo WSDS:

$$L_{wsds} = 2L_{sign} + L^{c \leftrightarrow g}_{comm} + L^{g \to s}_{tom} + L^{s \to g}_{comm} + (f+2)L_{ver} \quad (6.2)$$

Dentro destas mesmas premissas, a latência esperada para o DEPSPACE é $L_{ds} = L^{g \to s}_{tom} + L^{s \to g}_{comm}$. Conseqüentemente, o custo adicional do WS-DEPENDABLESPACE em relação ao DEPSPACE consiste nas assinaturas da requisição e da resposta, na comunicação extra para acesso ao *gateway* e na verificação da autenticidade da requisição e das $f+1$ respostas.

Visando analisar se essa latência é observada na prática, alguns experimentos foram realizados com 4 servidores DEPSPACE (um em cada máquina), um *gateway* (executado em uma máquina junto com um servidor DEPSPACE) e um cliente (executado separadamente em uma máquina). Esta seção apresenta os resultados destes experimentos, que não contemplam a propriedade de confidencialidade, visto que o objetivo é apenas verificar os acréscimos nos custos das operações em relação ao DEPSPACE (conforme as fórmulas anteriormente apresentadas). Todos os valores reportados aqui compreendem o tempo médio necessário para a execução de uma operação por um cliente do sistema, recolhido a partir de 1000 execuções da operação e excluindo-se os 5% dos valores com maior desvio.

A figura 6.8 apresenta a latência média para a execução das quatro operações[5], variando-se o tamanho das tuplas. Nesta figura, podemos perceber que a latência apresentou um crescimento suave com o aumento do tamanho das tuplas, o que indica que o sistema é escalável em relação a este fator. Além disso, pode-se notar que os desvios apresentados ($2-6ms$) são apropriados, levando em consideração o tipo de serviço oferecido.

[5] O molde utilizado na operação *cas* sempre combinava com alguma tupla contida no espaço, representando os casos onde uma tupla é lida do espaço.

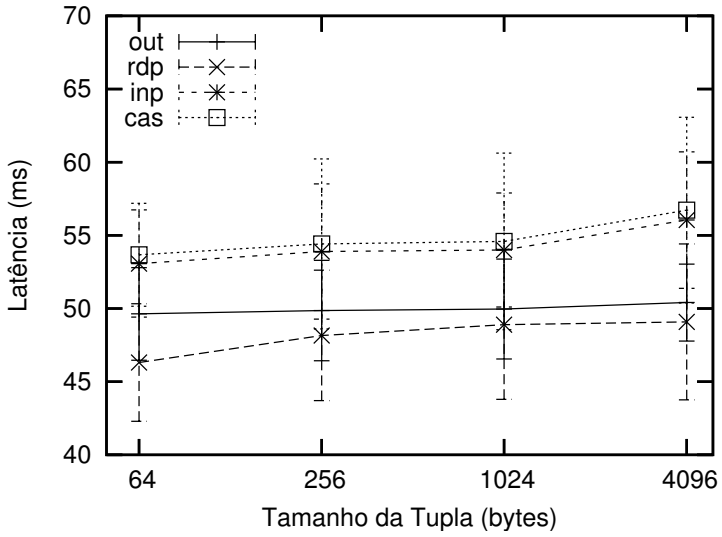


Figura 6.8: Latência do WS-DEPENDABLESPACE.

A tabela 6.1 apresenta os custos relacionados com cada fase do protocolo (para o processamento da operação OUT de uma tupla de 64 *bytes*), juntamente com sua percentagem correspondente na latência total observada pelo cliente, de acordo com a equação 6.2. Nesta tabela, podemos observar que as comunicações entre o cliente e o *gateway* (SOAP) é a fase que consome mais tempo, seguido pelas assinaturas (em redes de larga escala, a tendência é que esta fase tenha uma participação menor na latência).

Latência	Custo (ms)	$\%L_{wsds}$
$L_{comm}^{s \to g}$	0.63	$1,27$
$L_{comm}^{c \leftrightarrow g}$	13.53	$27,27$
L_{sign}	13.51	$54,45$
L_{ver}	0.85	$5,13$
$L_{tom}^{g \to s}$	5.90	$11,88$
L_{wsds}	49.63	100
L_{ds}	6.53	$13,15$

Tabela 6.1: Custos da latência.

6.5 Considerações Finais

Este capítulo apresentou uma série de experimentos realizados com os diversos mecanismos desenvolvidos e abordados no decorrer deste texto. Estes experimentos contemplaram vários cenários, incluindo execuções com faltas e execuções onde a propriedade de confidencialidade foi garantida (dentre outros).

O principal indicador de desempenho investigado nestes testes foi a latência do sistema (tempo necessário para um cliente executar uma operação no sistema). No entanto, alguns testes abordaram

outros indicadores como o *throughput* (fornecido pelo DEPSPACE). Além disso, foram apresentadas algumas análises sobre o comportamento do protocolo usado pela camada de replicação em cenários com faltas e com concorrência.

Apesar das limitações destes experimentos, através de algumas análises sobre os resultados obtidos é possível ter uma previsão do comportamento das diversas partes que compõem o sistema e do sistema como um todo. Conclusões mais precisas, obtidas a partir de experimentos mais significativos, poderiam ser formuladas com a utilização de um *testbed* melhor.

Capítulo 7

Conclusão

O presente capítulo conclui esta dissertação. Primeiramente, uma visão geral sobre o trabalho é apresentada. Após isso, os objetivos desta dissertação são relembrados e as contribuições da mesma são abordadas. Por fim, algumas perspectivas de trabalhos futuros são expostas.

