Alysson Neves Bessani

Coordenação Desacoplada Tolerante a Faltas Bizantinas

FLORIANÓPOLIS
Julho de 2006

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

Coordenação Desacoplada Tolerante a Faltas Bizantinas

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

Alysson Neves Bessani

Florianópolis, Julho de 2006.

Coordenação Desacoplada Tolerante a Faltas Bizantinas

Alysson Neves Bessani

'Esta Tese foi julgada adequada para a obtenção do título de Doutor em Engenharia Elétrica, Área de Concentração em *Automação e Sistemas*, 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:

Lau Cheuk Lung (Co-Orientador)

Raimundo José de Araújo Macêdo

Luiz Eduardo Buzato

Miguel Pupo Correia

Rômulo Silva de Oliveira

À minha mãe Marilse, por tudo...

À minha avó Otávia (in memoriam), pelo exemplo de vida...

À minha Cássia, pelo amor e pelo apoio...

AGRADECIMENTOS

Em primeiro lugar agradeço a Deus por me dar força e inspiração para a realizar este trabalho, além de me cercar de pessoas maravilhosas.

Agradeço a minha família (meu pai, Odorico, meus irmãos, Alencar e Luana, e todos os meus tios e primos) pelo apoio indispensável durante todos esses anos. Em especial à minha mãe Marilse Dias Neves, pois tudo que tenho e sou, eu devo a ela.

Muitas pessoas contribuiram para concepção deste trabalho. Agradeço aos meus orientadores, Joni e Lau, pelo apoio contínuo e por me colocarem nos trilhos nos momentos em que estes pareciam me faltar. Agradeço aos colegas do DAS/PGEEL (Brandão, Fábio Favarim, Michele, Emerson) pelo companherismo e pelas discussões e sugestões que muito melhoraram o presente trabalho. Não posso deixar de fazer um agradecimento especial a Rafael Rodrigues Obelheiro, Eduardo Adílio Pelinson Alchieri e Wagner Saback Dantas pelas inúmeras discussões sobre os temas envolvidos nesta tese, e por ajudarem com sugestões e correções em diversos aspectos da mesma.

Agradeço também aos outros amigos do DAS/LCMI que tornaram mais fáceis esse 5 anos e meio de mestrado e doutorado: Ricardo, Fabiano Baldo, Fábio Pinga, Luís Fernando, Renato Machado, Tati, Luciana, Leandro, Tércio, Cris e Priscila. Um agradecimento especial ao Fábio Baiano pelo companheirismo e pelas nossas espetaculares "surfadas".

Esta tese seria muito mais pobre se eu não tivesse contado com a ajuda direta do professor Miguel Correia durante minha estada em Portugal. Pelo rigor, dedicação, simpatia e várias boas idéias deixo-lhe meus mais sinceros agradecimentos. Agradeço também aos diversos amigos que fiz no LaSIGE/Universidade de Lisboa (Henrique Moniz, Paulo Sousa, João Antunes e demais amigos) pelo ambiente amigável e propício a pesquisa que me proporcionaram. Finalmente, obrigado a Dona Odete por fazer eu me sentir em casa durante aqueles frios (e produtivos) quatro meses de 2005/2006.

Agradeço aos meus amigos de fé, meus irmãos camaradas: Marim, Marcel, Truper, Vladi, Rorso, Beto, Rafael, Adamô, Guidão, Fabian, Sr. Regis e demais pessoas de bem que tive o prazer de encontrar pela vida. Deus cria, o homem espalha e o diabo junta!

Agradeço ao meu amor, Cássia, pelo carinho , paciência e suporte infinitos. Não posso listar aqui a quantidade de vezes que seu apoio foi fundamental, só posso esperar que eu possa contar com esse apoio para toda a vida.

"A desobediência é uma virtude indispensável para a criatividade"

- Raul Seixas

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

Coordenação Desacoplada Tolerante a Faltas Bizantinas

Alysson Neves Bessani

Julho de 2006

Orientador: Joni da Silva Fraga
Área de Concentração: Automação e Sistemas
Palavras-chave: Algoritmos Distribuídos, Coordenação, Tolerância a Faltas Bizantinas, Segurança Computacional
Número de Páginas: xiii + 128

Sistemas distribuídos abertos são tipicamente compostos por um número desconhecido e variável de processos executando em um ambiente heterogêneo, onde as comunicações muitas vezes requerem desconexões temporárias e segurança contra ações maliciosas. A coordenação por espaço de tuplas é um modelo de comunicação bastante conhecido para estes ambientes pois provê comunicação desacoplada tanto no tempo (os processos não precisam estar ativos ao mesmo tempo para interagirem) quanto no espaço (os processos não necessitam saber os endereços uns dos outros). Vários trabalhos têm tentado melhorar a segurança de funcionamento dos espaços de tuplas através do uso de replicação e transações para tolerância a faltas ou controle de acesso e criptografia para segurança. Entretanto, muitas aplicações práticas na Internet requerem ambas estas dimensões. Nesta tese, o modelo de coordenação por espaços de tuplas é usado para resolver o problema da coordenação desacoplada em ambientes não confiáveis, i.e., onde os processos estão sujeitos a falhas bizantinas (podem desviar-se arbitrariamente de suas especificações). Os resultados aqui apresentados atacam dois problemas básicos: (1) como construir espaços de tuplas com segurança de funcionamento (seguros e tolerantes a faltas bizantinas), e (2) como usar estes espaços para resolução de problemas fundamentais em computação distribuída. Os resultados referentes a (1) são uma arquitetura para espaço de tuplas com segurança de funcionamento que integra mecanismos de segurança e tolerância a faltas, duas contruções eficientes para espaços de tuplas tolerantes a faltas bizantinas baseadas em uma nova filosofia de replicação, e um esquema de confidencialidade para espaços de tuplas replicados. Com relação a (2), é mostrado que um espaço de tuplas aumentado protegido por políticas de granularidade fina pode ser usado para resolver eficientemente vários problemas em computação distribuída mesmo com processos sujeitos a faltas bizantinas.

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

Decoupled Byzantine Fault-Tolerant Coordination

Alysson Neves Bessani

July/2006

Advisor: Joni da Silva Fraga
Area of Concentration: Automation and Systems
Key words: Distributed Algorithms, Coordination, Byzantine Fault Tolerance, Computer Security
Number of Pages: xiii + 128

Open distributed systems are typically composed by an unknown number of processes running in heterogeneous hosts. Their communication often requires tolerance to temporary disconnections and security against malicious actions. Tuple spaces are a well-known coordination model for this sort of systems. They can support communication that is decoupled both in time (processes do not have to be active at the same time) and space (processes do not need to know each others addresses). Several works have tried to improve the dependability of tuple spaces. Some of them made tuple spaces fault-tolerant, using mechanisms like replication and transactions, while others have focused on security, using access control and cryptography. However, many practical applications in the Internet require both these dimensions. In this thesis, the tuple space coordination model is used to solve the problem of decoupled coordination in untrusted environments, i.e., where processes can be subject to Byzantine failures (deviating arbitrarily from their specifications). The results presented in this thesis solve two problems: (1) how to build a dependable tuple space (secure and Byzantine fault-tolerant), and (2) how to use this tuple space to solve fundamental coordination problems in distributed computing. The results regarding (1) are a dependable tuple space architecture that integrates security and fault tolerance mechanisms, two new efficient constructions for Byzantine fault-tolerant tuple spaces based on a new replication philosophy, and an scheme for replicated tuple space confidentiality. Considering (2), this thesis shows that an augmented tuple space protected by fine-grained security policies can be used to efficiently solve several distributed computing problems even with processes that can suffer Byzantine failures.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Capítulo 1

Introdução

O crescimento do uso da Internet em praticamente todas as áreas do conhecimento estimula o desenvolvimento de novas tecnologias e aplicações nesse ambiente de larga escala. Neste tipo de ambiente, os participantes de uma aplicação distribuída são caracterizados pela heterogeneidade, capacidade de comunicação variável (e por vezes, irregular) e pela não confiabilidade. Em um futuro próximo, a necessidade de sistemas para suporte a aplicações distribuída em larga escala se tornarão comuns e para tratar deste potencial problema novos modelos e paradigmas de computação vêm sendo propostos. Alguns exemplos são as redes *ad hoc*, os serviços *web*, as redes par a par, os sistemas de computação móvel e a computação em grade. Em muitos aspectos, estes modelos/paradigmas se baseiam no conceito de sistemas abertos[1].

Os sistemas abertos são caracterizados como sendo compostos por um número desconhecido de processos executando em ambientes heterogêneos que interagem em diferentes momentos de uma execução do sistema. Outro aspecto fundamental deste tipo de sistema é a possibilidade de que seus componentes evoluam (por exemplo, oferecendo novos serviços). Em geral, a concepção de sistemas (distribuídos) abertos só é possível através de mecanismos de comunicação flexíveis e padrões abertos.

Os mecanismos de comunicação usualmente empregados para a interação em sistemas distribuídos abertos, como a comunicação direta por passagem de mensagens, não são adequados para os cenários de computação ubíqua dos sistemas abertos do futuro. Os requisitos de anonimato e não confiabilidade nas comunicação implicam diretamente na necessidade de interações desacopladas. Assim, novos modelos de comunicação se fazem necessários.

Um aspecto fundamental que estes novos modelos de coordenação devem contemplar é a **segurança de funcionamento** (*dependability*) [10] dos sistemas neles baseados. O conceito de segurança de funcionamento está diretamente relacionado à capacidade do sistema em sobreviver a ataques e falhas de seus componentes. Os ataques geralmente objetivam comprometer as propriedades de segurança do sistema e quando bem sucedidos podem causar falhas nos componentes atacados. Estas

[1] Consideramos o conceito de **sistemas abertos** da teoria de sistemas, em que um sistema é dito aberto se seus limites não são bem definidos. No nosso caso, um sistema aberto é um sistema com um número variável e desconhecido de elementos. Não confundir com sistemas baseados em padrões abertos.

falhas podem também ser provocadas pelo mal funcionamento de componentes ou da infra-estrutura (hardware ou mecanismo de comunicação) do sistema. Neste contexto, a tolerância a este tipo de faltas[2] é um requisito de qualidade de serviço de grande importância.

1.1 Motivação

Um modelo de coordenação pode ser definido a partir da especificação das entidades que interagem, da mídia usada para a interação e das regras que devem ser obedecidas pelas entidades para o uso desta mídia. Existem vários tipos de modelos de coordenação, porém a coordenação generativa (também chamada "por espaço de tuplas") se apresenta como o modelo mais flexível e simples, sendo portanto, o mais adequado para sistemas abertos. Neste modelo, os participantes de uma computação distribuída interagem através de um espaço de memória compartilhado em que estruturas de dados genéricas (chamadas tuplas) são inseridas, lidas e coletadas durante as interações. A coordenação ocorre de maneira **desacoplada** no tempo e no espaço: os participantes não precisam conhecer uns aos outros e nem estar engajados na interação simultaneamente para que ela aconteça. Além disso, o número reduzido de operadores e sua generalidade implicam em uma grande simplicidade (em termos de programação) na implementação de sistemas distribuídos.

O modelo de coordenação generativa [59] foi inicialmente proposto para interação de processos em aplicações paralelas nos anos 80. Porém, a partir de meados dos anos 90, ele têm sido aplicado no contexto de sistemas abertos devido a adequação de suas características a este tipo de sistema. Entretanto, nenhum dos trabalhos desenvolvidos apresenta preocupações em relação a requisitos de tolerância a faltas maliciosas e relacionadas ao mal funcionamento de componentes, comumente descritas como **faltas bizantinas** [79]. Este tipo de falta contempla tanto aspectos de tempo quanto de valor, e pode ser usado inclusive para modelar intrusões (ataques bem sucedidos) no sistema computacional [126].

Desta forma, um modelo de coordenação generativa tolerante a faltas bizantinas permite que participantes não confiáveis interajam de maneira desacoplada e confiável, podendo ser aplicado em um grande espectro de aplicações de grande relevância atualmente, oferecendo uma qualidade de serviço bastante útil na implementação de paradigmas modernos de segurança, como a tolerância a intrusões [57, 126].

[2] Neste documento é adotada a terminologia introduzida em [125]:

- A **falha** do sistema ocorre quando o serviço fornecido se desvia das condições mencionadas em sua especificação (descrição do serviço esperado);
- A falha ocorre porque o sistema estava incorrecto: um **erro** é a parte do estado do sistema que pode conduzir à falha;
- A causa de um erro - no seu sentido fenomenológico - é uma **falta**.

Assim, um processo em um sistema distribuído está sujeito a falhas que podem causar erros que levariam a falha do sistema como um todo. Isto implica no fato de que a falha do processo é considerada uma falta do ponto de vista do sistema. Desta forma, empregamos os termos "**tolerância a faltas**" (o sistema tolera fenômenos que o levariam a estados errados); e "**detecção de falhas**" (monitoração de componentes do sistema que podem falhar).

A figura 1.1 ilustra o funcionamento do modelo proposto neste trabalho considerando uma rede heterogênea, dinâmica e não confiável.

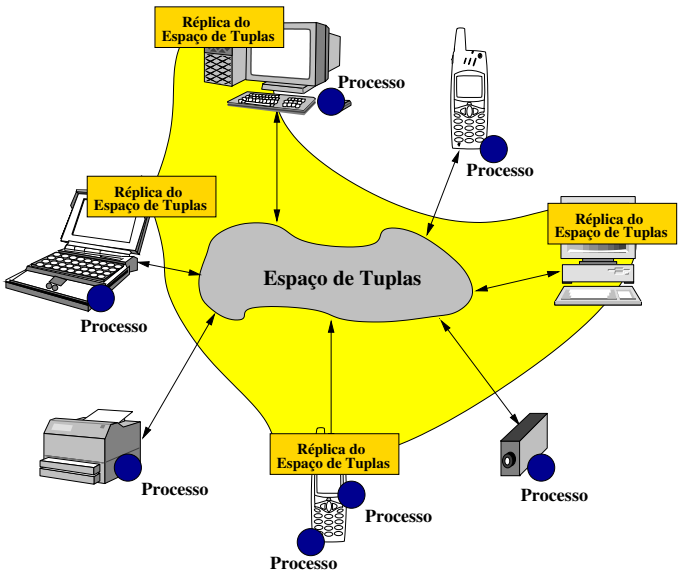


Figura 1.1: Interações via coordenação generativa.

Nesta figura, um conjunto de dispositivos executam aplicações distribuídas interligadas através de um espaço de tuplas confiável. Este espaço é composto por um conjunto de réplicas executando em alguns destes dispositivos. Os processos de aplicação executam tarefas que chamam operações do espaço de tuplas (*in*, *out* e *rd*).

Alguns cenários de interesse em que uma infra-estrutura de coordenação generativa tolerante a faltas bizantinas pode ser usada são:

- **Redes *ad hoc*:** As redes *ad hoc* (sem infra-estrutura pré-definida) têm recebido grande atenção da comunidade de pesquisa em redes de computadores. Entretanto, apenas recentemente aspectos relacionados ao *middleware* e as aplicações nestas redes têm sido explorados. Alguns trabalhos como o LIME [97] e o TOTA [89] propõem plataformas de *middleware* que suportam mobilidade baseados em espaço de tuplas, entretanto, nenhum destes suportes oferecem espaços confiáveis;

- **Computação móvel:** Em aplicações baseadas em agentes móveis, os processos agentes migram por vários nós de uma rede realizando computações. A interação entre estes agentes é sempre complexa devido a sua mobilidade. O uso do espaço de tuplas como meio de coordenação entre os agentes permite que estes se comuniquem sem se preocupar com sua localização nem seu estado de ativação (o agente pode receber mensagens enquanto estiver em processo de migração ou desativado temporariamente). Além disso, grande parte das aplicações de agentes são relacionadas a coleta e análise de informações e o próprio espaço de tuplas pode ser usado como um repositório para estas informações de tal forma a tirar proveito da regra de combinação de tuplas (memória associativa) [30]. A fragilidade em termos de segurança destes

ambientes móveis sugere que um paradigma de tolerância a faltas seja também suportado nesse modelo;

- **Computação em grade:** No modelo de computação em grade um número desconhecido e dinâmico de processos se coordenam para usar diferentes recursos compartilhados através do espaço de tuplas ou este mesmo espaço pode ser usado como repositório de informações e tarefas relacionadas à computações de larga escala distribuída. Dadas as questões de escala de uma grade computacional, nem todos os recursos ou usuários agregados ao sistema têm uma confiança justificada sobre eles, portanto, essa infra-estrutura de coordenação deve ser resistente a ataques e faltas maliciosas;

- **Serviços *web*:** A especificação *WS-Coordination* [74] define um conjunto de interfaces através das quais os serviços *web* podem requisitar interações através de mecanismos arbitrários de comunicação. Em aplicações em que a quantidade de participantes envolvidos em transações é maior que dois e estas entidades podem chegar em diferentes momentos, a coordenação generativa pode ser aplicada. Um exemplo de aplicação para esse modelo de interação é apresentado em [82].

Além destes cenários, novos modelos de computação como computação sensível a contexto e coordenação baseada em campos, entre outras, também pedem meios de comunicação mais desacoplados que o tradicional mecanismo de passagem de mensagens.

Em todos os cenários apresentados existe a possibilidade de processos maliciosos se envolverem nas computações, o que indica a necessidade de tolerância a faltas bizantinas para o modelo de coordenação generativa visando especificamente sistemas abertos.

1.2 Objetivos da Tese

O objetivo geral desta tese é propor um suporte de coordenação generativa que garanta segurança de funcionamento, mesmo com a possibilidade da ocorrência de falhas bizantinas nos processos do sistema, e estudar o que pode ser computado usando este suporte.

No que tange a definição do suporte de coordenação generativa, o objetivo geral da tese é apresentar uma arquitetura de segurança de funcionamento para espaços de tuplas, contemplando a provisão dos diversos atributos de segurança de funcionamento e a definição de diferentes algoritmos de replicação tolerantes a faltas bizantinas para concretização de diversos espaços com diferentes limitações e características.

Uma vez definido este espaço, estudamos o poder da abstração construída, analisando como este espaço pode ser usado na coordenação de processos sujeitos a falhas bizantinas. Este estudo é feito a partir da investigação de quais problemas de sistemas distribuídos podem ser resolvidos utilizando espaço de tuplas como um objeto de memória compartilhada acessada por processos não-confiáveis.

Baseado nestes objetivos mais gerais, são definidos uma série de objetivos específicos para o trabalho:

1. Definição do modelo de coordenação generativa dentro de um arcabouço teórico de sistemas distribuídos. Este modelo deve representar tanto as diferentes estratégias para implementação do espaço de tuplas quanto os processos de aplicação que interagem através do espaço;

2. Desenvolvimento de uma arquitetura para espaço de tuplas com segurança de funcionamento que suporte diversos mecanismos de tolerância a faltas (ex. replicação) e segurança (ex. controle de acesso);

3. Definição de algoritmos de replicação para espaços de tuplas tolerantes a faltas bizantinas baseados tanto no modelo de replicação Máquina de Estados [76, 114] quanto em sistemas de quóruns bizantinos [86];

4. Desenvolvimento de um mecanismo para confidencialidade em espaço de tuplas tolerantes a faltas bizantinas que mantenha as informações associadas as tuplas confidenciais, mesmo que alguns servidores falhem;

5. Investigação do poder do modelo de coordenação generativa como mecanismo de coordenação entre processos. Esta investigação se dá através do desenvolvimento de algoritmos para problemas fundamentais em sistemas distribuídos como consenso.