7.1 Visão Geral do Trabalho

Este trabalho apresentou estudos sobre o desenvolvimento de uma infra-estrutura de coordenação e/ou cooperação de serviços *web*. Esta infra-estrutura possui segurança de funcionamento, i.e., mesmo que uma parte dos componentes que a compõem sofrer faltas bizantinas [54] (o tipo mais genérico de faltas), seus serviços continuam sendo oferecidos segundo suas especificações.

O trabalho pode ser dividido em três partes básicas. A primeira compreende a concretização do espaço de tuplas com segurança de funcionamento descrito em [12], chamada de DEPSPACE. Este espaço de tuplas foi construído em camadas, conforme projetado em [12], sendo que para a camada de replicação foi necessário o desenvolvimento de um protocolo de difusão atômica tolerante a faltas bizantinas, pois esta camada utiliza replicação Máquina de Estados [68].

A segunda parte deste trabalho está relacionada com o projeto e o desenvolvimento da infra-estrutura de coordenação e/ou cooperação de serviços *web* propriamente dita. Esta infra-estrutura tem como base o DEPSPACE, onde uma série de novos mecanismos foram introduzidos para manter as propriedades de segurança de funcionamento do sistema [9]. Esta infra-estrutura, chamada de WS-DEPENDABLESPACE, representa um objeto de memória compartilhada acessível na forma de um serviço *web*, que pode ser usado tanto para coordenar serviços *web* (de forma segura e confiável) quanto por serviços *web* cooperantes (como um repositório seguro e confiável de dados).

A última parte deste trabalho compreende uma série de experimentos realizados com os diversos mecanismos e protocolos abordados no decorrer desta dissertação. Os resultados destes experimentos são apresentados e analisados, destacando-se quais são os fatores que levam a estes resultados.

7.2 Revisão dos Objetivos e Contribuições desta Dissertação

Os objetivos desta dissertação, enunciados na seção 1.2, são relembrados nesta seção, onde é indicado que capítulo apresenta o estudo que visa cumprir cada objetivo. Além disso, as principais contribuições desta dissertação, que vão ao encontro do cumprimento destes objetivos, são abordadas.

1. **Desenvolvimento de um espaço de tuplas com segurança de funcionamento.**

   O capítulo 3 apresentou o DEPSPACE, que é a concretização do espaço de tuplas com segurança de funcionamento projetado em [12]. O DEPSPACE foi construído em camadas, sendo que em cada camada uma funcionalidade diferente é adicionada visando garantir as propriedades de segurança de funcionamento. Além disso, este sistema suporta a criação de espaços de tuplas lógicos, sendo que cada espaço de tuplas lógico pode ser configurado de acordo com as necessidades do usuário, i.e., pode-se determinar quais camadas serão ativadas juntamente com suas configurações. Neste sentido, este trabalho contribuiu com a concretização deste espaço de tuplas, onde uma série de questões relacionadas com a implementação do mesmo foram consideradas com o intuito de facilitar o uso e a configuração do sistema.

2. **Projeto e desenvolvimento de um protocolo de difusão atômica tolerante a faltas bizantinas.**

   O capítulo 4 discute o projeto e desenvolvimento deste protocolo, que é baseado no algoritmo de consenso PAXOS BIZANTINO [26, 83]. Para chegarmos a este protocolo de difusão atômica, algumas alterações no algoritmo PAXOS BIZANTINO foram necessárias, além da integração de novos mecanismos que visam garantir as propriedades relacionadas com o problema da difusão atômica (ver seção 2.2.2). Este mesmo capítulo apresenta uma valiosa discussão sobre o algoritmo do PAXOS BIZANTINO, necessária para entendermos o protocolo proposto, mas que é interessante por si só. A definição deste protocolo foi a principal contribuição desta dissertação relacionada com o sistema DEPSPACE. Vale ressaltar que este protocolo foi implementado como uma biblioteca independente do DEPSPACE, podendo ser utilizado por qualquer aplicação.

3. **Projeto e desenvolvimento de uma infra-estrutura de coordenação e/ou cooperação de serviços *web*.**

   O capítulo 5 apresenta o projeto e desenvolvimento do WS-DEPENDABLESPACE, que representa esta infra-estrutura de coordenação e/ou cooperação de serviços *web*. Este sistema tem como base o DEPSPACE, onde uma série de novos mecanismos foram integrados com o objetivo de manter as propriedades de segurança de funcionamento do sistema. Além disso, este mesmo capítulo apresenta algumas aplicações que utilizam esta infra-estrutura como suporte para a execução de suas tarefas e aborda o relacionamento deste modelo de coordenação com algumas especificações para serviços *web* encontradas na literatura.

4. **Analisar os custos envolvidos no acesso ao WS-DEPENDABLESPACE.**

   O capítulo 6 apresenta uma análise detalhada dos custos envolvidos no acesso à infra-estrutura WS-DEPENDABLESPACE. Os vários mecanismos e protocolos desenvolvidos neste trabalho

foram analisados, onde foi possível determinar as causas relacionadas com os custos reportados pelos experimentos. Vários cenários foram analisados, variando-se desde a configuração do sistema até a presença de faltas e de concorrência no mesmo. Acreditamos que este é o primeiro trabalho que analisa este tipo de sistema, onde mecanismos de replicação e de confidencialidade são utilizados de forma integrada. No entanto, o *testbed* utilizado nestes experimentos foi um tanto limitado, o que prejudicou uma análise mais aprofundada sobre o desempenho destes sistemas (DEPSPACE e WS-DEPENDABLESPACE). Mesmo assim, é possível ter uma previsão do comportado destes sistemas nos diversos cenários analisados.

7.3 Perspectivas Futuras

Os estudos realizados nesta dissertação certamente não finalizam a discussão sobre tolerância a faltas e intrusões em sistemas desenvolvidos para operar tanto na Internet quanto em redes locais. Além do mais, este assunto é muito amplo e dificilmente as opções de pesquisa que o envolvem se esgotarão.