1.3 Organização do Texto

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

O capítulo 2 apresenta o modelo de sistema adotado nesta tese e o arcabouço geral para o espaço de tuplas tolerante a faltas bizantinas. Este capítulo também descreve os principais conceitos e ferramentas, tanto da literatura de algoritmos distribuídos quanto de criptografia, usados no desenvolvimento das abstrações apresentadas na tese. Dentre as ferramentas descritas neste capítulo, destacam-se o algoritmo PAXOS BIZANTINO e seu uso na resolução dos problemas de consenso, difusão com ordem total e exclusão mútua e as duas principais abordagens para replicação tolerante a faltas bizantinas: modelo Máquina de Estados e sistemas de quóruns bizantinos.

No capítulo 3 são descritos detalhadamente o conceito de coordenação e o modelo de coordenação generativa. Este segundo tópico, fundamental no desenvolvimento desta tese, é formalizado e comparado com outros modelos de coordenação. Além disso, algumas extensões que incrementam o poder do espaço de tuplas são descritas. Na segunda parte deste capítulo, é introduzido o conceito de espaço de tuplas com segurança de funcionamento e uma arquitetura para concretização deste tipo de espaço é proposta. Esta arquitetura se baseia na integração da replicação Máquina de Estados com

um conjunto de mecanismos de segurança. Esta arquitetura, juntamente com o mecanismo de confidencialidade definido no capítulo 5, serve de base para construção do espaço de tuplas mais poderoso, do ponto de vista da segurança de funcionamento, apresentado nesta tese.

Enquanto no capítulo 3 é descrita a construção de espaço de tuplas tolerante a faltas bizantinas usando replicação Máquina de Estados, no capítulo 4 investigamos a possibilidade de concretização deste modelo usando sistemas de quóruns bizantinos. Neste capítulo é discutido o porque dos sistemas de quóruns sozinhos não serem capazes de implementar um espaço de tuplas e a partir desta constatação, introduzimos (informalmente) a abordagem de sistemas de quóruns ativos. Esta abordagem consiste em executar protocolos de quóruns para operações de leitura e operações escrita (que são normalmente implementadas por sistemas de quóruns) e algoritmos de acordo (exclusão mútua, consenso e difusão com ordem total) na implementação de operações de leitura-escrita. A partir daí, duas construções para espaços de tuplas com diferentes características são apresentadas e comparadas com aquela baseada em replicação Máquina de Estados (descrita no capítulo 3).

O capítulo 5 apresenta uma solução para a manutenção da confidencialidade de tuplas no espaço replicado mesmo na presença de alguns servidores maliciosos. Esta solução se baseia no uso de um tipo específico de esquema de compartilhamento de segredo verificável, onde os servidores armazenam fragmentos da tupla e a mesma só pode ser recuperada se um número suficiente de servidores se engajarem no protocolo de leitura.

O estudo do impacto do uso de espaços de tuplas como memória compartilhada por processos sujeitos a falhas bizantinas é apresentado no capítulo 6. Neste capítulo introduzimos o conceito de objetos de memória compartilhada protegidos por políticas e apresentamos diversos algoritmos eficientes e elegantes para resolução de diversas variantes de consenso e emulação de objetos (construção universal) baseadas em um espaço de tuplas aumentado e protegido por políticas. Os resultados apresentados neste capítulo mostram que a abordagem introduzida nesta tese apresenta enormes benefícios em termos de simplicidade e desempenho quando comparada a abordagem previamente usada na literatura, em que os objetos são protegidos por listas de controle de acesso.

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

Capítulo 2

Conceitos Básicos

Este capítulo apresenta os conceitos básicos de sistemas distribuídos que serão utilizados no decorrer de todo o texto. Em especial, são apresentadas as diversas ferramentas utilizadas no seu desenvolvimento bem como o modelo de sistema assumido nas várias implementações do espaço de tuplas tolerante a faltas bizantinas (capítulos 3 e 4). Este capítulo também apresenta uma breve descrição das duas principais técnicas utilizadas para implementação de sistemas tolerantes a faltas bizantinas: replicação Máquina de Estados e sistemas de quóruns bizantinos.

2.1 Modelo de Sistema

Modelos de sistema definem as premissas sobre as quais algoritmos e sistemas são projetados; sendo portanto, fundamentais para o entendimento das características e limitações de qualquer solução. Quando falamos de modelos de sistemas distribuídos, consideramos uma série de "sub-modelos" que definem cada um dos aspectos particulares deste tipo de sistema. Nesta seção definimos as premissas fundamentais, para cada um destes "sub-modelos", que são assumidas no decorrer deste trabalho.

2.1.1 Processos

Assume-se um conjunto infinito de processos no sistema dividido em dois subconjuntos. Um contendo um número ilimitado de clientes $\Pi = \{p_1, p_2, ...\}$ e outro contendo n servidores $U = \{s_1, ..., s_n\}$. Cada um destes processos executam um ou mais algoritmos.

2.1.2 Modelo de Falhas nos Processos

Todos os processos do sistema estão sujeitos a **falhas bizantinas**[1] [79]. Um processo que apresenta este tipo de falha pode exibir qualquer comportamento, podendo parar, omitir o envio e o rece-

[1] Este tipo de falha também é conhecida como **maliciosa** ou **arbitrária**.

bimento de algumas mensagens, enviar mensagens inesperadas e/ou incorretas ou mesmo mudar de estado arbitrariamente. Um processo que apresenta comportamento de falha é dito **falho**, de outra forma é dito **correto**.

As falhas bizantinas podem ser de natureza **acidental** (de projeto, defeito em algum componente do sistema, etc.) ou **maliciosa** (ataques bem sucedidos que levam a intrusões [126]). Neste trabalho assumimos que os dois tipos de falhas podem acontecer em diferentes processos de forma independente, i.e. a probabilidade de um processo sofrer uma falha é independente da probabilidade de outro processo sofrer uma falha. Esta propriedade, chamada **independência de falhas**, pode ser substanciada em sistemas reais através do uso agressivo de diversidade [49, 81], conforme discutido em [37, 99].

A independência de falhas é uma premissa importante no desenvolvimento de algoritmos tolerantes a faltas bizantinas, pois elas permitem raciocinar de forma simples sobre o comportamento do sistema em caso de falhas parciais. Não obstante, trabalhos recentes demonstram que algoritmos fundamentados nesta premissa podem ser convertidos para modelos mais sofisticados, que consideram falhas dependentes, como estruturas adversárias [31], sistemas sujeitos a falha [86] ou núcleos e conjuntos sobreviventes [72].

2.1.3 Modelo de Sincronismo

Existem vários modelos de sincronismo para sistemas distribuídos [9, 66, 83]. Desde os sistemas em que a noção de tempo é completamente inexistente (ex. [56]) até sistemas completamente dependentes do tempo, em que tanto as computações locais quanto as comunicações entre os processos têm limites de tempo fixos e conhecidos (ex. [72]).

Neste trabalho utilizamos o **modelo de sistema com sincronia terminal** (*eventually synchronous system model*) [54]. Este modelo estipula que em todas as execuções do sistema existe um limite Δ e um instante de tempo GST (*Global Stabilization Time*) de tal forma que toda mensagem enviada por um processo correto após um instante $u >$ 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 nem tampouco eles precisam ser os mesmos em diferentes execuções do sistema. A intuição por trás deste modelo é de que o sistema trabalha de forma assíncrona (não respeitando nenhum limite de tempo) a maior parte do tempo porém, durante períodos de estabilidade, o tempo para transmissão de mensagens é limitado[2]. Esta premissa garante que um algoritmo baseado em *rounds* que só termina quando um *round* for executado de forma síncrona (dentro de algum limite de tempo), acaba por terminar.

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

[2]Na prática, estes períodos devem ser longos o suficiente para que o algoritmo distribuído termine.

Estas premissas foram escolhidas para serem usadas neste trabalho pela necessidade do uso de um algoritmo específico de consenso para a implementação de algumas operações do espaço de tuplas e pela sua viabilidade prática. Estas premissas são suficientes para garantir a terminação deste tipo de algoritmo 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).

Nos próximos capítulos o conceito de **execução favorável** será utilizado para definir determinadas condições em que os algoritmos terminam de forma mais eficiente. Uma execução é dita favorável se o limite Δ é válido desde seu início (GST $= 0$) e não ocorrem falhas nos processos.

2.1.4 Canais de Comunicação

Assumimos que os processos do sistema se comunicam através de canais ponto a ponto que podem perder, duplicar ou alterar mensagens, entretanto, estes canais satisfazem a seguinte propriedade:

- **Limitação de Perda Forte**: Se uma mensagem m é enviada infinitas vezes, então m é recebida infinitas vezes.

Canais que satisfazem esta propriedade são chamados canais justos (*fair links*) [83]. Utilizando estes canais (e algumas ferramentas criptográficas) podemos implementar primitivas de envio e recepção de mensagens, denotadas por $send(p,m)$ (envio de m ao processo p) e $receive(p,m)$ (recebimento de m enviada pelo processo p), respectivamente, que fornecem as seguintes propriedades:

- **Integridade**: Se um processo q recebe uma mensagem m de um processo p, então p enviou m a q;
- **Confiabilidade**: Se um processo p envia uma mensagem m a outro processo q, então q termina por receber m;
- **Ordenação FIFO (*First-In First-Out*)**: Se um processo envia uma mensagem m a um processo p antes de enviar uma mensagem m' a este mesmo processo, então p recebe m antes de m'.

Canais que satisfazem estas propriedades são ditos **confiáveis autenticados com ordenação FIFO**. Uma implementação que satisfaz estas propriedades pode se basear nas seguintes estratégias:

1. Usando chaves simétricas de sessão [23] ou algum esquema de assinatura de chave pública (ver Seção 2.2), os processos comunicantes assinam digitalmente cada uma de suas mensagens enviadas de tal forma a garantir que *(i.)* a corrupção do conteúdo de suas mensagens possa ser facilmente detectada e *(ii.)* a autoria de uma mensagem possa ser verificada;
2. Toda mensagem m a ser enviada pelo processo p ao processo q é retransmitida até que p receba uma confirmação de recebimento de q. Esta confirmação pode vir de "carona" em alguma mensagem enviada de q para p (ex. usando um campo adicional nas mensagens contendo o identificador das mensagens sendo confirmadas);

3. Toda mensagem m tem um número de seqüência $seq(m)$ associado a ela. Um processo receptor só entrega uma mensagem m se já tiver entregado todas as mensagens com número de seqüência menor que $seq(m)$. Note que esta estratégia implica em cada processo tendo que guardar o número de seqüência da última mensagem entregue de cada outro processo com que ele interage.

É fácil ver que estas três estratégias combinadas satisfazem as três propriedades do canal confiável autenticado com ordenação FIFO apresentado acima. Os algoritmos apresentados nesta tese usam este tipo de canal.

Como uma última nota a respeito dos canais de comunicação usados nos algoritmos, vale lembrar que eles podem facilmente ser implementados na prática através do uso de SSL/TLS [50] sobre TCP ou TCP sobre IPSEC [102].

2.1.5 Formalizando o Espaço de Tuplas Tolerante a Faltas Bizantinas

O formalismo adotado para descrever os espaços de tuplas desenvolvidos nesta tese é o **Autômato de I/O** [83, 84]. A principal motivação para a escolha deste modelo é a generalidade das entidades (autômatos de I/O) que podem ser usadas para representar qualquer componente de um sistema distribuído, incluindo processos, canais e recursos compartilhados. Esta generalidade tornou este formalismo um dos mais usados na descrição e verificação de algoritmos distribuídos complexos (ex. [3, 36, 67, 83]).

Os sistemas de coordenação generativa tolerantes a faltas bizantinas propostos neste trabalho podem ser descritos a partir da composição de uma série de autômatos, que correspondem aos diversos tipos de algoritmos próprios a este modelo. O primeiro passo nesse sentido é a definição de um autômato que represente o espaço de tuplas através do qual os processos de aplicação interagem, de forma semelhante a um sistema distribuído com memória compartilhada. Em seguida, é preciso definir os autômatos dos processos que interagem utilizando o espaço de tuplas como meio de coordenação.

Assim, são definidos o autômato correspondente ao espaço de tuplas e m autômatos para os processos de aplicação. O autômato correspondente ao espaço, por sua vez, é definido a partir da composição de $m+n$ outros autômatos, sendo m o número de processos que interagem com o mesmo e seus respectivos autômatos clientes no espaço de tuplas (*stubs*) e considerando n réplicas deste espaço. A figura 2.1 apresenta o modelo conceitual do sistema.

Na figura 2.1, o espaço de tuplas TS é composto por autômatos clientes C_i (*stubs* associados a cada processo conectado ao espaço) e réplicas R_i. A cada cliente C_i do espaço também está associado um autômato P_i que representa o processo de aplicação que está se coordenando através do espaço de tuplas. Cada um destes "sub-autômatos" são responsáveis pela implementação de diferentes algoritmos:

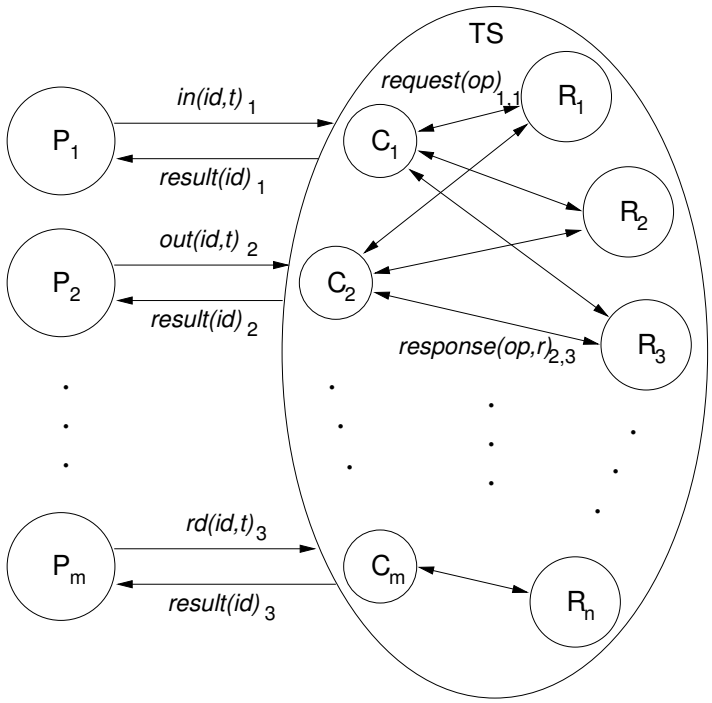


Figura 2.1: Modelo conceitual para o sistema baseado em coordenação generativa.

- **Processos (P_i):** São os processos de aplicação. Estes processos se coordenam através do espaço de tuplas TS. Os autômatos correspondentes aos processos têm em sua assinatura as ações de saída relativas às operações no espaço de tuplas que representam requisição de operações no espaço (*in*, *rd*, *out*, etc...). A resposta a estas operações é representada pelas ações $result(id)_i$ onde *id* é o identificador único para cada operação requisitada. Os autômatos processos vêem o espaço de tuplas como um objeto compartilhado e eles podem implementar qualquer aplicação baseada no modelo de coordenação generativa;

- **Clientes (C_i):** Os clientes são os componentes do sistema responsáveis por receber as requisições dos processos e, através da parte cliente do protocolo de replicação definido, invocar a operação requerida nas réplicas do espaço de tuplas de forma a manter a consistência deste objeto. Estes autômatos podem ser vistos como *stubs* do espaço de tuplas e toda interação entre os processos e este espaço ocorre através deles;

- **Réplicas (R_i):** As réplicas implementam a parte servidor do protocolo de replicação do espaço de tuplas. Cabe a cada autômato réplica também implementar o comportamento de um espaço de tuplas não replicado.

Com relação ao mapeamento desses diferentes autômatos nos componentes do sistema, consideramos que cada servidor $s_i \in U$ executa o algoritmo representado pelo autômato R_i, e que cada processo $p_i \in \Pi$ executa os autômatos P_i e C_i. Desta forma, assumimos que se um processo p_i é falho, ambos os autômatos P_i e C_i falham.

Assumimos que o modelo descrito acima satisfaz a condição de **boa formação** [83] (também chamada **interação correta** [9]): as interações entre os autômatos ocorrem uma de cada vez, i.e. se

um autômato A gera um evento de saída para outro autômato B (ex. uma invocação), A só vai gerar um novo evento deste tipo após receber um evento de entrada produzido por B (ex. uma resposta). Em nosso modelo, esta condição é observada mesmo por processos falhos. Esta condição serve, principalmente, para simplificar a apresentação dos algoritmos correspondentes aos autômatos, já que com ela não é necessário considerar invocações concorrentes. Na prática, esta premissa pode ser facilmente implementada fazendo com que os autômatos corretos ignorem eventos (invocações) realizadas por processos que têm invocações pendentes.

O modelo da figura 2.1 pode ser considerado como o arcabouço geral onde as diversas contribuições desta tese podem ser agrupadas. Algoritmos correspondentes aos autômatos C_i e R_i, que descrevem o algoritmo de replicação do espaço de tuplas são apresentados nos capítulos 3 e 4, enquanto o capítulo 6 descreve a implementação dos autômatos P_i. Na representação destes autômatos não utilizamos o modelo de pré-condição/efeito normalmente utilizado para escrita de autômatos de I/O [83] devido a sua pouca legibilidade quando se trata de algoritmos complexos. Ao invés disso, os representamos através de algoritmos formais equivalentes aos autômatos de I/O, sendo porém, descritos de forma muito mais intuitiva.

Duas premissas importantes do modelo da figura 2.1 merecem ainda alguma explicação: *(i.)* as interações (mesmo as remotas, entre processos clientes e servidores) são representadas por eventos de entrada e saída; e *(ii.)* o modelo de autômato de I/O é completamente assíncrono. A premissa *(i.)* se justifica pelo fato de que os canais assíncronos autenticados e confiáveis usados no algoritmos podem ser representados com bastante exatidão pelas transições assíncronas provocadas por eventos de entrada e saída dos autômatos de I/O. Esta premissa permite um modelo mais legível. A premissa *(ii.)* também objetiva legibilidade. Conforme apresentado na seção 2.1.3, assumimos um modelo de sistema parcialmente síncrono. Este tipo de sistema não pode ser representado de forma exata usando os autômatos de I/O tradicionais, e requer o uso de algumas extensões que permitem o uso da variável tempo no modelo [83]. Tendo em vista que nenhum dos algoritmos propostos nesta tese consideram tempo explicitamente, apenas requerendo tais premissas na concretização de primitivas subjacentes[3], optou-se por não se usar essas extensões.

2.2 Ferramentas Criptográficas

Nesta seção apresentamos as principais ferramentas criptográficas utilizadas pelos algoritmos desenvolvidos nesta tese. As primitivas criptográficas específicas para o mecanismo de confidencialidade serão apresentadas no capítulo que trata deste mecanismo.

2.2.1 Resumo Criptográfico

Uma **função criptográfica de resumo resistente a colisões** H é uma função que mapeia uma entrada de tamanho arbitrário em uma saída de tamanho fixo. Esta função deve satisfazer três propri-

[3] As premissas de tempo são utilizadas apenas para garantir a terminação dos algoritmos de acordo empregados na construção do espaço de tuplas com segurança de funcionamento.

edades básicas:

1. **Eficiência**: para qualquer v, é fácil computar $\mathsf{H}(v)$;
2. **Resistência a colisão**: é computacionalmente inviável[4] obter dois valores $v \neq v'$ de tal forma que $\mathsf{H}(v) = \mathsf{H}(v')$;
3. **Unidirecionalidade**: dada uma saída da aplicação da função de resumo, é computacionalmente inviável encontrar a entrada que produz essa saída.

Durante o texto, o resultado da aplicação da função de resumo em um dado será chamado de resumo do valor. Um exemplo de uma função de resumo com essas características é a função SHA-1, que gera uma saída de 160 bits [98].