Deste modo, alguns trabalhos relacionados com esta dissertação poderão ser explorados no futuro. O primeiro destes trabalhos é a realização de testes mais amplos, onde análises mais significativas a respeito dos sistemas apresentados nesta dissertação seriam apuradas. Os experimentos apresentados no capítulo 6 poderiam ser executados em um *testbed* melhor, com um número maior de computadores, o que possibilitaria uma análise mais precisa da escalabilidade destes sistemas. Além disso, um trabalho muito interessante é a execução destes testes em um sistema de larga escala (Internet). Deste modo, poderíamos observar o comportamento destes sistemas neste ambiente, onde é sabido que a complexidade de troca de mensagens do protocolo de consenso PAXOS BIZANTINO (quadrática) prejudica a escalabilidade do sistema na medida que o número de servidores participantes da computação distribuída aumenta. Estes experimentos poderiam ser executados no PLANETLAB (*http://www.planet-lab.org/*), que é uma plataforma gratuita para o desenvolvimento e análise de sistemas na escala planetária. Note que, no WS-DEPENDABLESPACE o protocolo de difusão atômica pode ser executado em rede local, enquanto o sistema como um todo fica disponível na Internet, através dos *gateways* de acesso (ver capítulo 5).

Um trabalho que não foi explorado neta dissertação, mas que é muito interessante, é a realização de estudos no sentido de analisar a possibilidade de melhorar a escalabilidade do algoritmo de consenso PAXOS BIZANTINO (e indiretamente do protocolo de difusão atômica apresentado neste trabalho), através da diminuição tanto da complexidade de troca de mensagens quanto do número de passos necessários para decidir o consenso. Talvez uma forma de conseguir isso é através do uso de criptografia assimétrica nas mensagens trocadas entre os processos participantes do consenso. Considerando-se ambientes de larga escala, o custo relacionado com este processamento criptográfico pode ser menor do que o relacionado com as trocas de mensagens.

Além das questões envolvendo a escalabilidade, outros requisitos desejáveis quando consideramos sistemas tolerantes a intrusões, e que não foram explorados aqui, são a instalação de réplicas

(servidores) dos sistemas em diferentes domínios administrativos e fazer uso de diferentes plataformas de *software*. Estes tipos de diversidade substanciam a premissa de independência de faltas [60], comumente usada em algoritmos tolerantes a faltas bizantinas, fazendo, por exemplo, com que o sistema não seja completamente atingido por um ataque bem sucedido em um domínio administrativo ou uma vulnerabilidade apresentada por uma plataforma de *software*.

Outro trabalho que deve ter continuidade é o aperfeiçoamento das implementações realizadas e o desenvolvimento de uma aplicação completa que explore a maior quantidade possível das potencialidades fornecidas pela infra-estrutura WS-DEPENDABLESPACE.

Apêndice A

Manuais de Uso dos Sistemas

Este apêndice apresenta os manuais de uso dos sistemas DEPSPACE e WS-DEPENDABLESPACE (ou WSDS).

A.1 Manual de Uso do DEPSPACE

Esta seção descreve a forma de utilização do DEPSPACE, detacando as configurações necessárias no sistema e mostrando como proceder para que uma aplicação possa criar e acessar espaços lógicos.

A.1.1 Instalação

Para instalar o sistema DEPSPACE, siga as seguintes instruções:

1. Obtenha a última versão do sistema no site `http://www.das.ufsc.br/~neves/jitt/`;
2. Descompacte o pacote DepSpace.tgz (obtido no passo anterior) em algum diretório (chamado de "$< DepSpaceDir >$");
3. Configure as propriedades do sistema conforme discutido na seção A.1.2;
4. A instalação do DEPSPACE está completa! Veja como programar usando o DEPSPACE na seção A.1.3 e como executar este sistema na seção A.1.4.

A.1.2 Configurações do Sistema

Uma série de propriedades do sistema devem ser configuradas. Estas propriedades, que estão localizadas no diretório "$< DepSpaceDir > /config$", podem ser divididas em três partes:

1. Configurações do arquivo "***hosts.config***": Este arquivo descreve os *hosts* usados para executar as réplicas (servidores) do DEPSPACE. Devemos especificar o identificador[1] de cada servidor, seu endereço (*hostname* ou o endereço IP) e a porta na qual cada servidor estará recebendo conexões. Cada servidor deve ser especificado em uma nova linha.

2. Configurações do arquivo "***system.config***": Este arquivo contém a definição das diversas propriedades do sistema. Estas propriedades são divididas em três conjuntos:

   (a) Configurações de comunicação: Define as seguintes propriedades, relacionadas com o mecanismo de comunicação.

   ```
   # Authenticate the channels (are authenticated channels enables?).
   system.authentication = true

   # Diffie-Hellmam key exchange parameters (parameters for key
   # exchange and stabilishment of session keys). If not seted, the
   # default values will be used.
   system.authentication.P =
   system.authentication.G =

   # Auto connect with all hosts with id lower than the supplied id.
   system.autoconnect = 4
   ```

   (b) Configurações do protocolo de difusão atômica: Define as seguintes propriedades, relacionadas com o protocolo de difusão com ordem total (atômica) apresentado no capítulo 4 e utilizado pela camada de replicação do DEPSPACE. Normalmente não é necessário modificar estas propriedades, o que apenas deve ser feito por um usuário experiente.