2.2.2 Assinatura Digital

Um esquema de assinatura digital é usualmente definido sobre um sistema de criptografia assimétrica envolvendo pares de chaves pública e privada $\langle pk, sk \rangle$, usados para cifragem e decifragem de dados, respectivamente. Nesta tese, assumimos que cada servidor correto tem uma chave privada que é conhecida apenas por ele. As chaves públicas de todos os servidores do sistema são conhecidas por todos os outros processos. Um esquema de assinaturas digital utiliza um algoritmo de assinatura e um de verificação baseados em criptografia assimétrica para garantir que uma determinada unidade de informação foi mesmo assinada por um processo. A função de assinatura $\mathsf{sign}(sk, v)$ usa a chave privada sk, um valor de tamanho arbitrário v e produz uma assinatura σ. A função de verificação $\mathsf{vsign}(pk, v, \sigma)$ produz uma saída lógica (*true* ou *false*) que indica se a assinatura σ é valida, i.e., se ela foi obtida a partir da aplicação da função assinatura com os argumentos sk e v. A principal propriedade deste esquema é que é computacionalmente inviável obter uma assinatura válida σ sem conhecer a chave privada sk (**não forjável**). Neste trabalho, denotamos uma mensagem m assinada pelo processo p por m_{σ_p}. Em todos os algoritmos em que processos esperam por mensagens assinadas assumimos que as mensagens recebidas são sempre verificadas e que mensagens com assinaturas inválidas são descartadas. Um exemplo de algoritmo de criptografia assimétrica usado para assinatura é o RSA [109].

2.2.3 Código de Autenticação de Mensagens

Os códigos de autenticação de mensagens (MAC - *Message Autentication Codes*) [23] são utilizados para autenticar mensagens trocadas entre partes que compartilham uma chave secreta. Neste esquema um MAC μ é produzido através da aplicação da função $\mathsf{mac}(k, m)$ na mensagem m utilizando a chave k. Tendo sido computado o MAC, o processo emissor envia m juntamente com μ aos destinatários. Qualquer outro processo que conheça a chave k pode verificar a autenticidade e integridade de m comparando o valor de $\mathsf{mac}(k, m)$ com μ.

[4] Não existe um algoritmo que possa executar esta tarefa de forma eficiente [63].

Nesta tese assumimos que as mensagens são autenticadas através de um tipo especial de MAC, os HMACs (*Hashed MAC*) [123]. Este tipo de código é produzido através da computação de um resumo da mensagem juntamente com a chave, i.e. $\mathsf{mac}(k,m) \triangleq \mathsf{H}(m \otimes k)$. Onde '$\otimes$' denota algum tipo de computação que combine m e k. Este tipo de primitiva é interessante pois permite a computação de MACs de forma muito eficiente.

2.3 Problemas de Acordo

Um conceito fundamental em sistemas distribuídos é o acordo entre processos. Grande parte das computações distribuídas envolve algum tipo de acordo, desde o envio de uma mensagem entre dois processos (em que o emissor e o receptor concordam sobre qual é a mensagem transmitida) até uma transação distribuída envolvendo vários servidores (em que o processamento executado só é confirmado quando todos executam sua parte do trabalho).

Existem vários tipos de problemas de acordo em sistemas distribuídos, a maioria destes se baseia no seguinte padrão: todos os processos envolvidos devem chegar a uma decisão comum, esta decisão depende da natureza do problema. Nesta seção apresentamos três problemas fundamentais de acordo em sistemas distribuídos.

2.3.1 Consenso

O **consenso** [104] é o problema mais estudado entre os problemas de acordo em sistemas distribuídos. Isto se deve ao fato deste problema ser considerado a forma canônica de acordo, e portanto ter importância crucial tanto teórica (definindo limites para os sistemas computacionais em que o acordo pode ser implementado) quanto prática (formando a base algorítmica para quase todos os problemas de acordo).

O problema de consenso consiste em um conjunto de processos, em que cada processo p_i propõe um valor $v_i \in \mathscr{V}$ e ao final do processamento, os processos decidem de maneira unânime um valor relacionado aos valores propostos[5]. Formalmente o problema é definido por duas primitivas:

- $propose(G,v)$: o valor v é proposto ao grupo de processos G;
- $decide(v)$: chamado pelo protocolo de consenso para informar à aplicação que o valor v é o valor decidido.

Estas primitivas devem satisfazer as seguintes propriedades [7, 9, 83][6]:

[5] Quando $\mathscr{V} = \{0,1\}$ temos o **consenso binário**, para domínios de valores maiores, o problema é geralmente chamado **consenso multivalorado**.

[6] Existem várias definições diferentes para o problema de consenso [12, 32, 38, 56, 66] (entre outras), mas a semântica final é sempre parecida.

1. **Acordo:** Se um processo correto decide v, então todos os processos corretos terminam por decidir v;
2. **Validade:** Se um processo correto decide v, então v foi proposto por algum processo;
3. **Terminação:** Todos os processos corretos terminam por decidir.

A propriedade de acordo captura a essência do problema: todos os processo corretos decidem um mesmo valor. A validade relaciona o valor decidido com os valores propostos e sua alteração dá origem a outros tipos de consenso. Já a propriedade de terminação define o requisito de vivacidade do problema.

Em uma parcela substancial dos modelos de computação distribuída sujeitos a faltas bizantinas, não é suficiente implementar consenso respeitando a condição de validade descrita acima, uma vez que através dela é possível que sejam decididos valores propostos por processos falhos. Nestes ambientes, uma condição de validade diferente chamada **Não trivialidade** é muitas vezes satisfeita [41, 83]:

1. **Não trivialidade:** Se todos os processos corretos propõem um valor inicial v, então o valor decidido é v.

O nome desta condição se deve ao fato de que ela impede a implementação de algoritmos de consenso triviais, que decidem sempre o mesmo valor independentemente das propostas dos processos [83]. Alguns autores argumentam que esta definição de validade não tem utilidade prática, uma vez que segundo ela, quando os processos propõem valores diferentes, o valor decidido não precisa ter ligação nenhuma com as propostas [7, 12]. A condição de consenso ideal para cenários com faltas bizantinas é a **Validade Forte** [85], que garante que o valor decidido foi proposto por algum processo correto:

1. **Validade Forte:** Se um processo correto decide v, então v foi proposto por algum processo correto.

O problema com esta condição de validade reside no fato de que é geralmente impossível distinguir um processo falho (agindo como um correto, no que diz respeito ao protocolo) de um processo correto. Desta forma, este tipo de validade só pode ser satisfeita se utilizamos um número de processos diretamente proporcional ao tamanho do domínio de valores considerado ($|\mathscr{V}|$), conforme será visto no capítulo 6.

PAXOS BIZANTINO

O protocolo de consenso utilizado nesta tese para a construção das diferentes implementações do espaço de tuplas tolerantes a faltas bizantinas é o **PAXOS BIZANTINO** [36] (doravante chamado apenas de PAXOS), acrescido de algumas modificações para terminação rápida [91, 130]. Este protocolo

foi escolhido devido ao seu bom desempenho em execuções favoráveis e livres de falha (dois passos de comunicação) e à sua resistência ótima ($n \geq 3f+1$ processos são requeridos para tolerar até f falhas bizantinas simultâneas).

O algoritmo original PAXOS (para faltas de parada) considera três classes/papéis de agentes: **proponentes**, os quais propõem os valores; **aceitantes**, os quais escolhem um único valor entre os propostos; **aprendizes**, os quais precisam aprender o valor decidido [77, 78]. Os conjuntos de processos que representam estes papéis são P, A e L, respectivamente. O algoritmo garante que um único valor proposto por um dos proponentes é escolhido pelos aceitantes e aprendido por todos os aprendizes. Na maior parte das implementações deste algoritmo, todos os processos do sistema desempenham estes três papéis ao mesmo tempo ($P = A = L$). Apesar da implementação original do PAXOS para faltas bizantinas não fazer a diferenciação entre estes três papéis [36], versões mais modernas deste algoritmo consideram esta diferenciação [91]. Para que o algoritmo implemente consenso, deve haver pelo menos um proponente correto para que garantidamente haja proposta, i.e. $|P| \geq f+1$, e, como são os aceitantes que estabelecem o acordo sobre o valor, o número de processos que representam este papel não deve ser menor que $3f+1$, i.e. $|A| \geq 3f+1$ [91].

Este algoritmo é executado em *rounds*, sendo que, em cada *round r*, um proponente p_r é escolhido como 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 o acordo. Por fim, quando estabelecida, a decisão de consenso é enviada aos aprendizes. As propriedades de segurança (*safety*) sempre são mantidas pelo protocolo, mas a vivacidade só é satisfeita em *rounds* favoráveis. Um *round* é considerado **favorável** quando seu líder é correto (cada *round* tem apenas um líder, o processo com identificador r **mod** n) e o sistema está num período de sincronia: as comunicações e computações ocorrem dentro de um período de tempo limitado. Nesta situação, um valor proposto pode ser aprendido dentro do período de um *round*. Adicionalmente, um *round* é dito **muito favorável** se ele é favorável e não existem falhas nos aceitantes. 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.

Cada *round r* do PAXOS compreende as seguintes fases[7] [91]:

1. O proponente p_r é o líder do *round r*, e conseqüentemente é o processo responsável por enviar aos clientes sua proposta v_r em uma mensagem PROPOSE;

2. Um aceitante que recebe uma proposta em r a **aceita** se ela foi proposta por p_r e se ela é justificada, i.e. não existe nenhum processo correto que decidiu um valor diferente de v_r (para detalhes, ver [91, 130]). Um aceitante que aceita um proposta v_r difunde uma mensagem ACCEPT contendo v_r aos demais aceitantes. Nesta fase, v_r é dito fracamente aceito;

3. Quando um aceitante recebe $\lceil \frac{n+f}{2} \rceil$ mensagens ACCEPT contendo v_r ele marca este valor como fortemente aceito. Um valor fortemente aceito será o valor decidido se o líder for correto. A partir daí, uma mensagem DECIDE com v_r é enviada a todos os aceitantes;

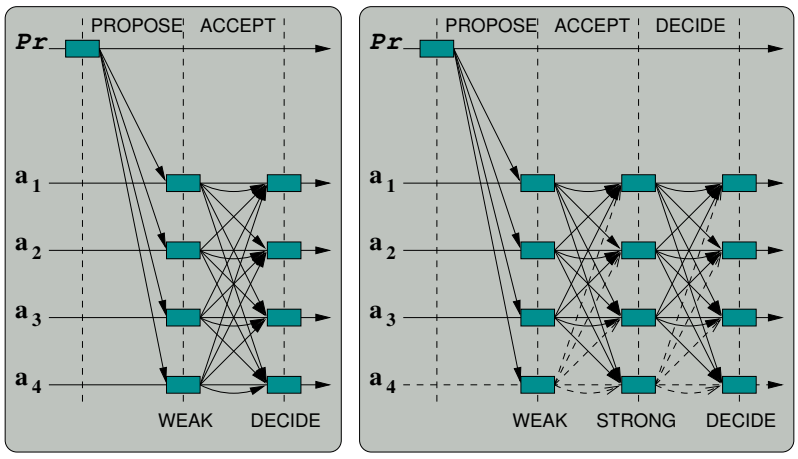
[7] A versão do PAXOS BIZANTINO descrita em [36] é ligeiramente diferentes desta já que define um mecanismo de ordenação total ao invés de um algoritmo de consenso.

- **Terminação rápida:** Se um aceitante recebe $\lceil \frac{n+3f}{2} \rceil$ mensagens ACCEPT para um mesmo valor v_r, este valor é decidido [91, 130].

4. Se um aceitante recebe $\lceil \frac{n+f}{2} \rceil$ mensagens DECIDE com v_r, então este é o valor decidido nesta execução do consenso. Esta fase de decisão objetiva garantir que se um processo correto decide v_r, então todos os outros processos corretos terminarão por decidir v_r, mesmo que o líder seja falho e um novo *round* seja iniciado.

Quando $A \neq L$, as mensagens ACCEPT e DECIDE são enviadas para todos os processos em $A \cup L$, e os processos aprendizes utilizam estas mensagens apenas para descobrir o valor aprendido. Isto significa que os processos que participam do algoritmo apenas como aprendizes são passivos, e não influem no funcionamento deste.

As figuras 2.2(a) e 2.2(b) ilustram alguns cenários de execução do PAXOS BIZANTINO[8]. A figura 2.2(a) mostra uma execução em que o protocolo executa um *round* muito favorável e consegue terminar em apenas dois passos de comunicação (terminação rápida)[9]. O caso normal de operação do PAXOS, onde o *round* é apenas favorável, é apresentado na figura 2.2(b).



(a) Muito favorável. (b) Favorável com a_4 falho.

Figura 2.2: Execuções do PAXOS BIZANTINO.

Caso um *round* não seja completado em um determinado intervalo de tempo, um novo *round* é iniciado através de um protocolo de transição, que requer dois passos de comunicação. Este protocolo permite a eleição de um novo líder e o início do *round* seguinte ao mesmo tempo que garante que este novo líder verifique se algum valor proposto em *rounds* anteriores foi aceito fracamente, fortemente, ou decidido pelos aceitantes, definindo sua proposta a partir desta informação.

Conforme apresentado, o algoritmo PAXOS sempre garante as propriedades de segurança, i.e. se um processo correto decide um valor v, todos os processos corretos decidem v. Além disso, o

[8] O padrão de comunicação apresentado nestas figuras não considera o fato de que as trocas de mensagens não são sincronizadas. Esta simplificação tem como objetivo melhorar a visualização de diagramas espaço $\times$ tempo como o apresentado nesta figura.

[9] Note que, como os próprios aceitantes são os aprendizes, não é necessário difundir as mensagens ACCEPT e DECIDE para estes últimos.

algoritmo satisfaz a condição de validade convencional: o valor decidido foi proposto por algum processo (correto ou falho). Em relação a terminação, no modelo de sistema adotado neste trabalho (sincronia terminal) é garantida a existência de um *round* favorável, e portanto toda execução do PAXOS termina.

2.3.2 Difusão com Ordem Total

O problema da **difusão com ordem total** (ou **difusão atômica**) [66, 76] está relacionado à provisão de confiabilidade na disseminação de uma mensagem (ou informação) para um conjunto de processos. Adicionalmente, um algoritmo para difusão com ordem total deve garantir que as mensagens difundidas sejam entregues aos membros corretos do grupo de receptores em uma mesma ordem total.

A difusão com ordem total é definida sobre duas primitivas básicas:

- *TO-multicast*(G,m): A mensagem m é difundida para todos os processos pertencentes ao grupo G.
- *TO-deliver*(p,m): Chamado pelo protocolo de difusão com ordem total para entregar à aplicação uma mensagem m difundida por p em G.

Formalmente, uma difusão com ordem total é uma primitiva de difusão que satisfaz as seguintes propriedades [66]:

1. **Validade:** Se um processo correto difundiu m em G, então algum processo correto pertencente à G terminará por entregar m ou nenhum processo do grupo está correto;
2. **Acordo:** Se um processo correto pertencente a G entrega a mensagem m, então todos os processos corretos pertencentes a G acabarão por entregar m;
3. **Integridade:** Para qualquer mensagem m, cada processo correto pertencente a G entrega m no máximo uma vez e somente se m foi previamente difundida em G;
4. **Ordenação Total Local:** Se dois processos corretos p e q entregam as mensagens m e m' difundidas em G, então p entrega m antes de m' se e somente se q entregar m antes de m'.

Um resultado teórico interessante é que a difusão com ordem total e o consenso são problemas equivalentes tanto em sistemas com processos sujeitos a falhas de parada [38] quanto em sistemas em que faltas bizantinas são consideradas [43].

Difusão com Ordem Total usando o PAXOS BIZANTINO

A implementação de difusão com ordem total usando o PAXOS BIZANTINO 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 o resultado da i-ésima execução do protocolo PAXOS BIZANTINO corresponde a mensagem m [36, 91].

Quando consideramos a versão do PAXOS para faltas de parada, a implementação de difusão com ordem total é feita de forma direta, usando o esquema descrito acima [78]. No entanto, quando consideramos faltas bizantinas, existem muitos detalhes que devem ser tratados.

Na implementação da difusão com ordem total baseada no PAXOS consideramos um conjunto de servidores U onde todos implementam os três papéis do PAXOS ($U = P = A = L$), e um conjunto ilimitado de clientes. O algoritmo funciona da seguinte forma [36]: para difundir uma mensagem m com ordem total, um cliente deve primeiro enviar m assinada a todos os servidores. O líder desses servidores, ao receber m, inicia a execução de número i do PAXOS propondo $\mathsf{H}(m) = d$ como o valor de acordo. Ao final da execução do algoritmo de acordo, todos os processos saberão que a mensagem cujo resumo criptográfico corresponde a d é a i-ésima mensagem a ser entregue. Um servidor entrega a mensagem m quando *(i.)* esta mensagem é recebida, *(ii.)* o valor da instância i do PAXOS é $\mathsf{H}(m)$ e *(iii.)* todas as mensagens decididas por instâncias $j < i$ do algoritmo já foram entregues.

A implementação deste algoritmo requer algumas alterações no algoritmo básico de consenso PAXOS. Primeiramente, um valor proposto d pelo líder só é **aceito** por um servidor se, além dos critérios de aceite definidos para o consenso (ver seção 2.3.1), o número de instância i do algoritmo está dentro de uma janela de valores de seqüência préviamente definida e o servidor recebeu a mensagem cujo resumo é d (que tem de ser m, graças a propriedade de resistência a colisão do resumo criptográfico). Estas regras extras objetivam impedir que um líder malicioso invente mensagens e defina números de seqüência muito elevados para mensagens recebidas (criando lacunas na seqüência de mensagens a ser entregue). Além disso, a cada troca de líder os demais servidores devem enviar seu estado em relação às mensagens já ordenadas e às lacunas existentes na seqüência de mensagens entregues. De posse destas informações, um novo líder é capaz de definir mensagens *nop* (que não alteram o estado das réplicas) para instâncias já iniciadas do algoritmo que tenham como proposta mensagens inválidas.

A figura 2.3 ilustra a difusão com ordem total de uma requisição usando este algoritmo em uma execução muito favorável (PAXOS termina em 2 passos de comunicação). Nesta figura a mensagem m é entregue com número de seqüência i.

Uma importante otimização que pode ser feita no algoritmo de ordenação PAXOS é a **ordenação em lote**: ao invés de executar uma instância do algoritmo de consenso para cada mensagem recebida, o algoritmo é executado para grupos de mensagens. Esta otimização tende a melhorar a escalabilidade do algoritmo, permitindo um bom desempenho mesmo em execuções com grande carga [22].

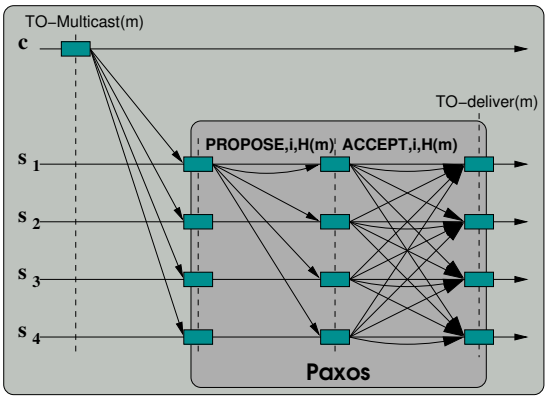


Figura 2.3: Difusão com ordem total usando o PAXOS BIZANTINO.

2.3.3 Exclusão Mútua

O problema da **exclusão mútua** [52] consiste em gerenciar um conjunto de processos que desejam acessar um recurso indivisível que não pode sofrer acessos simultaneamente. Um algoritmo que resolve este problema deve garantir que processos nestas condições tenham acesso ao recurso um por vez. Os processos obtém acesso ao recurso exclusivo executando uma seção **crítica** de seu código. Os processos que estão tentando executar suas seções críticas são ditos na seção de **entrada** e os processos que já executaram suas seções críticas são ditos na seção de **saída**.