   ```
   # Number of servers in the group (the number of servers "n" must
   # be more than "3f", to suport "f" bizantine faults).
   system.servers.num = 4
   system.servers.f = 1

   # Paxos Configurations
   # Timeout to freeze a round.
   system.paxos.freeze.timeout = 500

   # High mark, for safety margin.
   system.paxos.highMarc = 10

   # Period between executions of the paxos algorithm
   ```

[1] Este identificador é representado por um número (inteiro), sendo que os n servidores DEPSPACE são definidos como *hosts* de 0 até $n-1$.

```
# (to batch executions).
system.totalordermulticast.period = 10
```

(c) Configurações do mecanismo de confidencialidade: Define as propriedades relacionadas com o esquema de confidencialidade (ver seção 3.3.3). Devemos especificar os parâmetros do esquema PVSS e as chaves públicas e privadas de cada servidor (a chave privada de cada servidor deve ser definida apenas no arquivo armazenado em tal servidor). Estes parâmetros podem ser gerados automaticamente através da classe `br.ufsc.das.util.ConfidentialityParametersGenerator`, onde deve ser fornecido o número de servidores n (esta ferramenta gera n pares de chaves, além dos parâmetros do esquema PVSS).

3. Configurações das ***chaves RSA***: As chaves de todos os servidores são armazenadas no diretório "$< DepSpaceDir > /config/keys$". Cada servidor acessa todas as chaves públicas e apenas sua chave privada. Estas chaves podem ser geradas automaticamente através da classe `br.ufsc.das.util.RSAkeyPairGenerator`, onde o identificador do servidor (relacionado com o par de chaves que será gerado) deve ser fornecido.

A.1.3 Utilizando o DEPSPACE na Implementação de uma Aplicação

Com o objetivo de mostrar como uma aplicação cliente deve proceder para criar e acessar espaços lógicos, esta seção apresenta uma implementação de 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()*).

A implementação desta aplicação 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 que indica o número de processos que podem falhar dentre os que estão se coordenando (sincronizando). Este parêmetro (f) é usado na implementação da operação *enter()*, que compreende 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 código para uma classe Java que implementa esse serviço[2] é apresentado na figura A.1.

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 A.1 suporta um parâmetro f que define o número máximo de processos

[2]O código não contém tratamento de erros.

que assumimos que podem não chegar na 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 [2], o uso eficiente desta abstração em ambientes abertos só faz sentido desta forma.

```
public class Barrier {
  private DepSpaceAccessor accessor; // referência para o espaço de tuplas lógico
  private String name; // nome da barreira
  private Collection c; // conjunto de processos a serem esperados
  private int f; // número de processos que podem falhar, dentre os processos esterados
  private int myself; // identificador do processo em questão
  private static int POOL_TIME = 100; // período de consulta ao DepSpace

  public Barrier(DepSpaceAccessor accessor, String name, Collection c, int f, int myself) {
        this.accessor = accessor;
        this.name = name;
        this.myself = myself;
        DepTuple template = DepTuple.createTuple("BARRIER",name,"*","*");
        DepTuple tupla = DepTuple.createTuple("BARRIER",name,c,f);
        DepTuple ret = accessor.cas(template,tupla); // tenta criar a barreira "name"
        if(ret == null){ // criou a barreira "nome"
              this.c = c;
              this.f = f;
         }else{ // barreira "name" já estava criada
             this.c = (Collection) ret.getFields()[2];
             this.f = Integer.valueOf(ret.getFields()[3].toString());
        }
   }

   public void enter(){
       if(accessor.rdp(DepTuple.createTuple("BARRIER",name,"*","*")) != null){
            accessor.out(DepTuple.createTuple("ENTER",name,myself));
            do{
                for(int i =0; i < c.size(); i++){
                      if(accessor.rdp(DepTuple.createTuple("ENTER",name,c.get(i))) != null){
                            p.remove(i);
                            i--;
                        }
                 }
                 wait(POOL_TIME);
             }while(c.size() > f);
        }else{
             throw new RuntimeException("The barrier "+name+" does not exists.");
        }
    }
}
```

Figura A.1: Código para barreira de sincronização.

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 A.2 que apresenta a classe `PolicyBarrier`. Note que esta classe estende a classe `PolicyEnforcer` conforme discutido na seção 3.4.4.

Vale notar que esta implementação é muito simples. No entanto, dada a versatilidade do DEPSPACE, mesmo características avançadas das barreiras, como as apresentadas em [2], podem ser im-

plementadas. Neste mesmo trabalho podem ser encontradas várias aplicações para as barreiras com semântica fraca.

```
public class PolicyBarrier extends PolicyEnforcer{

  public boolean canExecuteOut(int invoker, DepTuple tuple, Context ctx){
    if(tuple.getFields() != null){
       if(tuple.getFields()[0].equals("ENTER") && tuple.getFields().length == 3){
           DepTuple t = rdp(DepTuple.createTuple("BARRIER",tuple.getFields()[1],"*","*"),ctx);
           // Somente o próprio processo pode invocar sua entrada na barreira
           // e para isso tal barreira deve existir.
           if((t != null) && (Integer.valueOf(tuple.getFields()[2].toString()) == invoker)){
              Collection c = (Collection) t.getFields()[2];
              // Tal processo deve ser um dos processo aguardados
              if (c.contains(invoker)){
                t = DepTuple.createTuple("ENTER",tuple.getFields()[1],tuple.getFields()[2]);
                // Apenas é permitido entrar uma vez na barreira
                return (rdp(t,ctx) == null);
              }
           }
        }
     }
     return false;
  }

  public boolean canExecuteRdp(int invoker, DepTuple template, Context ctx){
     //Todo processo pode ler toda tupla
     return true;
  }

  public boolean canExecuteInp(int invoker, DepTuple template, Context ctx){
     //Nenhum processo pode remover tuplas
     return false;
  }

  public boolean canExecuteCas(int invoker, DepTuple template, DepTuple tuple, Context ctx){
    if(tuple.getFields() != null){
       if(tuple.getFields()[0].equals("BARRIER") && tuple.getFields().length == 4){
          DepTuple t = DepTuple.createTuple("BARRIER",tuple.getFields()[1],"*","*");
          //Não é permitido inserir duas barreiras com o mesmo nome
          return (rdp(t,ctx) == null);
        }
     }
     return false;
  }
}
```

Figura A.2: Política de controle de acesso para barreira de sincronização.

Além de implementar a classe que representa a barreira (`Barrier`) e definir as políticas de segurança que controlam o acesso ao espaço lógico (`PolicyBarrier`), também é necessário criar o espaço lógico e o objeto (`DepSpaceAccessor`) responsável pelo acesso a este espaço. Este processamento é executado pela classe `Main` (figura A.3), onde o espaço lógico é criado através do método *main*[3]. Após isso, dez (10) barreiras são criadas e utilizadas na sincronização dos processos (método *run*). Na execução desta aplicação devemos informar (como parâmetros): o identificador do processo

[3]Este método considera que a política de segurança apresentada na figura A.2 foi salva no arquivo *PolicyBarrier.txt*.

em questão (*myself*), o número de processos esperados (p), o número de processos que podem ser faltosos dentre os processos esperados (f) e um quarto parâmetro (a *string create*) indicando a necessidade de criar o espaço lógico (apenas um processo criará o espaço lógico). A forma correta de executar esta aplicação é abordada na seção A.1.4.

```
public class Main{

  public static void main(String[] args){
    int myself = Integer.valueOf(args[0]);
    int p = Integer.valueOf(args[1]);
    int f = Integer.valueOf(args[2]);
    // aumentar o identificador para não dar conflito com os identificadores dos servidores.
    myself = myself+100;
    Collection c = new LinkedList();
    for(int i = 0; i< p;i++){ c.add(100+i); }
    Properties prop = new Properties();
    prop.put(DPS_NAME,"barrier space"); // nome do espaço lógico
    prop.put(CLIENT_ID,String.valueOf(myself)); // identificador do processo
    prop.put(DPS_POLICY_ENFORCEMENT,loadFile(PolicyBarrier.txt)); // política de segurança
    prop.put(DPS_CONFIDEALITY,"false"); // uso de confidencialidade
    // objeto DepSpaceAccessor responsável pelo acesso ao DepSpace.
    DepSpaceAccessor accessor = null;
    if(args.length >= 4 && args[3].equals("create")){//cria o DepSpaceAccessor e o espaço lógico
      accessor = new DepSpaceAdmin().createSpace(prop);
    }else{ // cria apenas o DepSpaceAccessor
      accessor = new DepSpaceAdmin().createAccessor(prop);
    }
    new Main().run(accessor,"Barreira Demo",c,f,myself);
  }

  private static String loadFile(String path) throws Exception{
    BufferedReader fr = new BufferedReader(new FileReader(new File(path)));
    String s =""; String ret="";
    while((s=fr.readLine()) != null){ ret = ret+s+"\n";}
    return ret;
  }

  public void run(DepSpaceAccessor accessor, String name, Collection c, int f, int myself){
    for(int i = 0; i < 10; i++){
       wait(500+(int)(10000*Math.random())); // espera um tempo aleatório e entra na barreira
       System.out.println("Entrando na "+name+" "+i);
       new Barrier(accessor,name+" "+i,c,f,myself).enter();
       System.out.println("Passou pela "+name+" "+i);
    }
    System.out.println("FIM!");
  }
}
```

Figura A.3: Criação do espaço lógico.

É importante perceber que todo o processamento nas camadas do cliente é transparente para a aplicação, i.e., o programador não precisa se preocupar com nada, tudo é controlado pelas camadas localizadas no cliente, isto também vale para o sistema WSDS (como por exemplo a invocação de mais f_g *gateways* devido ao atendimento incompleto de requisições - ver seção 5.5.3.2).

Outro ponto a destacar é que qualquer objeto pode ser utilizado como campo das tuplas. Entre-

tanto, estes objetos devem implementar os métodos *toString* (usado pelo esquema de confidencialidade - ver seção 3.4.2) e *equals* (usado pela camada de replicação – ver seção 3.4.1 – e na comparação entre tuplas e moldes). Outras aplicações exemplo são encontradas no pacote `br.ufsc.das.demo`.

A.1.4 Executando o sistema

A execução do sistema é bastante simples. Basicamente é necessário inicializar as réplicas (servidores) usando o *script main.bat* (ou apenas *main* para o sistema operacional Linux) encontrado no diretório $< DepSpaceDir >$. O identificador de cada servidor deve ser fornecido em cada execução.

Ex: Para executar o sistema com quatro servidores (suportando uma falta) devemos executar:
>main.bat 0
>main.bat 1
>main.bat 2
>main.bat 3

Após isso, os clientes podem acessar o espaço de tuplas, criando espaços lógicos e/ou executando operações nestes espaços lógicos já criados. No caso da aplicação utilizada como exemplo neste apêndice (ver seção A.1.3) os seguintes comandos devem ser executados[4] na sincronização de quatro processos sendo que um deles pode ser faltoso:
>java Main 0 4 1 create
>java Main 1 4 1
>java Main 2 4 1
>java Main 3 4 1

A.2 Manual de Uso do WS-DEPENDABLESPACE

A forma de utilização do WS-DEPENDABLESPACE (ou WSDS) é idêntica a do DEPSPACE. Sendo assim, tudo aquilo que foi discutido na seção A.1 também vale para o WSDS. A única diferença é na criação do objeto `DepSpaceAccessor` (responsável pelo acesso ao espaço) que deve ser realizada através da classe `WSDepSpaceAdmin`. Esta nova interface administrativa é idêntica à interface `DepSpaceAdmin` (ver seção 3.4.6), sendo responsável pela configuração das camadas no cliente, fazendo com que o mesmo acesse os *gateways* (de forma transparente para a aplicação).