Um algoritmo de exclusão mútua é muitas vezes modelado como um serviço onde os processos executam as operações $enter()$, para tentar obter o acesso ao recurso (seção de entrada), e $exit()$, para liberar o recurso (seção de saída). As operações $enter()$ e $exit()$ só retornam quando o processo é bem sucedido na obtenção e liberação do recurso, respectivamente.

Um algoritmo de exclusão mútua precisa satisfazer algumas propriedades [5, 83]. Uma destas propriedades fundamenta a segurança (*safety*) do algoritmo:

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

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

1. **Progresso (*Deadlock-freedom*):** Se um processo correto está na seção de entrada (executando $enter()$), então **algum** processo correto termina por executar sua seção crítica.
2. **Progresso justo (*Starvation-freedom*):** Se um processo correto está na seção de entrada (executando $enter()$), então **este** processo termina por executar sua seção crítica.

Note que a diferença entre a propriedade de progresso e a de progresso justo está no fato de que na primeira um processo pode ficar esperando por um recurso indefinidamente (“**algum** processo

correto termina por executar sua seção crítica") enquanto que na segunda não ("**este** processo termina por executar sua seção crítica").

Quando consideramos o problema da exclusão mútua em sistemas com processos sujeitos a falhas bizantinas [20], pelo menos uma premissa adicional deve ser assumida. Conforme observado por Lynch [83], o progresso de um algoritmo de exclusão mútua depende dos processos concorrentes saírem de sua seção crítica (liberarem o recurso compartilhado). Desta forma, assumimos que todos os processos que entram em sua seção crítica acabam por sair desta seção (mesmo que de maneira forçada). A forma de implementar esta premissa é dependente da aplicação. Nesta tese, quando usamos primitivas de exclusão mútua assumimos que um processo que entra em sua seção crítica executa uma tarefa exatamente uma vez e sai desta seção automaticamente com o término desta tarefa (ver seção 4.3.4).

Exclusão Mútua baseada no PAXOS BIZANTINO

Usando uma primitiva de difusão com ordem total é fácil implementar um serviço de exclusão mútua tolerante a faltas bizantinas utilizando um conjunto de servidores. Para tanto, é requerido um conjunto U com pelo menos $n \geq 3f + 1$ servidores que controlam os processos que competem pelo acesso a suas seções críticas, i.e. o conjunto de servidores controla os acessos em exclusão mútua provendo operações *enter*() e *exit*() de tal forma que as propriedades da exclusão mútua sejam satisfeitas. Clientes que desejam executar o algoritmo de exclusão mútua acessam este serviço. O algoritmo 1 apresenta o protocolo.

No protocolo do algoritmo 1, cada servidor s tem uma variável local $Queue_s$, que representa uma fila onde são armazenados os processos que utilizam o serviço de exclusão mútua. Esta fila é gerenciada através das operações usuais de lista como *append*, *empty*, *tail* e *head*. O processo que se encontra em primeiro na fila de um servidor é o detentor do recurso controlado para este servidor. A idéia fundamental do algoritmo é fazer com que o cliente envie uma mensagem avisando que deseja entrar em sua seção crítica usando difusão com ordem total (linha 1) para então ficar bloqueado a espera de $n - f$ respostas provenientes de diferentes servidores liberando o acesso ao recurso (linha 2). Todos os servidores corretos, ao receberem a requisição do cliente, colocam-no na fila de processos (linha 4), e esperam pela liberação do recurso compartilhado. O acesso ao recurso é dado aos processos conforme este é liberado por outros processos (linhas 5-7 e 12-14). A liberação do recurso é feita através do procedimento *exit*(): o processo envia mensagens a todos os servidores informando que ele saiu de sua seção crítica (linhas 3 e 8-10). O algoritmo 1 também define o predicado *hasLock*, usado para verificar se um processo está em sua seção crítica. Uma ilustração da execução deste algoritmo é apresentada na figura 2.4.

Levando-se em consideração que, por premissa, todo processo que obtém acesso ao recurso controlado acaba por liberá-lo (o procedimento *exit*() é sempre invocado para informar a liberação do recurso), é fácil ver que o algoritmo satisfaz as propriedades de "exclusão mútua" e "progresso justo" definidas na seção anterior. A primeira é garantida devido ao uso da difusão com ordem total (que

Algoritmo 1 Exclusão mútua (processo p e servidor s).

$\{Cliente\}$
procedure *enter*()
1: *TO-multicast*$(U, \langle \text{ENTER} \rangle)$
2: **wait until** *receive*$(s, \langle \text{GO} \rangle)$ de $n-f$ servidores $s \in U$
procedure *exit*()
3: $\forall s \in U, send(s, \langle \text{EXIT} \rangle)$
$\{Servidor\}$
Variável: $Queue_s = \langle\rangle$
upon *TO-receive*$(p, \langle \text{ENTER} \rangle)$
4: $Queue_s \leftarrow append(Queue_s, p)$
5: **if** $head(Queue_s) = p$ **then**
6: $\quad send(p, \langle GO \rangle)$
7: **end if**
upon *receive*$(p, \langle \text{EXIT} \rangle)$
8: **if** $head(Queue_s) = p$ **then**
9: $\quad unlock()$
10: **end if**
procedure *unlock*()
11: $Queue_s \leftarrow tail(Queue_s)$
12: **if** $\neg empty(Queue_s)$ **then**
13: $\quad send(head(Queue), \langle GO \rangle)$
14: **end if**

Predicado: $hasLock(p) \triangleq (head(Queue_s) = p)$

garante que as filas dos servidores conterão sempre a mesma ordem de processos) e pela impossibilidade de $n-f$ servidores darem permissão de acesso a dois processos[10]. A propriedade de progresso justo é conseqüência direta da propriedade de validade da difusão com ordem total (toda mensagem difundida é entregue) e da premissa que os processos sempre saem da seção crítica. Desta forma, todo processo que pede a entrada na seção crítica (executando a difusão com ordem total - linha 1), acaba recebendo permissão para entrar nesta seção (linha 2).

2.4 Replicação Tolerante a Faltas Bizantinas

A replicação tolerante a faltas bizantinas é um mecanismo fundamental para a construção de sistemas **tolerantes a intrusões** [57, 126]. Estes sistemas são capazes de prover um serviço correto mesmo que uma parte de seus componentes sejam atacados e controlados por entidades maliciosas. A necessidade da construção destes sistemas advém da impossibilidade de se construir um sistema e provar que este é seguro. Esta impossibilidade implica na fraqueza inerente da abordagem baseada apenas na prevenção de ataques e intrusões, já que mesmo os mecanismos preventivos podem se apresentar vulneráveis a determinados tipos de ataques. A tolerância a intrusões reconhece esse

[10] Considerando $|U| \geq 3f+1$, isso só seria possível se um servidor correto desse permissão a dois processos.

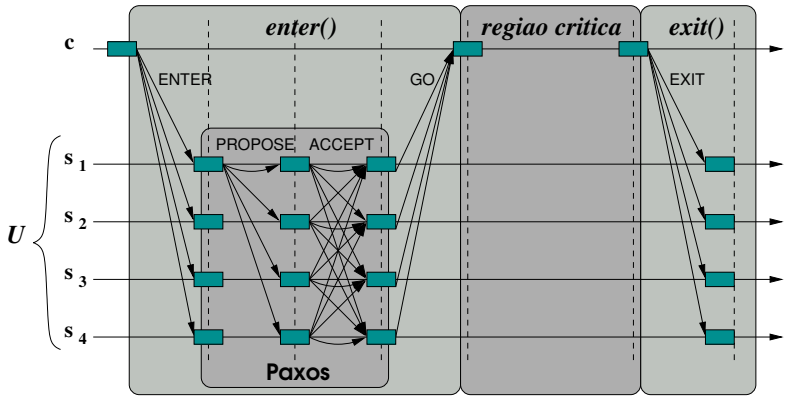


Figura 2.4: Exclusão mútua usando o PAXOS BIZANTINO.

problema, e tenta tolerar o comprometimento de alguns componentes do sistema.

Nesta seção apresentamos os dois principais modelos de replicação tolerante a faltas bizantinas descritos na literatura.

2.4.1 Replicação Máquina de Estados

O modelo de replicação mais comum para tolerância a faltas em sistemas sujeitos a faltas bizantinas é a replicação **Máquina de Estados** (também chamada replicação ativa) [76]. Este modelo considera o conjunto de réplicas que oferece um serviço composto por máquinas de estado deterministas que recebem entradas (requisições), as processam e produzem as saídas correspondentes (respostas). Estas máquinas de estado são caracterizadas da seguinte forma [114]:

Caracterização semântica de uma máquina de estado: As saídas de uma máquina de estados dependem unicamente de seu estado inicial e da seqüência de entradas processadas pela mesma.

A caracterização de uma máquina de estados desconsidera a interferência de outros fatores, que não as entradas, no estado da máquina. Por esta razão, processos não deterministas, cujo estado depende de algum tipo de aleatoriedade e/ou característica particular de seu ambiente de execução, não são consideradas no modelo Máquina de Estados. Esta característica esta relacionada ao **determinismo de réplicas** [114]: réplicas partindo de um mesmo estado inicial e sujeitas a mesma seqüência de requisições de entrada submetidas na mesma ordem devem chegar ao mesmo estado final.

O modelo de replicação Máquina de Estados deve ser entendido como um conjunto de máquinas de estados que começam a executar a partir de um mesmo estado inicial e processam a mesma seqüência de requisições. Este requisito é também chamado de **coordenação de réplicas** e pode ser dividido em dois requisitos particulares [114]:

1. **Acordo:** Todas as réplicas de máquina de estados corretas recebem as mesmas requisições;

2. **Ordem:** Todas as réplicas de máquina de estados corretas processam as requisições recebidas em uma mesma ordem.

Estes requisitos podem ser atendidos se utilizarmos um protocolo de difusão com ordem total (ver seção 2.3.2) para enviar as requisições ao conjunto de réplicas. Ao receber a requisição, cada réplica executa a operação, atualiza seu estado (quando necessário) e envia ao cliente o resultado da operação. Um cliente aceita um resultado da operação se recebe $f + 1$ respostas iguais das réplicas, sendo f o número máximo de servidores que podem sofrer falhas bizantinas.

Uma otimização usualmente implementada em replicação máquina de estados é a tentativa de execução de algumas operações sem a necessidade de execução do protocolo de ordem total. Com esta otimização, toda operação que não altere o estado do serviço (operação apenas de leitura) é enviada aos servidores, que respondem imediatamente. Se o cliente obtém $n - f$ respostas iguais, a operação termina; caso contrário, a requisição é reenviada através da difusão com ordem total. Esta otimização permite que uma leitura seja completada em dois passos de comunicação (envio e resposta) em ocasiões onde não existem faltas ou operações de escrita sendo executadas concorrentemente.

Nesta tese, e em especial no capítulo 3, utilizamos este modelo de replicação implementado através da difusão com ordem total baseada no PAXOS BIZANTINO.

2.4.2 Sistemas de Quóruns Bizantinos

Os sistemas de quóruns bizantinos [86] são uma abordagem alternativa à replicação Máquina de Estados para a implementação de segurança de funcionamento em sistemas assíncronos sujeitos a faltas bizantinas. Estes sistemas emulam objetos de memória compartilhados em sistemas em que os processos se comunicam por passagens de mensagens. O grande atrativo desta abordagem é o fato de que ela não necessita da resolução de problemas de acordo, não estando portanto sujeita a impossibilidade FLP [56]. Esta característica permite sua utilização em sistemas que não assumem premissas de sincronismo. Em contrapartida, um sistema de quóruns não pode implementar objetos mais fortes que registradores (que suportam apenas operações de leitura e escrita), i.e. todo serviço que suporte operações de leitura-escrita (como a operação *test&set*) não pode ser implementado de forma determinista e livre de espera (terminação da operação garantida [67]) em sistemas assíncronos.

Um sistema de quóruns para um universo de servidores de dados é um conjunto de vários conjuntos de servidores, chamados quóruns, que se intersectam. O princípio por trás de seu uso em serviços de armazenamento é que, se uma variável compartilhada é replicada entre todos esses servidores, as operações de leitura e escrita precisam ser feitas apenas em um dos quóruns destes servidores, e não em todo o sistema de quóruns. A existência de intersecções entre os quóruns garante a possibilidade de se construir protocolos de leitura e escrita que garantam a integridade destas operações mesmo que elas sejam realizadas em diferentes quóruns do sistema. Os sistemas de quóruns bizantinos são caracterizados por duas propriedades básicas [86]:

- **Consistência**: A intersecção entre quaisquer dois quóruns do sistema contém um **número suficiente** de servidores corretos[11];
- **Disponibilidade**: Sempre existe pelo menos um quórum no sistema composto apenas por servidores corretos.

A propriedade de consistência garante que é sempre possível obter o valor mais recente escrito no quórum, enquanto a propriedade de disponibilidade garante que sempre existe um quórum disponível para ser acessado no sistema.

A seguir apresentamos os dois tipos de sistemas de quóruns bizantinos utilizados nesta tese.

Sistemas de quóruns f-mascaramento [86]

Este é o tipo de sistema de quóruns mais simples, em que operações de leitura e escrita podem ser executadas em diferentes quóruns de servidores sujeitos a faltas bizantinas e mesmo assim o sistema ainda é capaz de permanecer consistente.

A idéia fundamental neste tipo de sistema é construir os quóruns de tal forma que: dado o sistema de quóruns $\mathscr{Q}$, a maioria dos servidores na intersecção entre quaisquer dois quóruns de $\mathscr{Q}$ sejam corretos. Mais precisamente, $\forall Q_1, Q_2 \in \mathscr{Q}, |Q_1 \cap Q_2| \geq 2f+1$. Dado este requisito, cada quórum do sistema $\mathscr{Q}$ deve conter $q = \lceil \frac{n+2f+1}{2} \rceil$ servidores e o número de servidores total no sistema deve ser $n \geq 4f+1$.

A figura 2.5 apresenta um exemplo de sistema de quórum deste tipo considerando $n = 5$ e $f = 1$.

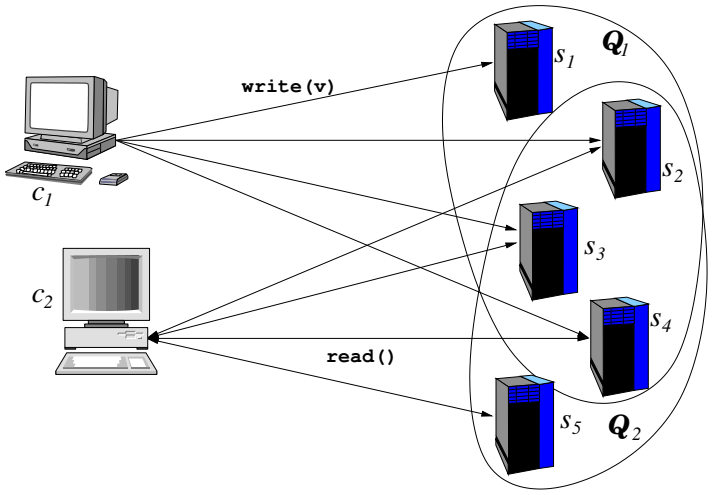


Figura 2.5: Exemplo de sistema de quóruns f-mascaramento ($n = 5$ e $f = 1$).

Nesta figura temos um cliente c_1 executando uma operação de escrita em um quórum $Q_1 = \{s_1, s_2, s_3, s_4\}$ e um cliente c_2 executando uma operação de leitura em um quórum $Q_1 = \{s_2, s_3, s_4, s_5\}$. Note que a intersecção entre esses dois conjuntos contém 3 servidores, respeitando o requisito de $2f+1$.

[11] Dependendo do tipo de sistema de quóruns, este número pode ser 1, $f+1$ ou até $2f+1$ [86].

Sistemas de quóruns assimétricos [93]

Este tipo de sistema de quórum se distingue dos sistemas de quóruns convencionais pelo uso de diferentes tamanhos de quóruns para diferentes operações. A condição de consistência básica neste tipo de sistema é que a intersecção entre quaisquer quóruns de leitura e escrita contenha uma maioria de servidores corretos, o que requer pelo menos $2f+1$ servidores.

Esta condição é semelhante a descrita anteriormente para os sistemas de quóruns f-mascaramento, porém nos sistemas assimétricos, é diminuído em f o número de servidores requeridos, fazendo com que escritores acessem um número maior de servidores na operação de escrita. Tendo em vista que o número de servidores acessados na escrita é maior que $n-f$, a propriedade de disponibilidade não pode ser garantida para os quóruns acessados na escrita (alguns podem ser falhos). Desta forma, o cliente escritor não aguarda confirmações da realização da operação, e portanto não sabe quando sua escrita termina. Este tipo de protocolo é chamado **não confirmável** [93].

Considerando estas particularidades, os sistemas de quóruns assimétricos de mascaramento requerem um número de servidores $n \geq 3f+1$ e dois tipos de quóruns ($\mathscr{Q} = \mathscr{Q}_r \cup \mathscr{Q}_w$) acessados em operações de leitura ($Q_r \in \mathscr{Q}_r$) e escrita ($Q_w \in \mathscr{Q}_w$), contendo $q_r = \lceil \frac{n+f+1}{2} \rceil$ e $q_w = \lceil \frac{n+f+1}{2} \rceil + f$ servidores, respectivamente.

A figura 2.6 apresenta um exemplo de sistema de quórum deste tipo considerando $n = 4$ e $f = 1$.

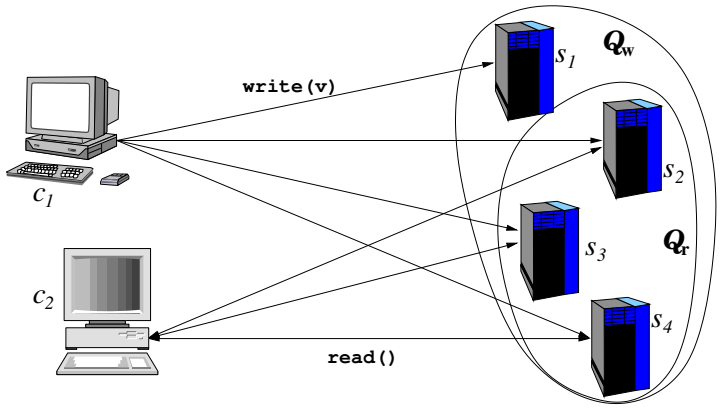


Figura 2.6: Exemplo de sistema de quóruns assimétricos ($n = 4$ e $f = 1$).

Esta figura apresenta um quórum de escrita $Q_w = \{s_1, s_2, s_3, s_4\}$ e um quórum de leitura $Q_r = \{s_2, s_3, s_4\}$. Dois clientes, c_1 e c_2, interagem com o sistema de quóruns invocando operações de leitura e escrita nos quóruns Q_w e Q_r, respectivamente.

2.5 Conclusões do Capítulo

Este capítulo apresentou o modelo de sistema considerado nesta tese e as principais ferramentas e conceitos utilizados em todos os demais capítulos. Além disso, foram apresentadas as duas principais técnicas para replicação tolerante a faltas bizantinas. Estas técnicas serão utilizadas nas diferentes construções de espaços de tuplas desenvolvidas nesta tese.

Capítulo 3

Segurança de Funcionamento em Espaço de Tuplas

Este capítulo inicia o estudo do espaço de tuplas com segurança de funcionamento: sua definição e os mecanismos necessários para sua implementação.

3.1 Contexto e Motivação

Toda atividade realizada por um conjunto de processos em um sistema distribuído pode ser caracterizada como computação ou interação. O conceito de coordenação advoga o desacoplamento destes dois tipos de atividades, evidenciando somente os aspectos de interação entre os processos. A partir desta separação, podem ser identificados quatro modelos de interação básicos chamados modelos de coordenação. No contexto de sistemas abertos, o modelo de coordenação generativa (também chamado de coordenação por espaço de tuplas) tem se mostrado o mais adequado graças as suas características de desacoplamento.

Desde a introdução do modelo de coordenação por espaço de tuplas, há alguma pesquisa sobre a provisão de tolerância a faltas no mesmo, tanto através da construção de espaços de tuplas tolerantes a faltas (tolerância a faltas em nível de espaço de tuplas) [11, 128], quanto em mecanismos que permitam a construção de aplicações tolerantes a faltas sobre o espaço de tuplas (tolerância a faltas em nível de aplicação) [71, 111]. O objetivo destes trabalhos é essencialmente garantir que *(i.)* o serviço provido pelo espaço de tuplas esteja sempre disponível mesmo que alguns dos servidores que o implementam falhem por parada, e *(ii.)* o estado do espaço de tuplas seja válido, de acordo com a semântica da aplicação, mesmo que algum processo desta aplicação falhe durante a execução de uma ou mais operações no espaço. O principal mecanismo usado para prover *(i.)* é a replicação, enquanto a provisão de *(ii.)* fica geralmente a cargo do suporte a transações. Mais recentemente, alguns esforços sobre espaços de tuplas seguros têm sido relatados na literatura [26, 44, 95, 127]. O objetivo destes trabalhos é garantir que processos não executem operações no espaço de tuplas sem permissão, através de mecanismos de controle de acesso.

Estes trabalhos em tolerância a faltas e segurança para espaço de tuplas têm um foco limitado em pelo menos dois sentidos: consideram apenas faltas acidentais por parada e ataques simples (acesso inválido); os mesmos tratam **ou** da tolerância a faltas **ou** da segurança do espaço de tuplas.

Neste capítulo vamos iniciar o estudo de segurança de funcionamento em espaço de tuplas introduzindo a noção de espaços de tuplas seguros e tolerantes a faltas. Esta solução se baseia na definição clara do que é um espaço de tuplas com segurança de funcionamento: um espaço de tuplas que apresenta os atributos de confiabilidade, disponibilidade, integridade e confidencialidade [10], mesmo com a ocorrência de faltas bizantinas. A aplicação dos mecanismos necessários para implementação destes atributos resulta em um espaço de tuplas capaz de tolerar faltas acidentais e maliciosas. Sendo portanto um sistema tolerante a intrusões [57, 126].

Os atributos de segurança de funcionamento descritos acima são suportados através de uma arquitetura em camadas em que cada nível da estratificação é responsável por implementar um ou mais destes atributos. O espaço de tuplas resultante é concretizado sobre um conjunto de servidores que implementam replicação segundo o modelo máquina de estados (ver seção 2.4.1). Esta abordagem garante que o espaço de tuplas se comporta de acordo com sua especificação mesmo que uma parte dos servidores falhe, acidentalmente ou maliciosamente. A construção proposta também tolera faltas em um número ilimitado de clientes que acessam o espaço de tuplas, além de prover mecanismos de segurança para evitar acessos de clientes não autorizados. Uma ilustração desse espaço de tuplas é apresentada na figura 3.1.

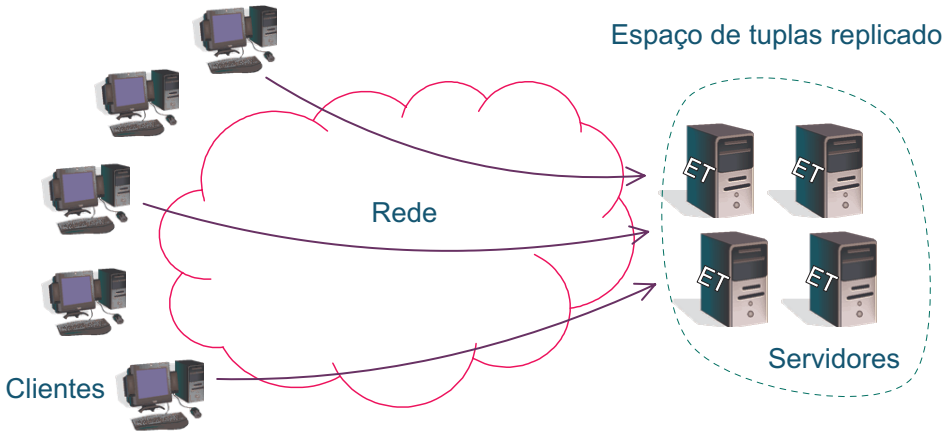


Figura 3.1: Espaço de tuplas com segurança de funcionamento.

Este capítulo introduz a definição de espaço de tuplas com segurança de funcionamento, no sentido mais forte deste termo: seguro, tolerante a faltas e intrusões. Após esta definição apresentamos uma arquitetura para implementação de espaço de tuplas com segurança de funcionamento. A definição desta arquitetura envolve uma combinação não-trivial de mecanismos de segurança e tolerância a faltas: replicação Máquina de Estados, controle de acesso em nível de espaço e tupla, criptografia para obtenção de confidencialidade (explicado no capítulo 5) e suporte à verificação de políticas de segurança de granularidade fina para impedir interações não previstas (de acordo com a aplicação).

Porém, antes de apresentar estas contribuições da tese, serão discutidos detalhadamente o conceito

de coordenação e o modelo de coordenação generativo.

3.2 Coordenação

O paradigma de coordenação, enquanto modelo conceitual para aplicações, é uma proposta interessante para o desenvolvimento de aplicações distribuídas complexas. Este paradigma estipula que a atividade de programação em sistemas distribuídos pode ser dividida em duas sub-atividades complementares [103]: as **computações** executadas pelos processos da aplicação (algumas vezes chamados agentes), representando a manipulação dos dados da aplicação, e a **coordenação**, representando as interações entre estes processos. Assim, qualquer modelo de programação pode ser construído a partir de um modelo de coordenação e um modelo de computação: a coordenação é o elemento que une as entidades ativas (computações) em uma aplicação.

Pode-se definir coordenação como a troca de informações (interação) entre agentes ativos cooperantes de uma aplicação [61]. Um elemento central para o conceito de coordenação é o **meio de coordenação**. Este é representado pelos mecanismos de transporte utilizados pelos processos para que suas interações se efetivem. A figura 3.2 ilustra esta idéia.

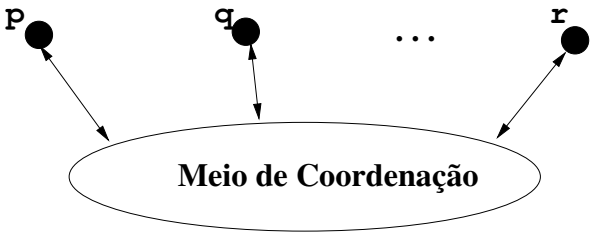


Figura 3.2: Processos interagem via seu meio de coordenação.

Formalmente, um modelo de coordenação pode ser definido como uma tupla $\langle E, M, L \rangle$, em que E representa as **entidades coordenadas**, M o **meio de coordenação** e L as **regras semânticas** as quais o modelo adere [103]. Além destes elementos, a cada modelo de coordenação podemos associar uma ou mais **linguagens de coordenação**, que definem a sintaxe das informações trocadas entre as entidades durante as interações bem como os itens linguísticos usados para realizar tais interações [101].

O conceito de coordenação está intimamente ligado a heterogeneidade das entidades coordenadas, importante característica dos sistemas abertos. A separação clara entre coordenação e computação permite que entidades com diferentes características e capacidades possam interagir plenamente em um sistema computacional.

Existem quatro tipos básicos de modelos de coordenação [30], conforme apresentado na tabela 3.1.

Estes modelos são classificados a partir do grau de acoplamento temporal e espacial apresentado pelas entidades que interagem. O acoplamento temporal está relacionado com a sincronização das

Tipo de Acoplamento	Tempo		
Espaço	Acoplamento	Acoplado	Desacoplado
	Acoplado	**Direta**	**Quadro Negro**
	Desacoplado	**Orientada a Encontro**	**Generativa**

Tabela 3.1: Tipos de modelos de coordenação.

comunicações entre entidades durante as interações. Em um modelo acoplado temporalmente é preciso que as entidades estejam engajadas em comunicações em um mesmo instante para que exista interação. O acoplamento espacial está relacionado com o fato de só existir comunicação entre entidades se estas conhecem a localização de seus pares. Em um sistema acoplado espacialmente, as entidades devem conseguir endereçar umas às outras para que as interações aconteçam. Os quatro modelos apresentados na tabela 3.1 são definidos da seguinte forma [30]:

- **Direta:** Este é o modelo de coordenação mais usado pelas aplicações distribuídas atuais, nele as entidades se comunicam de forma direta, através de comunicações sincronizadas e explicitamente endereçadas. Algumas concretizações deste modelo são as redes par a par (*peer-to-peer*), em que as entidades se comunicam diretamente, e cliente/servidor, em que entidades clientes requisitam serviços à entidades diferenciadas denominadas servidores. Os *sockets* TCP/IP [106] e o modelo RMI do CORBA [100] são representantes deste modelo;

- **Orientada a Encontro:** O modelo de coordenação orientado a encontro permite que processos se comuniquem através de encontros de forma espacialmente desacoplada. Encontros são canais públicos ou listas de distribuição de eventos e mensagens, em que as entidades se registram visando interação. Apesar da coordenação orientada a encontro prover comunicação desacoplada no espaço (já que as entidades não conhecem os endereços umas das outras), ela não fornece desacoplamento temporal, pois as interações ocorrem de forma sincronizada: cada informação produzida no encontro é recebida por todas as entidades presentes no momento. Alguns exemplos de mecanismos de interação baseadas neste modelo são os serviços de eventos e de filas de mensagens. Dentre os vários exemplos de implementação de modelos deste tipo podemos citar os serviços de eventos e notificação CORBA definidos pela OMG [100] e a especificação WS-Notification [65] para serviços *web*;

- **Quadro Negro:** No modelo baseado em quadro negro as entidades interagem através de espaços de dados compartilhados localizados em determinados nós da rede. Este modelo é temporalmente desacoplado, já que as entidades comunicantes podem deixar e ler informações no quadro sempre que desejarem. Estas operações de escrita e leitura não necessariamente precisam ser feitas ao mesmo tempo, porém o acoplamento espacial existe devido ao fato das entidades precisarem identificar as informações deixadas no quadro tanto na leitura quanto na escrita. Como conseqüência, os leitores do quadro devem conhecer os identificadores das informações deixadas no quadro negro para obtê-las. O modelo de coordenação baseado em quadro negro foi criado inicialmente para aplicações de inteligência artificial e ainda hoje sua utilização se dá em maior parte neste contexto, sendo empregado principalmente para a coordenação de agentes

inteligentes. O modelo de computação distribuída em que os processos se coordenam através de um banco de registradores compartilhados também pode ser visto como uma instância deste tipo de coordenação, uma vez que estes só podem se comunicar se souberem os endereços dos registradores usados para troca de informações;

- **Generativa:** A coordenação generativa faz uso de um espaço de memória compartilhado (de forma análoga ao modelo de quadro negro) pelas entidades juntamente com mecanismos associativos de acesso às informações neste espaço [59]: a informação é organizada em tuplas e acessada de modo associativo através de casamento de padrões. Os mecanismos associativos usados reforçam ainda mais o desacoplamento entre as entidades permitindo que estas possam se comunicar sem nem mesmo saber o formato exato das comunicações. A implementação original deste modelo de coordenação foi realizada na linguagem LINDA [59], porém, inúmeras extensões e sistemas foram posteriormente propostos, sempre tirando proveito de sua flexibilidade.

Mais recentemente, alguns autores adicionaram um novo tipo de desacoplamento aos já conhecidos desacoplamento temporal e espacial: o **desacoplamento de bloqueio**[1] [55]. Nesta forma de desacoplamento, as primitivas utilizadas pelos processos para se coordenarem não são bloqueantes. Isto implica na inexistência de sincronização entre diferentes processos no sistema. Um exemplo de modelo de coordenação desacoplado no tempo, no espaço e no bloqueio é o *Publisher/Subscriber* [55]. As primitivas nele utilizadas para o envio de eventos são não bloqueantes e, uma vez um evento produzido, o produtor não espera até que os demais processos interessados recebam este evento. No caso do espaço de tuplas, o desacoplamento de bloqueio não existe, já que todas as primitivas de acesso ao espaço são bloqueantes (ver próxima seção). No entanto, extensões ao modelo básico, visando prover primitivas não bloqueantes para o espaço de tuplas (ex. [112]), tornam o modelo desacoplado também no bloqueio.

Uma outra taxonomia para a classificação de modelos de coordenação é proposta em [103]. Nela os modelos são divididos em **orientados a dados** e **orientados a controle**. A principal característica dos modelos orientados a dados é a existência de um espaço de dados compartilhado entre as entidades, enquanto nos modelos orientados a controle as entidades interagem através de eventos comunicados entre elas. Aplicando essa definição aos modelos apresentados na tabela 3.1 temos que a coordenação direta e a orientada a encontros são modelos de coordenação orientados a controle enquanto a baseada em quadro negro e a generativa são orientadas a dados.

No contexto de sistemas abertos, em que entidades heterogêneas podem participar em momentos diversos das computações distribuídas, o modelo generativo mostra-se o mais adequado. Em ambientes dinâmicos, como redes *ad hoc* compostas de dispositivos móveis, é praticamente impossível ter conhecimento completo e atualizado sobre as entidades comunicantes. Desta forma, um mecanismo de coordenação completamente desacoplado se mostra muito mais atraente. Além disso, algumas aplicações podem tirar proveito das propriedades de memória associativa para lidar com a incerteza, dinamicidade e heterogeneidade da informação, a partir da combinação de padrões entre tuplas.

[1] Evitamos utilizar a tradução direta do original, desacoplamento de sincronização (*synchronization decoupling*), para evitar confusões com relação aos aspectos de tempo do sistema (sincronismo).

3.3 Coordenação Generativa

A **coordenação generativa** [60] (também chamada coordenação por espaço de tuplas) foi introduzida no contexto da linguagem de programação para sistemas paralelos LINDA [59]. O mecanismo de coordenação definido nesta linguagem permite aos processos distribuídos interagirem sobre um espaço de memória compartilhado em que tuplas são adicionadas, lidas e removidas. A comunicação é chamada generativa devido ao fato de uma vez criadas no espaço, as tuplas têm existência completamente separada dos processos que as geraram, e podem inclusive, conter outros processos.

A figura 3.3 apresenta o modelo de coordenação generativa. Nesta figura os processos interagem através da inclusão e remoção de tuplas em um espaço de memória associativa denominado **espaço de tuplas**.

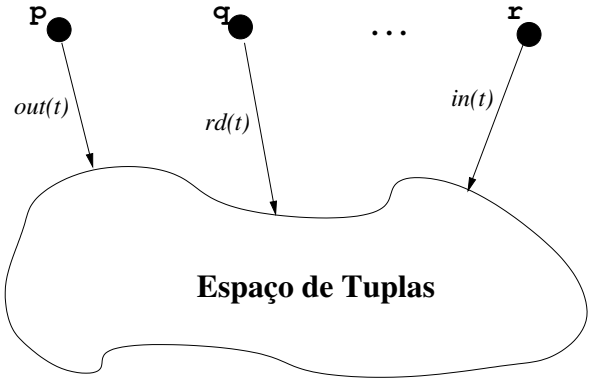


Figura 3.3: Processos interagindo via um espaço de tuplas.

Uma tupla $t = \langle f_1, f_2, ..., f_n \rangle$ é um conjunto de campos ordenados. Cada campo f_i da tupla pode ser um atual, um formal ou o símbolo especial '$*$'. Um atual contém um valor de um determinado tipo. Um formal representa apenas o tipo do campo, e geralmente é representado por uma variável deste tipo precedida por um '?' (ex. $?v$). O símbolo especial '$*$' em um campo representa que tal campo pode ser qualquer variável de qualquer tipo. Uma tupla em que todos os parâmetros são atuais é também chamada de entrada (*entry*). Um molde (*template*) é uma tupla que pode conter campos formais e símbolos '$*$'. Neste texto, denotamos as entradas por t e os moldes por $\bar{t}$. O **tipo de um campo**, seja ele atual ou formal, é dado pela função $\tau : \mathscr{F} \rightarrow \mathscr{T}$, onde $\mathscr{F}$ é o conjunto de todos os possíveis campos, atuais ou formais, e $\mathscr{T}$ é o conjunto dos possíveis tipos de dados assumidos no modelo de computação usado. O **tipo de uma tupla** é definido como a seqüência dos tipos dos campos da tupla. Em princípio, duas tuplas combinam se elas compartilham algumas características como valores e tipos de campos.

As manipulações realizadas no espaço de tuplas consistem de invocações de três operações básicas [59, 60]:

- $out(t)$: adiciona a tupla t no espaço de tuplas. A operação *out* é chamada usualmente de operação de inserção;

- $in(\bar{t})$: retira uma tupla que combina com o molde $\bar{t}$ do espaço de tuplas. Caso não exista nenhuma tupla que combine com $\bar{t}$ no espaço, o processo fica parado esperando até que exista uma (operação bloqueante). Esta operação é usualmente chamada de leitura destrutiva, remoção ou coleta;

- $rd(\bar{t})$: lê uma tupla que combina com $\bar{t}$ no espaço. Caso não exista nenhuma tupla que combine com $\bar{t}$ no espaço o processo fica parado esperando até que exista uma (operação bloqueante). A grande diferença entre esta operação e a operação *in* é que ela não remove a tupla do espaço, apenas lê seus valores.

São também definidas duas variantes das operações *in* e *rd* não bloqueantes, estas operações são chamadas *inp* e *rdp* e seus funcionamentos são exatamente iguais as operações originais, a não ser pelo fato de que elas sempre retornam um valor lógico: se não houver uma tupla no espaço que combine com o molde passado, estas operações retornam um valor de falha *false*, caso contrário, um valor *true* é retornado juntamente com a tupla lida. Note que o não bloqueio destas operações é em relação à não espera pela existência no espaço de uma tupla que combine com o molde usado. *inp* e *rdp* ainda são bloqueantes no que diz respeito ao acesso ao espaço de tuplas. Algumas implementações deste modelo (em especial a linguagem LINDA) também definem a operação $eval(p)$ para a criação de novos processos. A execução de $eval(p)$ cria o processo p que pode interagir através do espaço de tuplas [59].

As operações definidas para um espaço de tuplas são inerentemente não deterministas. Se existirem várias tuplas no espaço que combinam com um molde passado como parâmetro em uma operação *in* ou *rd*, qualquer uma delas pode ser retornada. Da mesma forma, se dois ou mais processos estiverem parados esperando por tuplas com determinadas características (definidas nos moldes apresentados como argumentos nas operações) e uma entrada é inserida no espaço de tal forma que ela combina com os moldes de qualquer um dos processos esperando a tupla, qualquer um deles pode recebê-la, permanecendo os demais em estado de espera.