Além disso, neste sistema existe uma arquivo de configurações adicional ("$< DepSpaceDir > /config/ws.config$"), onde os endereços dos repositórios de interfaces devem ser informados. Neste mesmo arquivo, é possível definir o tempo que um cliente deve esperar pelo recebimento de uma resposta (*timeout*) antes de considerar que o *gateway* acessado falhou (ver seção 5.5.3).

Outro ponto a destacar é que os *gateways* devem ser instalados em algum servidor HTTP (ex. no container J2EE *Tomcat* 5.5.20 [5], conforme discutido na seção 5.6).

[4] A configuração do *classpath* não é apresentada, mas pode ser encontrada no *script client.bat* (ou *client* para o sistema operacional Linux) localizado no diretório $< DepSpaceDir >$.

Referências Bibliográficas

[1] Abd-El-Malek, M., Ganger, G., Goodson, G., Reiter, M., e Wylie, J. "Fault-Scalable Byzantine Fault-Tolerant Services". In *Proceedings of the 20th ACM Symposium on Operating Systems Principles - SOSP'05*, pp. 59–74, outubro de 2005.

[2] Albrecht, J., Tuttle, C., Snoeren, A. C., e Vahdat, A. "Loose Synchronization for large-scale networked systems". In *Proceedings of the 2006 Usenix Annual Technical Conference – Usenix'06*, 2006.

[3] Alchieri, E. A. P., Bessani, A. N., e da Silva Fraga, J. "Infra-Estrutura com Segurança de Funcionamento para Cooperação de Serviços Web". In *Anais do 25º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos- SBRC 2007*, Belém, PA, Brasil, maio de 2007.

[4] Alves A. et al. "Web Services Business Process Execution Language, Version 2.0". OASIS Public Draft. Disponível em `http://docs.oasis-open.org/wsbpel/2.0/`, março de 2007.

[5] Apache. "Apache Tomcat". Disponível em `http://tomcat.apache.org/`, março de 2007.

[6] Apache. "Axis: An Implementation of the SOAP Protocol.". Disponível em `http://ws.apache.org/axis/`, março de 2007.

[7] Aspnes, J. "Randomized Protocols for Asynchronous Consensus". *Distributed Computing*, 16 (2-3):165–175, 2003.

[8] Attiya, H. e Welch, J. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley Series on Parallel and Distributed Computing. Wiley-Interscience, 2nd edition, 2004.

[9] Avizienis, A., Laprie, J.-C., Randell, B., e Landwehr, C. "Basic Concepts and Taxonomy of Dependable and Secure Computing". *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, março de 2004.

[10] Bakken, D. E. e Schlichting, R. D. "Supporting Fault-Tolerant Parallel Programing in Linda". *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302, março de 1995.

[11] Bellur, U. e Bondre, S. "xSpace: a Tuple Space for XML and its Application in Orchestration of Web Services". In *Proceedings of the 2006 ACM Symposium on Applied Computing – SAC 2006*, pp. 766–772, 2006.

[12] Bessani, A. N. *Coordenação Desacoplada Tolerante a Faltas Bizantinas*. Tese de doutorado em engenharia elétrica, Universidade Federal de Santa Catarina, Florianópolis, 2006.

[13] Bessani, A. N., Alchieri, E. A. P., Correia, M., da Silva Fraga, J., e Lung, L. C. "DepSpace - Um Middleware para Coordenação em Ambientes Dinâmicos e Não Confiáveis". In *Anais do 25º Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (Salão de Ferramentas)*, Belém - PA - Brasil, maio de 2007.

[14] Bessani, A. N., Alchieri, E. A. P., da Silva Fraga, J., e Lung, L. C. "Design and Implementation of an Intrusion-Tolerant Tuple Space". In *Proceedings of the International Workshop on Recent Advances on Intrusion-Tolerant Systems (with EuroSys 2007)*, março de 2007.

[15] Bessani, A. N., Alchieri, E. A. P., Correia, M., Fraga, J. S., e Lung, L. C. "Provendo Confidencialidade em Espaços de Tuplas Tolerantes a Intrusões". In *Anais do 6o Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg 2006*, agosto de 2006.

[16] Bessani, A. N., Correia, M., Fraga, J. S., e Lung, L. C. "Sharing Memory Between Byzantine Processes Using Policy-Enforced Tuple Spaces". In *Proceedings of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*, julho de 2006.

[17] Bessani, A. N., da Silva Fraga, J., e Lung, L. C. "BTS: A Byzantine Fault-Tolerant Tuple Space". In *Proceedings of the 21st ACM Symposium on Applied Computing*, abril de 2006.

[18] Bessani, A. N., Dantas, W. S., Alchieri, E. A. P., e da Silva Fraga, J. "Analisando o Custo do Armazenamento Tolerante a Faltas Bizantinas: PAXOS X Sistemas de Quóruns". In *Anais do 7o. Workshop de Testes e Tolerância a Faltas - WTF 2006*, Curitiba, PR, Brasil, junho de 2006.

[19] Bichier, M. e Lin, K.-J. "Service-Oriented Computing". *IEEE Computer*, 39(3):99–101, março de 2006.

[20] Bishop, M. *Computer Security: Art and Science*. Addison-Wesley, 2002.

[21] Bright, D. e Quirchmayr, G. "Supporting Web-Based Collaboration Between Virtual Enterprise Partners". In *Proceedings of the 15th International Workshop on Database and Expert Systems Applications – DEXA 2004*, 2004.

[22] Burdett, D. e Kavantzas, N. "The WS-Choreography Model Overview". W3C Draft. Disponível em `http://www.w3.org/TR/ws-chor-model/`, março de 2007.