Uma característica fundamental da coordenação generativa é o acesso associativo a tuplas: os dados são acessados a partir de seu conteúdo, e não através de seu endereço. A idéia é que o espaço de tuplas é uma sacola onde itens de dados (tuplas) são colocados, lidos e retirados de acordo com suas características (conteúdo). Estes dados, uma vez no espaço de tuplas, não podem ser alterados. Desta forma, para se alterar uma tupla do espaço é preciso removê-la e criar outra tupla no espaço com os valores da antiga, porém alterada. Esta característica de memória associativa diferencia de maneira cabal este modelo de coordenação dos demais modelos baseados em memória compartilhada. Fundamental para este mecanismo é o conceito de **combinação de tuplas**. Diz-se que duas tuplas, uma entrada $t = \langle f_1, f_2, ..., f_n \rangle$ e um molde $\bar{t} = \langle \overline{f}_1, \overline{f}_2, ..., \overline{f}'_m \rangle$ combinam, denotado por $m(t, \bar{t})$, se e somente se as seguintes condições se verificam:

1. $n = m$;
2. $\forall i = 1..n : \overline{f}_i = * \vee \overline{f}_i = f_i \vee (formal(\overline{f}_i) \wedge \tau(f_i) = \tau(\overline{f}_i))$

A primeira condição verifica se o número de campos do molde é igual ao da tupla, já a segunda verifica se os campos comparados das tuplas são compatíveis. Esta segunda condição usa o predicado $formal(x)$, que é definido como verdadeiro se x é um campo formal e falso caso contrário.

Para fins de ilustração da regra definida, considere a entrada $\langle 1, 2, \text{“}request\text{”} \rangle$. Esta tupla combina com os seguintes moldes:

- $\langle *, *, * \rangle$;
- $\langle 1, *, * \rangle$;
- $\langle ?i, 2, ?s \rangle$ (i inteiro e s *string*);
- $\langle *, ?i, \text{“}request\text{”} \rangle$ (i inteiro).

Esta tupla, entretanto, não combina com os seguintes moldes:

- $\langle 1, ?s, * \rangle$ (s *string*): tipo do segundo campo não combina;
- $\langle ?i, 2, \text{“}response\text{”} \rangle$ (i inteiro): o valor do terceiro campo não combina e ele não é um formal;
- $\langle 1, *, *, * \rangle$: o número de campos do molde é maior que o da entrada.

Estes exemplos ilustram o uso da **nomeação estruturada** para restringir o espaço de busca em operações de leitura (buscando tuplas com campos contendo valores específicos) e abrir este mesmo espaço (gerando tuplas contendo campos formais que podem ser combinados com qualquer valor do mesmo tipo) [59].

Uma importante consideração a ser feita sobre o modelo de coordenação generativa, é o fato do espaço de tuplas ser uma memória compartilhada virtual. Isto significa que ele não necessariamente precise ser implementado em uma memória compartilhada ou em um nó centralizado de uma rede de computadores, o conceito apenas impõe que este espaço deve estar acessível aos processos que quiserem interagir através dele e que as tuplas depositadas no espaço estejam acessíveis a todos os processos interagindo através dele. As primeiras implementações do espaço de tuplas foram realizadas em ambientes com memória distribuída [59, 60]. Nesta tese, apresentamos vários algoritmos para implementação do espaço de tuplas em um conjunto U de servidores.

3.3.1 Definição formal da Coordenação Generativa

Formalmente, o modelo de coordenação generativa é definido através da tupla $\langle \Pi, TS, L_g \rangle$. Sendo Π o conjunto de todos os processos que podem participar de computações distribuídas, TS um espaço de tuplas compartilhado pelos processos e L_g o conjunto de regras semânticas definidas na tabela 3.2.

Nesta tabela, o comportamento das operações é definido através dos valores retornados por elas e pelas alterações que elas causam no espaço de tuplas. Denotamos por $TS \xrightarrow[r]{op} TS'$ o efeito da operação

$$TS \cup \{t\} \xrightarrow[t]{in(\bar{t})} TS, m(t,\bar{t}) \qquad TS \cup \{t\} \xrightarrow[t]{rd(\bar{t})} TS \cup \{t\}, m(t,\bar{t})$$

$$TS \cup \{t\} \xrightarrow[true,t]{inp(\bar{t})} TS, m(t,\bar{t}) \qquad TS \cup \{t\} \xrightarrow[true,t]{rdp(\bar{t})} TS \cup \{t\}, m(t,\bar{t})$$

$$TS \xrightarrow[false]{inp(\bar{t})} TS, \nexists t \in TS : m(t,\bar{t}) \qquad TS \xrightarrow[false]{rdp(\bar{t})} TS, \nexists t \in TS : m(t,\bar{t})$$

$$TS \xrightarrow[-]{out(t)} TS \cup \{t\}$$

Tabela 3.2: Semântica das operações em um espaço de tuplas (L_g).

op sobre o espaço de tuplas TS retornando para a entidade executora o valor r e colocando o espaço no estado TS'. A tabela mostra o resultado das 5 operações [59] definidas originalmente na coordenação generativa. As duas primeiras regras mostram o resultado das operações $in(\bar{t})$ e $rd(\bar{t})$ quando $m(t,\bar{t})$, sendo t uma tupla presente no espaço de tuplas. Note que t é removida do repositório na operação $in(t)$. As duas regras seguintes mostram o resultado das variantes não bloqueantes destas operações quando estas são bem sucedidas. Note que elas são iguais a não ser pelo valor *true* retornado junto com a tupla. O resultado das operações não bloqueantes quando não existe tupla nenhuma no espaço que combina com $\bar{t}$ é definido nas duas regras seguintes. Finalmente a operação $out(t)$ ser definida como a adição de t ao conjunto de tuplas no espaço TS.

3.3.2 Espaço de Tuplas Aumentado

Em [117] é provado que o espaço de tuplas é um tipo de objeto de memória compartilhada com número de consenso igual a 2, na hierarquia livre de espera definida por Herlihy [67]. Isto significa que ele pode ser usado para resolver consenso entre dois processos em sistemas assíncronos mesmo na presença de falhas [67]. Nesta tese queremos resolver consenso e apresentar construções universais para um número arbitrário de processos, assim, precisamos de objetos de memória compartilhada universais (com número de consenso n, em um sistema com n processos) [9, 67]. Conseqüentemente, usamos um **espaço de tuplas aumentado** [11, 117] que suporta uma operação extra de **inserção condicional atômica** (*Conditional Atomic Swap* - CAS). Esta operação, denotada por $cas(\bar{t},t)$ para um molde $\bar{t}$ e uma entrada t, funciona como uma execução atômica (indivisível) da seguinte instrução:

$$\textbf{if } \neg rdp(\bar{t}) \textbf{ then } out(t)$$

O significado desta instrução é "se a leitura de $\bar{t}$ falhar, insira a entrada t no espaço"[2]. Esta operação retorna *true* se a tupla é inserida no espaço e *false* caso contrário. O espaço de tuplas aumentado é um objeto de memória compartilhada universal, já que pode ser usado para resolver consenso livre de espera de forma trivial tanto no modelo de faltas por parada [11, 117] quanto com faltas bizantinas (como será visto no capítulo 6) para um número qualquer de processos.

A adição dessa operação implica na adição das regras definidas na tabela 3.3 àquelas apresentadas na tabela 3.2.

[2]Note que o significado da operação *cas* do espaço de tuplas é o oposto da conhecida operação *compare&swap* para registradores [9], onde o estado do objeto é modificado apenas se seu estado é **igual** ao valor comparado.

$$TS \cup \{t\} \xrightarrow[false,t]{cas(\bar{t},t')} TS \cup \{t\}, m(t,\bar{t}) \quad TS \xrightarrow[true]{cas(\bar{t},t')} TS \cup \{t'\}, \nexists t \in TS : m(t,\bar{t})$$

Tabela 3.3: Semântica da operação *cas*.

3.4 Definindo Espaço de Tuplas com Segurança de Funcionamento

Esta seção apresenta uma definição para espaço de tuplas com segurança de funcionamento e a seguir discute os mecanismos necessários para implementar este tipo de objeto.

3.4.1 Atributos de Segurança de Funcionamento

Um espaço de tuplas é dito com segurança de funcionamento se ele satisfaz os **atributos de segurança de funcionamento** [10]. Como muitos outros sistemas, alguns destes atributos não são aplicáveis ou são ortogonais ao projeto do espaço de tuplas (por exemplo, *safety* e capacidade de manutenção – *manutenability*). Os atributos relevantes neste caso são:

- **Confiabilidade:** as operações realizadas no espaço de tuplas fazem com que seu estado se modifique de acordo com sua especificação (tabelas 3.2 e 3.3);
- **Disponibilidade:** o espaço de tuplas sempre está pronto a 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 dos campos de uma tupla não podem ser revelados a partes não autorizadas.

A dificuldade em garantir estes atributos advém da ocorrência de faltas, de natureza acidental (um *bug* no software ou uma parada no servidor) ou maliciosa (um invasor que modifica uma tupla no servidor). O objetivo é evitar que estas faltas causem uma falha no espaço de tuplas, i.e. que um ou mais destes atributos sejam violados. Faltas maliciosas são particularmente difíceis de serem tratadas já que não é possível definir premissas a respeito do que um invasor pode ou não fazer em sistema [126]. Estas faltas são usualmente modeladas como a classe mais genérica de faltas – faltas arbitrárias ou bizantinas – de tal forma que a solução proposta neste capítulo é genérica em termos do tipo de falta tolerada.

O significado dos atributos de confiabilidade e disponibilidade são claros e podem ser facilmente entendidos, entretanto integridade e confidencialidade requerem alguma discussão. Uma alteração no espaço de tuplas é dita apropriada (vs. inapropriada) se e somente se *(i.)* ela é o resultado de uma das operações *out*, *in*, *inp* ou *cas*; e *(ii.)* se a operação satisfaz a **política de acesso** do espaço de tuplas. Uma política de acesso básica define quais operações no espaço de tuplas cada processo

tem direito de executar. A maneira mais simples e efetiva de prover este tipo de política é definindo quem pode inserir uma tupla no espaço e, para cada tupla inserida, quem pode lê-la e quem pode removê-la. O uso deste tipo de política em espaços de tuplas é apresentado na seção 3.5.3. Uma política mais elaborada seria usada para garantir ou negar a execução de operações no espaço de tuplas levando em consideração três parâmetros: a identidade do processo que invoca a operação, a operação a ser executada e seus argumentos, e as tuplas presentes no espaço. Um exemplo de política de acesso desse tipo definida informalmente pode ser: permita a execução apenas de *(i.)* qualquer operação *rd*/*rdp* e *(ii.)* qualquer operação *out* invocada pelos clientes p_1 ou p_2 em que o primeiro campo da tupla sendo inserida seja um inteiro positivo. Chamamos estas políticas de granularidade fina porque elas permitem o uso de uma variedade considerável de parâmetros [18, 95]. Este tipo de política é apresentado na seção 3.5.4 e exaustivamente discutido no capítulo 6. Políticas de acesso são importantes porque previnem clientes não autorizados de ler e remover tuplas do espaço bem como clientes maliciosos de "inundar" o espaço com uma grande quantidade de tuplas visando causar uma negação de serviço [26].

Esta discussão sobre controle de acesso também nos ajuda a clarificar o significado do atributo de confidencialidade. A idéia geral é que o conteúdo de uma tupla não pode ser revelado a não ser por uma operação que satisfaça a política de acesso do espaço de tuplas. Esta política define quais são as partes autorizadas (vs. não autorizadas) para cada operação em um dado instante. No entanto, a provisão de confidencialidade é um pouco mais complicada: confidencialidade pode ser requerida para alguns campos mas não para todos.

3.4.2 Mecanismos para Segurança de Funcionamento

A segurança de funcionamento de um espaço de tuplas pode ser implementada utilizando uma combinação de diversos mecanismos. A técnica básica para implementar um espaço de tuplas com segurança de funcionamento é a **replicação**: o espaço de tuplas é replicado em um conjunto de n servidores de forma que a falha em alguns deles não viole a confiabilidade, disponibilidade e integridade do sistema. A idéia é utilizar um esquema de replicação que permita que mesmo em caso de falha em alguns servidores (sendo controlados por invasores e agindo maliciosamente ou apenas parando), o espaço de tuplas permaneça disponível com suas operações conforme sua especificação (mantendo confiabilidade e integridade). Obviamente, existe um limite no número f de réplicas que podem falhar, este número depende da solução de replicação empregada.

Outro mecanismo fundamental para segurança de funcionamento, especialmente para manutenção da integridade e confidencialidade, é o controle de acesso. Este tipo de mecanismo é necessário para prevenir clientes não autorizados de inserir (*out*), remover (*in* e *inp*) ou ler (*rd*, *rdp*) do espaço de tuplas. Controle de acesso é usualmente implementado em servidores replicados através de um monitor de referência que implementa a mesma política instalado em cada um dos servidores.

O terceiro tipo de mecanismo é a criptografia, que é usada para garantir confiabilidade nas comunicações e a confidencialidade das tuplas. Implementar confidencialidade em um espaço de tuplas replicado não é simples por diversas razões. A primeira, e mais importante, é que não podemos

confiar nos servidores individualmente para garantir a confidencialidade das tuplas uma vez que até f servidores podem falhar, possivelmente revelando o conteúdo das tuplas armazenadas a partes não autorizadas. A segunda complicação advém da necessidade de combinação entre tuplas e moldes. Campos cifrados geralmente não podem ser comparados com campos de um molde. Além disso, algumas políticas de acesso de granularidade fina podem impor limites nos campos que podem ser cifrados. Para satisfazer estes dois requisitos, muitas vezes conflitantes, definimos três tipos de campos: **campos públicos**, cujo conteúdo pode ser revelado e comparado; **campos privados**, cujo conteúdo não pode ser revelado ou comparado; e **campos comparáveis**, cujo conteúdo não pode ser revelado, mas permite comparações de igualdade. O uso destes três tipos de campos em uma tupla permite a implementação de controle de acesso por políticas de granularidade fina mantendo um certo nível de confidencialidade, o que de outra forma seria impossível dada a natureza conflitante destes mecanismos. Os aspectos específicos sobre o uso de criptografia para manutenção de confidencialidade em espaços de tuplas são discutidos no capítulo 5.

3.5 Uma Arquitetura para Espaço de Tuplas com Segurança de Funcionamento

Esta seção apresenta o projeto de um espaço de tuplas com segurança de funcionamento que satisfaz a definição apresentada na seção anterior. Inicialmente apresentaremos a arquitetura geral do sistema para então definir cada uma das camadas definidos nesta.

3.5.1 Arquitetura

A arquitetura do espaço de tuplas atendendo atributos de segurança de funcionamento proposto neste capítulo consiste em uma série de camadas que implementam os mecanismos necessários para provisão dos atributos descritos na seção 3.4. A figura 3.4 apresenta esta arquitetura.

Na figura temos duas pilhas de protocolos, uma para os clientes, e outra para os servidores. Considerando o modelo conceitual adotado nesta tese (seção 2.1.5), a pilha cliente, excluindo a camada de aplicação, corresponde aos autômatos C, enquanto a pilha servidor corresponde aos autômatos R. Ainda neste modelo, a camada de aplicação do cliente corresponde ao autômato P.

Na camada mais alta da pilha cliente temos a aplicação que usa o espaço, enquanto neste mesmo nível da pilha servidor temos uma instância do espaço de tuplas. A comunicação segue um esquema similar à chamada remota de procedimentos. A aplicação interage com o sistema chamando funções com as assinaturas usuais das operações em espaço de tuplas: $out(t)$, $rd(\bar{t})$, etc. Estas funções são acessadas através de um *stub*. A camada abaixo trata do controle de acesso em nível de tuplas (seção 3.5.3). Mais abaixo temos a camada que cuida da confidencialidade, que será explicada no capítulo 5. Finalmente temos a camada que implementa os protocolos para replicação (seção 3.5.2). O lado servidor é similar, exceto por uma nova camada para verificação da política de acesso em cada operação requisitada (seção 3.5.4). O *skeleton* chama localmente as operações do espaço de tuplas, de acordo

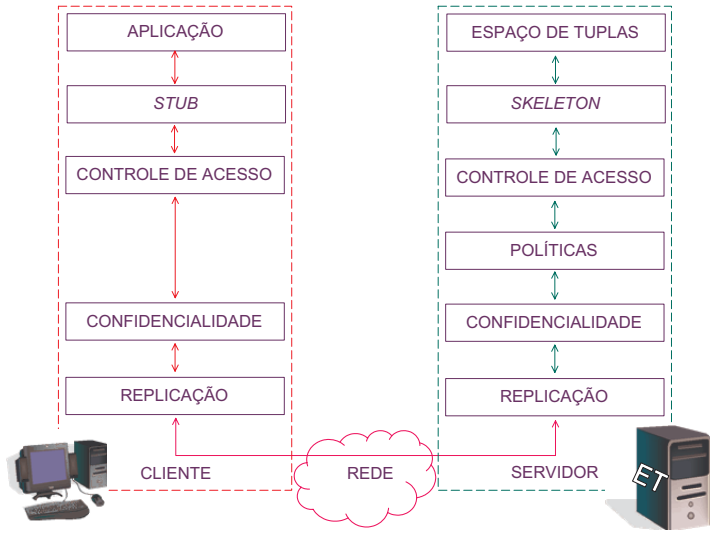


Figura 3.4: Arquitetura para o espaço de tuplas com segurança de funcionamento.

com as mensagens de requisição recebidas. Como nenhuma implementação de espaço de tuplas conhecida suporta a operação *cas*, esta operação deve ser implementada pelo *skeleton* usando operações providas pelo espaço de tuplas na camada subseqüente. Esta camada traduz cada chamada por $cas(\bar{t},t)$ no seguinte procedimento (atômico): $rdp(\bar{t})$ é executada e, se o seu resultado é *false*, a operação $out(t)$ é executada.

3.5.2 Espaço de Tuplas Replicado

Na seção 3.4 foi dito que o mecanismo básico para implementação de um espaço de tuplas com segurança de funcionamento é a replicação. A idéia de replicar um espaço de tuplas em n servidores, i.e. colocar um espaço de tuplas em cada um dos n servidores e garantir um nível de consistência entre estes espaços. Uma solução simples é usar replicação Máquina de Estados (ver seção 2.4.1). Esta abordagem garante a linearização [69], uma propriedade forte de consistência na qual operações executadas concorrentemente sobre o objeto implementado (neste caso, o espaço de tuplas) aparentam terem sido executadas seqüencialmente, uma após a outra.

Como discutido anteriormente, a replicação máquina de estados requer que todas as replicas *(i.)* comecem em um mesmo estado e *(ii.)* executem todas as requisições (i.e. operações no espaço de tuplas) na mesma ordem. A primeira condição é facilmente garantida iniciando os espaços de tuplas das réplicas sem nenhuma tupla. A segunda condição requer o uso de um protocolo de difusão com ordem total tolerante a faltas bizantinas. Um outro requisito implícito na replicação Máquina de Estados é o determinismo de réplicas: a mesma seqüência de operações executadas em réplicas no mesmo estado inicial levam as réplicas ao mesmo estado. Isto implica que as operações de leitura e remoção executadas em diferentes servidores devem ter o mesmo resultado. Este requisito limita o uso de espaços de tuplas COTS (*Commercial Off-The-Shelf*) nas réplicas, já que a definição original deste modelo de coordenação não requer que as operações sejam deterministas [59]. Entretanto, se o espaço de tuplas usado nas diferentes réplicas não apresenta leituras deterministas, o determinismo

pode ser emulado usando especificações e funções abstratas[3] [37].

Algoritmo 2 Replicação Máquina de Estados do espaço de tuplas (cliente p_i e servidor s_j).

{CLIENTE}

procedure *out*(t)
1: *execute_op*(OUT,t)

procedure *rdp*($\bar{t}$)
2: **return** *execute_rd*(RDP,$\bar{t}$)

procedure *inp*($\bar{t}$)
3: **return** *execute_op*(INP,$\bar{t}$)

procedure *cas*($\bar{t}$,t)
4: **return** *execute_op*(CAS,$\langle\bar{t},t\rangle$)

procedure *rd*($\bar{t}$)
5: **return** *execute_rd*(RD,$\bar{t}$)

procedure *in*($\bar{t}$)
6: **return** *execute_op*(IN,$\bar{t}$)

procedure *execute_rd*(op,$\bar{t}$)
7: $\forall s \in U$, *send*(s,$\langle op,\bar{t}\rangle$)
8: $R[s_1..s_n] \leftarrow \perp$
9: **repeat**
10: **wait** *receive*(s,$\langle$RESP,$r_s\rangle$))
11: $R[s] \leftarrow r_s$
12: **until** $\#R_\perp \leq f$
13: **if** $\exists t : (\#R_t \geq n-f)$ **then**
14: **return** t
15: **else**
16: **return** *execute_op*(op,$\bar{t}$)
17: **end if**

procedure *execute_op*(op,arg)
18: *TO-multicast*(U,$\langle op,arg\rangle$)
19: $R[s_1..s_n] \leftarrow \perp$
20: **repeat**
21: *receive*(s,$\langle$RESP,$r_s\rangle$)
22: $R[s] \leftarrow r_s$
23: $r \leftarrow$ *extract_response*(R)
24: **until** $r \neq \perp$
25: **return** r

{SERVIDOR}

ts: camada superior do lado servidor

upon *TO-deliver*(p_i,$\langle op,arg\rangle$)
26: **if** $op =$ OUT **then**
27: $t \leftarrow arg$
28: $ts.out(t)$
29: $t_r \leftarrow t$
30: **else if** $op =$ CAS **then**
31: $\bar{t} \leftarrow arg[0]$
32: $t \leftarrow arg[1]$
33: $t_r \leftarrow ts.cas(\bar{t},t)$
34: **else if** $op =$ IN $\vee\, op =$ INP **then**
35: $t_r \leftarrow ts.inp(arg)$
36: **else if** $op =$ RD $\vee\, op =$ RDP **then**
37: $t_r \leftarrow ts.rdp(arg)$
38: **end if**
39: **if** $t_r =$ NO_MATCH $\wedge\, (op =$ IN $\vee\, op =$ RD$)$ **then**
40: *add_blocked*(op,p_i,$\bar{t}$)
41: **else**
42: *send*(p_i,$\langle$RESP,$t_r\rangle$)
43: **end if**
44: **if** $op =$ OUT $\vee\, (op =$ CAS $\wedge\, t_r = t)$ **then**
45: *notify_blocked*(t)
46: **end if**

upon *receive*(p,$\langle op,\bar{t}\rangle$)
47: **if** $op =$ RD $\vee\, op =$ RDP **then**
48: $t_r \leftarrow ts.rdp(\bar{t})$
49: **end if**
50: **if** $t_r =$ NO_MATCH $\wedge\, op =$ RD **then**
51: *add_blocked*(op,p_i,$\bar{t}$)
52: **else**
53: *send*(p_i,$\langle$RESP,$t_r\rangle$)
54: **end if**

O uso de replicação Máquina de Estados garante alguns dos atributos de segurança de funcionamento. O sistema é disponível enquanto f ou menos servidores falhem; a integridade e a confiabilidade são mantidas também enquanto f ou menos servidores forem falhos devido a votação realizada para determinar o resultado de uma operação difundida aos servidores.

O protocolo de replicação do lado cliente e do servidor, é apresentado no algoritmo 2.

A parte cliente do protocolo (linhas 1-25) é muito simples. As operações que modificam o estado do espaço de tuplas são executadas através de uma chamada direta ao procedimento *execute_op*

[3]Uma possível função abstrata leria localmente todas as tuplas que combinam com o molde passado e escolheria uma destas tuplas de forma determinista (p.ex. a menor usando uma função de comparação qualquer).

passando o nome da operação a ser executada (OUT, RDP, INP, CAS, RD ou IN) e seus argumentos (uma tupla, um molde, ou ambos no caso do *cas*). Neste procedimento, o cliente envia a requisição da operação para todos os servidores usando a difusão com ordem total (linha 18) e espera respostas dos servidores até que algum resultado para a operação possa ser extraído do vetor *R* (linhas 19-25). Conforme discutido na seção 2.4.1, nos protocolos de replicação Máquina de Estados tolerantes a faltas bizantinas usuais, uma resposta é escolhida se ela é retornada por $f+1$ servidores distintos, garantindo assim que a resposta tenha sido produzida por pelo menos um servidor correto (e portanto será produzida por todos). Entretanto, se a camada de confidencialidade prevista em nossa arquitetura for utilizada, uma série de requisitos adicionais devem ser observados na extração do resultado. A forma de extração de resultado com essa camada será apresentada na seção 5.6.1 do capítulo sobre confidencialidade.

Para as operações de leitura (RD, RDP) uma pequena otimização foi incluída do lado cliente: antes de enviar a requisição através do protocolo de difusão com ordem total, o cliente difunde a requisição ponto a ponto aos servidores e espera $n-f$ respostas (linhas 7-12). Se as $n-f$ respostas recebidas forem iguais (linha 13), então esta é a resposta da leitura (linha 14), caso contrário a leitura é feita utilizando o protocolo usual (linha 16).

O lado servidor do protocolo (linhas 26-54) de replicação é ativado quando uma mensagem é entregue pelo protocolo de difusão com ordem total. Na primeira parte do algoritmo, a operação requisitada é invocada na camada superior ts do espaço de tuplas. Esta chamada chega ao *skeleton* do espaço de tuplas da réplica (ver figura 3.4). Note que as operações bloqueantes nunca são invocadas na camada superior para evitar que o servidor fique bloqueado. Quando estas operações são invocadas, suas versões não bloqueantes são executadas (linhas 35 e 37) e, se elas não são bem sucedidas devido à não existência de uma tupla que combine com o molde requisitado ($t_r = \text{NO\_MATCH}$), a operação é armazenada em um conjunto de operações pendentes (linhas 39-40). Se a operação requisitada é executada, sua resposta é enviada ao cliente (linha 42). Finalmente, se alguma tupla é inserida em ts, o procedimento *notify_blocked* verifica se alguma operação pendente deve ser executada (linhas 44-45).

Ainda do lado servidor, é interessante notar que apenas requisições de operações de leitura são aceitas de outro modo que não através da difusão com ordem total (linha 54). Isto impede que clientes maliciosos alterem o estado do protocolo enviando operações de atualização que não respeitam ordem total.

3.5.3 Controle de Acesso

O controle de acesso têm sido descrito como um mecanismo de segurança fundamental para espaço de tuplas [26, 44, 127]. A abordagem proposta em nossa arquitetura não define nenhum tipo de modelo de controle de acesso específico para controlar o acesso às tuplas e espaços, permitindo que diferentes implementações possam ser utilizadas. Por exemplo, listas de controle de acesso (ACLs) podem ser usadas em ambientes fechados enquanto algum tipo de controle de acesso baseado em papéis (RBAC) pode ser empregado em ambientes colaborativos abertos [122]. Para acomodar estes

diferentes mecanismos, o controle de acesso do espaço de tuplas com segurança de funcionamento é definido em termos de credenciais abstratas: cada espaço de tuplas TS tem um conjunto de credenciais requeridas C^{TS} e cada tupla t tem dois conjuntos de credenciais requeridas C^t_{rd} e C^t_{in}. Para inserir uma tupla no espaço TS, um cliente precisa apresentar credenciais que satisfazem C^{TS}. De forma análoga, para ler (resp. remover) a tupla t do espaço, um cliente precisa apresentar credenciais que satisfazem C^t_{rd} (resp. C^t_{in}).

As credenciais necessárias para inserir uma tupla em TS (C^{TS}) são definidas pelo administrador que cria e configura este espaço enquanto as credenciais necessárias para leitura e remoção de uma tupla t (C^t_{rd} e C^t_{in}) são definidas pelo processo que a inseriu no espaço. Se ACLs são usadas, podemos ter, por exemplo, $C^{TS} = \{p_1, p_2\}$. Isto significa que o conjunto de processos que podem inserir tuplas em TS são os listados em C^{TS}. Da mesma forma, usando ACLs as credenciais C^t_{rd} e C^t_{in} associadas com a tupla t seriam conjuntos de processos com capacidade de ler e remover t, respectivamente.

A implementação de controle de acesso em nossa arquitetura é feita através das camadas de controle de acesso nos clientes e nos servidores, como mostrado na figura 3.4. Do lado cliente, quando uma operação para inserção da tupla t (*out* ou *cas*) é invocada, as credenciais associadas à tupla C^t_{rd} e C^t_{in} são concatenadas nela como dois campos públicos (ver capítulo 5). Além disso, as credenciais do cliente são adicionadas como outro campo público da tupla, mesmo no caso da tupla ser um molde usado em operações de leitura/remoção.

Na camada do lado servidor, cada servidor correto testa se a operação pode ser executada através da comparação das credenciais fornecidas pelo cliente com as requeridas pela operação. Se a operação é uma inserção da tupla t, a credencial do cliente (concatenada a t) precisa satisfazer as credenciais requeridas pelo espaço. Se esta condição é satisfeita, t é inserida no espaço (com suas credenciais requeridas associadas C^t_{rd} e C^t_{in} como campos públicos). Se a operação requisitada é um leitura (remoção), as credenciais associadas com o molde $\bar{t}$ enviado na requisição precisam ser suficientes para leitura (remoção) de uma tupla que combine com este molde. Na prática, a camada de controle de acesso no servidor deverá ler várias tuplas do espaço de tuplas e tentar encontrar uma que possa ser lida/removida pelo cliente. Esta leitura de várias tuplas pode ser suportada diretamente pelo espaço de tuplas (ex. a operação *scan* do TSPACES [121]) ou deve ser implementada no *skeleton*. Se a credencial apresentada pelo cliente é suficiente para ler/remover várias tuplas que combinam com o molde passado, a escolha de qual delas será o resultado da operação deve ser feita de forma determinista em todos os servidores a fim de garantir que todos os corretos retornem a mesma tupla.

Note que um servidor falho pode retornar uma tupla para um cliente que não apresenta uma credencial que o qualifique para ler esta tupla. Entretanto, o mecanismo de confidencialidade não permite que uma tupla retornada por menos de $f+1$ servidores distintos seja lida, assim servidores maliciosos não podem revelar o conteúdo de uma tupla a clientes não autorizados. Um servidor falho pode também armazenar uma tupla inserida por um cliente que não tem permissão para realizar inserção. Esta tupla não será lida por processos corretos pela mesma razão.

3.5.4 Verificação de Políticas

No capítulo 6 será visto que espaços de tuplas aumentados com suporte a um controle de acesso mais refinado, baseado em políticas de granularidade fina, apresenta propriedades interessantes para a coordenação de clientes sujeitos a faltas bizantinas. Nesta seção apresentamos a camada de verificação de políticas e sua integração com nosso espaço de tuplas com segurança de funcionamento, conforme descrito na figura 3.4.

A idéia por trás da **verificação de políticas** é fazer com que as interações sobre o espaço de tuplas respeitem uma série de regras que definem as interações corretas entre os processos e o espaço de tuplas. Estas regras são dependentes da aplicação. As regras definidas nestas políticas podem liberar ou negar a execução de uma operação no espaço baseando-se em três informações básicas: a identidade do processo que a invoca; a operação e seus argumentos; e as tuplas presentes no espaço. A aplicação deste tipo de política para garantir segurança em espaços de tuplas foi proposta inicialmente em [95].

Em termos de implementação, assumimos que um espaço de tuplas tem uma única política de acesso, que é definida durante a criação do espaço por quem quer que o esteja criando. Quando uma operação é recebida por um servidor, a camada de verificação de política verifica se a operação satisfaz a política de acesso definida para o espaço. Este tipo de verificação é usualmente feita em um monitor de referência implementado de tal forma que seja impossível acessar o espaço de tuplas sem passar por ele [6]. Entretanto, em nosso espaço de tuplas replicado consideramos que os servidores corretos verificam a política de acesso, enquanto os servidores falhos podem agir arbitrariamente. Esta verificação é uma simples computação local da condição expressa na regra para a operação invocada. Quando uma operação é rejeitada, o servidor retorna um código de erros ao processo cliente. Clientes corretos só aceitam a rejeição de uma operação de uma operação se recebem $f+1$ respostas com o mesmo código de erro.

3.5.5 Custos da Segurança de Funcionamento

Esta seção faz uma análise dos custos de se implementar a arquitetura proposta para espaço de tuplas com segurança de funcionamento. Existem dois custos significativos relacionados aos mecanismos definidos para a arquitetura (camadas): o custo da replicação e o custo da confidencialidade. Nesta seção são apresentados os custos da replicação, já que o mecanismo de confidencialidade será discutido extensivamente no capítulo 5. Os mecanismos de verificação de políticas e controle de acesso incorrem em pouco acréscimo no desempenho geral do sistema, já que envolvem apenas a avaliação local de algumas poucas regras no processamento de cada requisição ao espaço.

Os principais custos do protocolo de replicação podem ser medidos em termos do *(i.)* número total de servidores requeridos para tolerar até f servidores falhos, e *(ii.)* o atraso imposto pelo protocolo em execuções livres de falha. Os valores para ambas estas métricas advêm diretamente do algoritmo de difusão com ordem total tolerante a faltas bizantinas utilizado, neste caso, o algoritmo baseado no PAXOS, descrito na seção 2.3.2. Este algoritmo requer $n \geq 3f+1$ servidores para tolerar até f

falhas bizantinas. Este valor corresponde a resistência ótima para este tipo de algoritmo [36, 43]. A difusão com ordem total implementada por este protocolo requer apenas três passos de comunicação em execuções muito favoráveis (ver figura 2.3). Outras soluções requerem mais passos [36, 91].

Uma outra característica inerente do protocolo de difusão com ordem total usado na implementação da replicação é sua complexidade de mensagens: $O(n^2)$. Esta complexidade é a melhor conhecido para protocolos de acordo tolerantes a faltas bizantinas. A tabela 3.4 apresenta o custo de cada uma das operações implementadas pelo espaço de tuplas em execuções muito favoráveis.

Operação	Complexidade de Mensagens	Passos de Comunicação
out	$\mathbf{O(n^2)}$	**4**
rdp	$\mathbf{O(n)}/O(n^2)$	$\mathbf{2}/6$
inp	$\mathbf{O(n^2)}$	**4**
cas	$\mathbf{O(n^2)}$	**4**

Tabela 3.4: Custo das operações do espaço de tuplas aumentado.

Apenas as operações não bloqueantes parecem na tabela. Note que dois valores aparecem em cada campo da linha correspondente a operação *rdp*. Os primeiros valores das métricas para esta operação correspondem a execução da leitura otimista, quando a requisição de leitura é feita sem o uso de difusão com ordem total. A leitura se completa nesta fase sempre que não houverem escritas de tuplas que combinam com o molde usado na operação concorrentemente a leitura.

3.6 Trabalhos Relacionados

Existem vários trabalhos que tentam melhorar a segurança de funcionamento do modelo de coordenação por espaços de tuplas através do uso de mecanismos de segurança ou tolerância a faltas. Além disso, vários artigos têm proposto a integração de mecanismos de segurança em espaços de tuplas. Estas propostas estão se tornando mais e mais relevantes na medida em que este modelo de coordenação vem mudando sua base de uso de aplicações paralelas executando em *clusters* de alto desempenho, para sistemas distribuídos heterogêneos abertos, em que ameaças de segurança são comuns. Entre as propostas já publicadas, algumas tentam reforçar políticas de segurança dependentes da aplicação [95], enquanto outras provêem integridade e confidencialidade usando controle de acesso em nível de espaço de tuplas [44] (controlam que processos podem executar que operações no espaço de tuplas), em nível de tupla [127] (controlam que processos podem ler e/ou remover uma tupla) ou ambos [26]. Todos estes trabalhos consideram espaços implementados em servidores centralizados **confiáveis e seguros**, não considerando a possibilidade de falha (benigna ou maliciosa) destes servidores. A arquitetura proposta neste capítulo faz uso das melhores idéias propostas neste trabalhos sobre segurança em espaços de tuplas, provendo tanto controle de acesso em nível de espaço de tuplas e de tupla (seção 3.5.3) como verificação de políticas de acesso de granularidade fina dependentes de aplicação (seção 3.5.4). Além disso, esses mecanismos foram adaptados para funcionar em um espaço de tuplas replicado tolerante a faltas bizantinas e com suporte a confidencialidade.

Alguns dos trabalhos sobre tolerância a faltas em espaço de tuplas tentam incrementar a segurança

de funcionamento da infra-estrutura de coordenação – o espaço de tuplas – usando replicação. Existe uma solução completa baseada em replicação Máquina de Estados, o sistema FT-LINDA [11], que se baseia no uso de um sistema de comunicação de grupo para implementar difusão com ordem total de requisições entre as réplicas do espaço. Um outra solução, proposta em [128], utiliza sistemas de quóruns para manter o espaço de tuplas replicado. O problema com estas duas soluções é que ambas toleram apenas faltas por parada, não considerando faltas maliciosas. A arquitetura apresentada neste capítulo, por outro lado, considera a possibilidade de existência de servidores maliciosos que podem agir ativamente contra o sistema.

Existem também alguns trabalhos que focam na provisão de mecanismos de tolerância a faltas em nível de aplicação, provendo funcionalidades como suporte a transações [71]. Estes sistemas provêem as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) [15], garantindo que, quando uma seqüência de operações no espaço de tuplas é executada como uma transação, ou todas ou nenhuma das operações da seqüência são executadas. Sistemas como o TSPACES [121] e o JAVASPACES [120] utilizam esta técnica para prover algum suporte a tolerância a faltas para aplicações. Existe uma solução alternativa para tolerância a faltas em nível de aplicação que faz uso de unidades de coordenação ao invés de transações [111]. Uma unidade de coordenação é um trecho de código que é enviado para ser executado no servidor do espaço de tuplas, garantindo atomicidade e consistência em sua execução mesmo em caso de falha no processo que o enviou. Esta abordagem é dita mais poderosa que o suporte a transações, no entanto, a introdução de código móvel no modelo de coordenação por espaço de tuplas pode trazer consigo vários problemas de segurança. Caso este tipo de suporte seja provido em um sistema sujeito a faltas bizantinas, uma série de mecanismos de segurança devem ser desenvolvidos para evitar que processos falhos enviem código malicioso para ser executado no servidor do espaço de tuplas. Reconhecemos a necessidade de suporte em nível de aplicação para tolerância a faltas, no entanto, a arquitetura apresentada neste capítulo não provê mecanismos deste tipo já que em sistemas sujeitos a faltas bizantinas eles são problemáticos. Por exemplo, se transações forem usadas, um processo malicioso pode iniciar e abortar uma transação continuamente objetivando atacar a vivacidade da aplicação. Atualmente, nossa abordagem consiste em desenvolver algoritmos que não necessitam deste tipo de suporte, como os apresentados no capítulo 6.

3.7 Conclusões do Capítulo

Neste capítulo foi definido o conceito de coordenação e os diversos níveis de desacoplamento que um modelo de coordenação pode prover. A partir daí, definimos formalmente o modelo de coordenação por espaços de tuplas, e quais atributos deve atender um espaço de tuplas que suporta segurança de funcionamento. A segunda parte do capítulo tratou da definição de uma arquitetura integrada para espaços de tuplas com segurança de funcionamento. Esta arquitetura baseia-se no uso de espaços de tuplas locais (não replicados) e de um algoritmo de replicação Máquina de Estados tolerante a faltas bizantinas. A construção satisfaz uma série de importantes atributos de segurança de funcionamento como confiabilidade, integridade, disponibilidade e confidencialidade (conforme será descrito no capítulo 5).

O próximo capítulo define outras construções para espaços de tuplas, fundamentados na abordagem de sistemas de quóruns bizantinos.