[23] Busi, N., Gorrieri, R., Lucchi, R., e Zavattaro, G. "SecSpaces: a Data-Driven Coordination Model for Environments Open to Untrusted Agents". In *Electronic Notes in Theoretical Computer Science*, volume 68, 2003.

[24] Cachin, C. e Samar, A. "Secure Distributed DNS". In *International Conference on Dependable Systems and Networks (DSN'04)*, pp. 423–433, 2004.

[25] Carriero, N. e Gelernter, D. "How to Write Parallel Programs: a Guide to the Perplexed". *ACM Computing Surveys*, 21(3):323–357, setembro de 1989.

[26] Castro, M. e Liskov, B. "Practical Byzantine Fault-Tolerance and Proactive Recovery". *ACM Transactions on Computer Systems*, 20(4):398–461, novembro de 2002.

[27] Castro, M., Rodrigues, R., e Liskov, B. "BASE: Using Abstraction to Improve Fault Tolerance". *ACM Transactions Computer Systems*, 21(3):236–269, 2003.

[28] Chandra, T. D. e Toueg, S. "Unreliable Failure Detectors for Reliable Distributed Systems". *Journal of the ACM*, 43(2), março de 1996.

[29] Charron-Bost, B. e Schiper, A. "Uniform Consensus is Harder than Consensus (extended abstract)". Technical Report DSC/2000/028, Swiss Federal Institute of Technology, Lausanne, Switzerland, maio de 2000.

[30] Codehaus. "Groovy: An Agile Dynamic Language for the Java Platform". Disponível em `http://groovy.codehaus.org/`, março de 2007.

[31] Correia, M., Neves, N. F., e Veríssimo, P. "From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols Without Signatures". *The Computer Journal*, 49(1), janeiro de 2006.

[32] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., e Shrira, L. "HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance". In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations (OSDI)*, Seattle, Washington, novembro de 2006.

[33] De Nicola, R., Ferrari, G. L., e Pugliese, R. "KLAIM: A Kernel Language for Agents Interaction and Mobility". *IEEE Transactions on Software Engineering*, 24(5):315–330, maio de 1998.

[34] Doudou, A. e Schiper, A. "Muteness Detectors for Consensus with Byzantine Processes". In *Proceedings of the 17th annual ACM Symposium on Principles of Distributed Computing (Brief announcements)*, p. 315, 1998. ISBN 0-89791-977-7.

[35] Dutta, P., Guerraoui, R., e Vukolic, M. "Best-Case Complexity of Asynchronous Byzantine Consensus". Technical Report EPFL/IC/200499 (revised version), Distributed Programming Laboratory - EPFL, fevereiro de 2005.

[36] Dwork, C., Lynch, N. A., e Stockmeyer, L. "Consensus in the Presence of Partial Synchrony". *Journal of ACM*, 35(2):288–322, abril de 1988.

[37] Favarim, F., Correia, M. P., Lung, L. C., e Fraga, J. "Fault-Tolerant Multiuser Computational Grids Based on Tuple Spaces". In *Proceedings of the International Workshop on Dependability in Service-oriented Grids (with SRDS 2006)*, outubro de 2006.

[38] Ferraza, R., Gonçalves, B., Sequeira, J., Correia, M., Neves, N. F., e Veríssimo, P. "An Intrusion-Tolerant Web Server Based on the DISTRACT Architecture". In *Proceedings of the Workshop on Dependable Distributed Data Management (WDDDM)*, Florianópolis - Brasil, 2004.

[39] Fischer, M. J., Lynch, N. A., e Paterson, M. S. "Impossibility of Distributed Consensus with One Faulty Process". *Journal of the ACM*, 32(2):374–382, abril de 1985.

[40] Fraga, J. e Powell, D. "A Fault- and Intrusion-Tolerant File System". In *Proceedings of the 3rd Int. Conference on Computer Security*, pp. 203–218, 1985.

[41] Gelernter, D. "Generative Communication in Linda". *ACM Transactions on Programing Languages and Systems*, 7(1):80–112, janeiro de 1985.

[42] Gifford, D. "Weighted Voting for Replicated Data". In *Proc. of the 7th ACM Symposium on Operating Systems Principles*, pp. 150–162, dezembro de 1979.

[43] GigaSpaces. "GigaSpaces Homepage". Disponível em `http://www.gigaspaces.com/`, março de 2007.

[44] Graham S. at al. "WS-Base Notification". Disponível em `ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BaseN.pdf`, março de 2007.

[45] Gupta, M., Parihar, M., Lasalle, P., e Scrimger, R. *TCP/IP A Bíblia*. Campus/Elsevier, 2002.

[46] Hadzilacos, V. e Toueg, S. "A Modular Approach to the Specification and Implementation of Fault-Tolerant Broadcasts". Technical Report TR94-1425, Department of Computer Science, Cornell University, New York - USA, maio de 1994.

[47] Herlihy, M. e Wing, J. M. "Linearizability: A Correctness Condition for Concurrent Objects". *ACM Transactions on Programing Languages and Systems*, 12(3):463–492, julho de 1990.

[48] Jeong, K. e Shasha, D. "PLinda 2.0: A Transactional Checkpointing Approach to Fault Tolerant Linda". In *Proceedings of the 13th Symposium on Reliable Distributed Systems.*, pp. 96–105, Dana Point, CA, USA, outubro de 1994.

[49] Junqueira, F. P. e Marzullo, K. "Synchronous Consensus for Dependent Process Failures". In *Proceedings of 23th IEEE International Conference on Distributed Computing Systems - ICDCS 2003*, 2003.

[50] L. F. Cabrera et al. "Web Services Coordination Specification - version 1.0". Disponível em `http://www-128.ibm.com/developerworks/library/specification/ws-tx/`, março de 2007.

[51] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System". *Communications of the ACM*, 21(7):558–565, julho de 1978.

[52] Lamport, L. "The Part-Time Parliament". *ACM Transactions Computer Systems*, 16(2):133–169, maio de 1998. ISSN 0734-2071.

[53] Lamport, L. "Paxos Made Simple". *ACM SIGACT News*, 32(4):18–25, dezembro de 2001.

[54] Lamport, L., Shostak, R., e Pease, M. "The Byzantine Generals Problem". *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, julho de 1982.

[55] Lampson, B. "The ABCD's of Paxos". In *Proceedings of the 20th annual ACM Symposium on Principles of Distributed Computing*, p. 13, 2001. ISBN 1-58113-383-9.

[56] Lucchi, R. e Zavattaro, G. "WSSecSpaces: a Secure Data-Driven Coordination Service for Web Services Applications". In *Proc. of the 19th ACM Symposium on Applied Computing*, pp. 487–491, março de 2004.

[57] Maamar, Z., Benslimane, D., Ghedira, C., Mahmoud, Q. H., e Yahyaoui, H. "Tuple Spaces for Self-Coordination of Web Services". In *Proceedings of the 2005 ACM Symposium on Applied Computing – SAC 2005*, pp. 1656–1660, 2005.

[58] Martin, J.-P. e Alvisi, L. "Fast Byzantine Consensus". *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, julho de 2006.

[59] OASIS. "Universal Description, Discovery and Integration v3.0.2 (UDDI).". Organization for the Advancement of Structured Information Standards (OASIS), 2004.

[60] Obelheiro, R. R., Bessani, A. N., e Lung, L. C. "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*, 2005.

[61] Object Management Group. "Fault-Tolerant CORBA Specification v1.0". OMG Standart, 2000.

[62] Object Management Group. "The Common Object Request Broker Architecture: Core Specification v3.0". OMG Standart formal/02-12-06, dezembro de 2002.

[63] Papadopolous, G. e Arbab, F. "Coordination Models and Languages". In *The Engineering of Large Systems*, volume 46 de *Advances in Computers*. Academic Press, agosto de 1998.

[64] Papazoglou, M. P. "Service-Oriented Computing: Concepts, Characteristics and Directions". In *Proceedings of 4th International Conference on Web Information Systems Engineering*, 2003.

[65] Peltz, C. "Web Services Orchestration and Choreography". *IEEE Computer*, 36(10):46–52, outubro de 2003.

[66] Ricci, A., Omicini, A., e Denti, E. "The TuCSoN Coordination Infrastructure for Virtual Enterprises". In *Proceedings of the 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pp. 348–353, 2001.

[67] Rivest, R. L., Shamir, A., e Adleman, L. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". *Communications of the ACM*, 21(2):120–126, 1978. ISSN 0001-0782.

[68] Schneider, F. B. "Implementing Fault-Tolerant Service Using the State Machine Aproach: A Tutorial". *ACM Computing Surveys*, 22(4):299–319, dezembro de 1990.

[69] Schoenmakers, B. "A Simple Publicly Verifiable Secret Sharing Scheme and its Application to Electronic Voting". In *Advances in Cryptology - CRYPTO'99*, volume 1666 de *LNCS*, pp. 148–164, agosto de 2004.

[70] Segall, E. J. "Resilient Distributed Objects: Basic Results and Applications to Shared Spaces". In *Proc. of the 7th Symposium on Parallel and Distributed Processing, SPDP'95*, pp. 320–327, outubro de 1995.

[71] Shotton, P. "High Performance Program Trading". Disponível em `http://www.gigaspaces.com/`, março de 2007.

[72] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., e Surana, S. "Internet Indirection Infrastructure". *IEEE/ACM Transactions on Networking*, 12(2):205–218, 2004.

[73] Sun Microsystems. "JavaSpaces Service Specification". Disponível em `http://java.sun.com/products/jini/2.0/doc/specs/html/js-spec.html`, março de 2007.

[74] T. J. Lehman et al. "Hitting the Distributed Computing Sweet Spot with TSpaces". *Computer Networks*, 35(4):457–472, março de 2001.

[75] Tsudik, G. "Message Authentication with One-Way Hash Functions". In *ACM Computer Communications Review*, pp. 29–38, 1992.

[76] Veríssimo, P., Neves, N. F., e Correia, M. P. "Intrusion-Tolerant Architectures: Concepts and Design". In *Architecting Dependable Systems*, volume 2677 de *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[77] Vitek, J., Bryce, C., e Oriol, M. "Coordination Processes with Secure Spaces". *Science of Computer Programming*, 46(1-2):163–193, janeiro de 2003.

[78] W3C Working Group. "Web Services Architecture". Disponível em `http://www.w3.org/TR/2004/NOTE-ws-arch-20040211`, março de 2007.

[79] Weerawarana, S., Curbera, F., Leymann, F., Storey, T., e Ferguson, D. F. *Web Services Platform Architecture*. Prentice Hall, 2005.

[80] Xu, A. e Liskov, B. "A Design for a Fault-Tolerant, Distributed Implementation of Linda". In *Proc. of the 19th Symposium on Fault-Tolerant Computing - FTCS'89*, pp. 199–206, junho de 1989.

[81] Yegneswaran, V., Barford, P., e Jha, S. "Global Intrusion Detection in the DOMINO Overlay System". In *Proceedings of Network and Distributed Security Symposium – NDSS 2004*, fevereiro de 2004.

[82] Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., e Dahlin, M. "Separating Agreement form Execution for Byzantine Fault Tolerant Services". In *Proceedings of the 19th ACM Symposium on Operating Systems Principles - SOSP'03*, pp. 253–267, outubro de 2003. ISBN 1-58113-757-5.

[83] Zielinski, P. "Paxos at War". Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK, junho de 2004.