Capítulo 4

Espaços de Tuplas baseados em Sistemas de Quóruns Bizantinos

No capítulo anterior apresentamos uma arquitetura baseada em replicação Máquina de Estados para espaço de tuplas tolerante a faltas Bizantinas. Esta construção se baseia no uso direto de um protocolo de difusão com ordem total para manter o estado do espaço de tuplas nas diversas réplicas idênticos após a execução de um mesmo conjunto de operações no espaço replicado. Neste capítulo, uma abordagem diferente é explorada, e duas outras construções para espaço de tuplas tolerantes a faltas bizantinas, só que baseadas em sistemas de quóruns bizantinos, são apresentadas.

4.1 Contexto e Motivação

Conforme já apresentado no capítulo anterior, o espaço de tuplas é um objeto de memória compartilhada com número de consenso igual a 2 [117]. Isto significa que este objeto pode ser usado para resolver consenso entre no máximo dois processos em sistemas assíncronos, mesmo na presença de falhas por parada [67]. Desta forma, ele não pode ser implementado usando apenas sistemas de quóruns (que não requerem premissas temporais), pois de outra forma teríamos um objeto que pode resolver consenso em um sistema assíncrono, o que contraria a impossibilidade FLP[1] [56]. Tendo em vista esta limitação, neste capítulo identificamos as operações do espaço de tuplas que requerem protocolos mais fortes, e mostramos como implementá-las usando o protocolo de consenso PAXOS BIZANTINO [36, 91, 130] (ver seções 2.3.1 e 2.3.2).

A filosofia[2] por trás das construções de espaço de tuplas aqui apresentadas é de que operações simples (leitura e escrita) são implementadas por protocolos para sistemas de quóruns eficientes, enquanto protocolos mais custosos baseados em algoritmos de acordo são usados para implementar

[1] Os sistemas de quóruns só podem implementar registradores, que não possuem poder de sincronização algum (número de consenso igual a 1 [67]), e portanto não podem ser usados para resolver o problema de consenso. Isto explica o porque dos sistemas de quóruns poderem implementar registradores em sistemas completamente assíncronos.

[2] Não chamamos de modelo ou técnica de replicação porque, até o momento, não foi formalizada uma forma de usá-la na implementação de qualquer serviço.

operações mais complexas de leitura-escrita[3]. Estes protocolos baseados em acordo são mais custosos em pelo menos dois sentidos. Primeiramente, eles têm uma maior complexidade de mensagens (em geral $O(n^2)$ ao invés de $O(n)$). Além disso, enquanto protocolos para sistemas de quóruns bizantinos podem ser estritamente assíncronos, os protocolos de acordo requerem algum tipo de incremento no modelo de sistema adotado (premissas de sincronismo ou oráculos) já que o problema do consenso não pode ser resolvido em sistemas assíncronos [56]. Nesta trabalho adotamos premissas de sincronismo extras para o sistema, conforme descrito na seção 2.1.3.

Neste capítulo apresentamos duas construções para espaço de tuplas tolerantes a faltas bizantinas baseadas na filosofia descrita acima. A primeira, chamada **BTS (*Bizantine Tuple Space*)**, requer $n \geq 3f+1$ servidores e apresenta protocolos simples e modulares, oferecendo uma semântica mais fraca. A segunda construção, chamada **LBTS (*Linearizable Bizantine Tuple Space*)**, requer $n \geq 4f+1$ servidores e apresenta protocolos extremamente eficientes, satisfazendo a propriedade de **linearização** [69]. Além disso, tanto o BTS quanto o LBTS satisfazem a propriedade de **liberdade de espera** [67], onde todas as invocações realizadas por clientes corretos terminam por receber respostas, independentemente do estado dos outros clientes. Ambas construções apresentam importantes melhorias, em termos de complexidade de mensagens e número de passos de comunicação, quando comparadas com construções equivalentes baseadas em replicação Máquina de Estados.

Uma importante contribuição deste capítulo é a asserção que as operações *out* e *rdp* podem ser implementadas a partir de protocolos para sistemas de quóruns, enquanto a concretização de *inp* requer consenso. Mais precisamente, para implementar corretamente um espaço de tuplas é necessário que todas as operações *inp* sejam executadas na mesma ordem em todos os servidores. Esta característica é exatamente o que dá algum poder de sincronização ao objeto de memória compartilhada espaço de tuplas, e é responsável direta por este tipo de objeto ter número de consenso 2.

Antes de apresentarmos o BTS e o LBTS, apresentamos as condições de correção para espaços de tuplas. Estas condições são importantes pois definem precisamente que tipo de garantias as construções apresentadas devem satisfazer. A definição destas condições também é uma contribuição deste trabalho.

Os espaços de tuplas apresentados neste capítulo não suportam a operação *cas*. Nosso interesse aqui é apresentar implementações simples e eficientes para o espaço de tuplas "clássico", usando a abordagem de sistemas de quóruns bizantinos.

Para a apresentação do BTS e do LBTS, definimos primeiramente protocolos para as operações não bloqueantes (*out*, *rdp* e *inp*). A seguir provamos sua correção e as comparamos com construções baseadas em replicação Máquina de Estados que oferecem semânticas equivalentes. Ao final do capítulo, são apresentadas várias modificações e extensões que podem ser feitos nas construções.

[3]Onde o valor escrito depende do estado do objeto no momento da operação.

4.2 Condições de Correção para Espaços de Tuplas Replicados

Uma implementação de um espaço de tuplas deve prover a semântica descrita na seção 3.3.1. Nesta seção, especificamos esta semântica usando a noção de **história** [69]. Esta representação da semântica do espaço é importante pois nos permite trabalhar com implementações de espaços de tuplas usando o ferramental de sistemas distribuídos.

Uma história H modela uma execução de um sistema concorrente composto por um conjunto de processos e um objeto de memória compartilhada (o espaço de tuplas, neste trabalho). Uma história é uma seqüência finita de eventos que sinalizam invocações de operações e suas respostas.

Especificamos as propriedades do espaço de tuplas em termos de **histórias seqüenciais**, que são histórias onde o primeiro evento é uma invocação e cada invocação é diretamente seguida pela sua resposta (ou algum outro evento sinalizando o término da operação). Representamos uma história seqüencial H através de um seqüência de pares $\langle operation, response \rangle$ separados por vírgula. Uma **sub-história** S de uma história H é uma subseqüência de eventos de H. Também separamos sub-histórias por vírgulas para formar uma nova sub-história. Usamos o operador de pertinência $\in$ com significado "é sub-história de". Uma sub-história P é um **prefixo** de H se e somente se existe uma outra sub-história $S \in H$ tal que $P,S = H$.

Um conjunto de histórias $\mathscr{H}$ é dito **prefixo-fechado** se para todo $H \in \mathscr{H}$, todo prefixo de H também pertence a $\mathscr{H}$. Uma **especificação seqüencial** de um objeto é um conjunto de histórias seqüenciais prefixo-fechado para este objeto.

Uma **especificação seqüencial para espaço de tuplas** é um conjunto de histórias prefixo-fechado do espaço de tuplas em que qualquer história H e qualquer sub-história $S \in H$ satisfaz as seguintes propriedades[4]:

1. $S, \langle rdp(\bar{t}), t \rangle \in H \;\Rightarrow\; \exists \langle out(t), ack \rangle \in S$
2. $S, \langle rdp(\bar{t}), t \rangle \in H \;\Rightarrow\; \nexists \langle inp(\bar{t}'), t \rangle \in S$
3. $S, \langle inp(\bar{t}), t \rangle \in H \;\Rightarrow\; \exists \langle out(t), ack \rangle \in S$
4. $\langle inp(\bar{t}), t \rangle, S \in H \;\Rightarrow\; \nexists \langle inp(\bar{t}'), t \rangle \in S$

A primeira propriedade diz que se uma tupla t é o resultado de uma leitura, então ela foi inserida em uma operação anterior a leitura (a resposta de $out(t)$ é uma confirmação ack). A segunda propriedade define que a leitura de uma tupla não pode ocorrer posteriormente à sua remoção. A próxima propriedade define que, para que uma tupla seja removida, ela deve ter sido inserida. Finalmente, a propriedade 4 diz que não existe mais de uma remoção para uma mesma tupla. Por simplicidade, as propriedades assumem que cada tupla é única, i.e. que cada tupla é inserida no espaço no máximo uma vez.

[4] Estas propriedades também especificam as operações bloqueantes, basta para isso substituir *inp* por *in* e *rdp* por *rd*.

Idealmente as propriedades 1-4 precisam ser satisfeitas mesmo quando o espaço é acessado concorrentemente por diversos processos. Isto é garantido se o espaço de tuplas satisfaz a propriedade de **linearização** [69]. Esta propriedade diz que operações invocadas concorrentemente parecem tomar efeito instantaneamente em algum momento entre sua invocação e o retorno de sua resposta. Este instante é chamado de **ponto de serialização** [83]. Desta forma, operações concorrentes sempre podem ser ordenadas tomando-se como referência seus pontos de serialização. Em outras palavras, qualquer história concorrente de um espaço de tuplas linearizável é equivalente a alguma história seqüencial. Um espaço de tuplas que satisfaz a propriedade de linearização é dito linearizável.

As quatro propriedades enunciadas acima são propriedades de segurança (*safety*) para o espaço de tuplas. A propriedade de vivacidade (*liveness*) que estamos interessados em prover é a terminação livre de espera (*wait-free termination*) [67]. Esta propriedade define que, em qualquer execução do sistema, todo processo que invoca uma operação termina por receber uma resposta, independentemente do estado de outros processos. Uma operação que satisfaz esta propriedade é dita livre de espera. Esta propriedade só pode ser satisfeita nas operações não bloqueantes (*out*, *rdp* e *inp*). Uma invocação a uma operação bloqueante (*rd* ou *in*) pode ficar bloqueada indefinidamente se, por exemplo, nenhuma tupla que combine com o molde passado for inserida no espaço.

Note que a implementação de espaço de tuplas apresentada no capítulo anterior, baseada em replicação Máquina de Estados, implementa de forma trivial estas condições. Isto se deve ao fato de que todas as requisições são executadas na mesma ordem em todas as réplicas do espaço de tuplas e portanto, operações concorrentes são naturalmente ordenadas pelo protocolo de difusão com ordem total.

4.3 BTS

Esta seção apresenta o **BTS (Byzantine Tuple Space)** [21], a primeira (e mais simples) construção introduzida nesta tese para espaço de tuplas tolerante a faltas bizantinas baseada em sistemas de quóruns bizantinos.

4.3.1 Premissas dos Algoritmos

O BTS considera um conjunto infinito de clientes e $n \geq 3f + 1$ servidores organizados como um sistema de quóruns bizantino assimétrico (descrito na seção 2.4.2). Um número ilimitado de clientes e até f servidores podem falhar arbitrariamente.

Os algoritmos do BTS assumem que cada servidor s do sistema contém um cópia local do espaço de tuplas, denotada por T_s, e um conjunto de tuplas "a serem removidas", denotado por R_s (cujo propósito será explicado mais a frente). A fim de que as operações usuais em conjuntos possam ser empregadas na manipulação local destes dois conjuntos, assumimos que tuplas idênticas não existem. Isto pode ser implementado, por exemplo, concatenando o identificador do processo que criou a tupla

e um número de seqüência da mensagem referente a sua inserção em um campo opaco (que não pode ser lido pelos clientes) da tupla.

Assumimos também que o mecanismo de controle de acesso em nível de espaço e tuplas descrito na seção 3.5.3 é utilizadas no BTS.

4.3.2 Inserção de tuplas - *out*

O algoritmo 3 apresenta o protocolo a ser executado para a realização da operação *out*, que insere um tupla t no espaço.

Algoritmo 3 Operação *out* do BTS (processo p e servidor s).

$\{Cliente\}$	$\{Servidor\}$
procedure $out(t)$	**upon** $receive(p, \langle \text{OUT}, t \rangle)$
1: $\forall s \in Q_w, send(s, \langle \text{OUT}, t \rangle)$	2: **if** $t \notin R_s$ **then**
	3: $T_s \leftarrow T_s \cup \{t\}$
	4: **else**
	5: $R_s \leftarrow R_s \setminus \{t\}$
	6: **end if**

Devido a natureza não determinista da coordenação baseada em espaço de tuplas e a confiabilidade dos canais de comunicação assumidos em nosso modelo de sistema, um cliente não precisa esperar confirmações dos servidores para sua mensagem de inserção da tupla no espaço. Desta forma, o algoritmo de inserção consiste apenas no envio da tupla a um quórum de escrita de servidores (linha 1). Quando um servidor s recebe uma requisição de inserção para a tupla t, ele adiciona a tupla ao espaço apenas se sua remoção não estiver pendente ($t \notin R_s$), i.e. t foi removida como resultado de uma execução de *inp* quando s ainda não havia recebido esta tupla (linhas 2-3). Caso t esteja com sua remoção pendente, ela apenas é removida do conjunto R_s (linha 5). Este tipo de controle é necessário para garantir que a mesma tupla não será removida duas vezes do espaço de tuplas (ver seção 4.3.4). Uma ilustração do funcionamento deste protocolo é apresentada na figura 4.1(b). Na execução representada nesta figura, o servidor s_2 é falho (representado por uma linha de tempo tracejada).

O protocolo de escrita empregado para a operação *out* no BTS é do tipo **não confirmável** [93]. Este tipo de protocolo requer menos trocas de mensagens e pode ser usado em sistemas onde o escritor não necessita saber o momento exato em que sua escrita termina. Desta forma, é assumido que uma operação *out* é completada quando todos os servidores corretos do quórum de escrita processam a tupla a ser inserida (executam as linha 2-6 do algoritmo 3). O evento *ack*, que representa o término da operação de *out* em uma história, corresponde justamente ao momento em que a operação é completada. Na figura 4.1(b), a operação é completada quando s_1 processa a tupla inserida.

Apesar de ser um protocolo de desempenho ótimo[5], a estratégia de implementação adotada para a operação *out* apresenta uma limitação na medida em que ela oferece uma semântica **não ordenada**

[5] O protocolo requer apenas 1 passo de comunicação.

para esta operação. Esta semântica é a interpretação mais fraca definida para a operação *out* [27], e não é suficiente para implementação de qualquer computação distribuída[6].

4.3.3 Leitura de tuplas - *rdp*

A operação de leitura não destrutiva (*rdp*) de uma tupla t, que combina com um molde $\bar{t}$, é feita no BTS através do protocolo apresentado no algoritmo 4.

Algoritmo 4 Operação *rdp* do BTS (processo p e servidor s).

$\{Cliente\}$	$\{Servidor\}$
procedure $rdp(\bar{t})$	**upon** $receive(p, \langle \text{RDP}, \bar{t} \rangle)$
1: $\forall s \in U$, $send(s, \langle \text{RDP}, \bar{t} \rangle)$	12: $T_s^{\bar{t}} \leftarrow \{t \in T_s : m(t, \bar{t})\}$
2: $Replies[s_1..s_n] \leftarrow \perp$	13: $send(p, \langle \text{REP-RDP}, T_s^{\bar{t}} \rangle)$
3: **repeat**	
4: **wait** $receive(s, \langle \text{REP-RDP}, T_s^{\bar{t}} \rangle))$	
5: $Replies[s] \leftarrow T_s^{\bar{t}}$	
6: **until** $\#Replies_{\perp} \leq n - q_r$	
7: **if** $\exists t : (\#Replies_{t\in} \geq f + 1)$ **then**	
8: **return** t	
9: **else**	
10: **return** $\perp$	
11: **end if**	

O protocolo de leitura começa com o cliente acessando um quórum de leitura de servidores[7] tentando obter todas as tuplas do espaço que combinam com $\bar{t}$ (linhas 1-6). Pelo algoritmo, cada servidor acessado s responde a requisição do cliente com um conjunto $T_s^{\bar{t}}$ contendo suas tuplas que combinam com $\bar{t}$ (linhas 12 e 13). O cliente coleta esses conjuntos e os armazena no vetor de respostas *Replies*. Quando este vetor contiver respostas de um quórum de leitura de servidores (menos de $n - q_r$ servidores não responderam – linha 6), o cliente escolhe uma tupla t que aparece em pelo menos $f + 1$ conjuntos de tuplas retornadas pelos servidores (denotado por $\#Replies_{t\in} \geq f + 1$ – linhas 7 e 8). Se não existe uma tupla como esta, o símbolo especial $\perp$ é retornado sinalizando que não existe uma tupla que combine com este molde no espaço (linha 10).

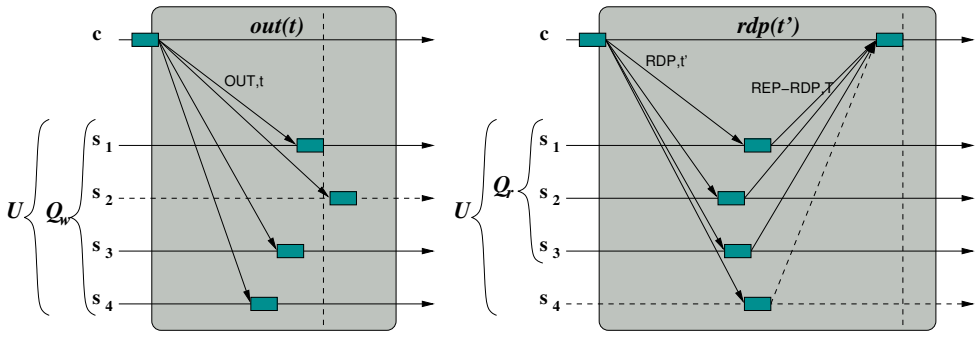
A figura 4.1(b) ilustra o funcionamento do protocolo de leitura. Nesta figura é possível verificar que o processo cliente escolhe uma resposta para a operação mesmo não recebendo resposta de s_4 (que é falho, sua omissão de resposta é representada pela seta tracejada).

4.3.4 Leitura destrutiva de tuplas - *inp*

A operação *inp* é a que possui implementação mais complexa dentre todas as operações suportadas pelo BTS. Esta complexidade é um resultado direto do requisito de que uma mesma tupla não

[6] Mais formalmente: um espaço de tuplas que suporta apenas semântica não ordenada para operação *out* não constitui uma linguagem *Turing-powerful* [27].

[7] No algoritmo, o cliente envia a requisição a todos os servidores (linha 1) e espera respostas de um quórum de leitura (linha 6). Uma implementação melhor para este acesso é o cliente enviar a requisição a um quórum e, periodicamente, enviar a requisição também a outros servidores, até que sejam coletadas respostas de um quórum completo de servidores.



(a) Operação *out*. (b) Operação *rdp*.

Figura 4.1: Execuções dos protocolos *out* e *rdp* do BTS.

pode ser removida por duas invocações distintas de *inp*. Esta condição implica na resolução de uma exclusão mútua entre os processos que tentam remover tuplas do espaço ou, colocando de outra forma, numa ordenação total entre todas as remoções.

Para implementar esta operação usando sistemas de quóruns bizantinos, dividimos o problema em dois sub-problemas e utilizamos duas primitivas de acordo largamente conhecidas:

- **Exclusão Mútua**: O problema de exclusão mútua considera um conjunto de processos acessando um recurso compartilhado, de tal forma que os acessos ao recurso sejam feitos sem concorrência. No protocolo para a operação *inp* do BTS, utilizamos um algoritmo de exclusão mútua para garantir que apenas um único processo por vez vai tentar remover uma tupla do espaço, i.e. a operação de leitura destrutiva é consolidada dentro da seção crítica do processo cliente. As primitivas de exclusão mútua utilizadas no BTS são implementadas através da aplicação direta do algoritmo de difusão com ordem total baseado no PAXOS BIZANTINO, conforme descrito na seção 2.3.3.

- **Consenso**: Conforme apresentado na seção 2.3.1, no algoritmo PAXOS os processos participantes do protocolo são divididos em três conjuntos (que podem se sobrepor) de acordo com os diferentes papéis representados por eles: **proponentes**, que propõem valores, **aceitantes**, que juntos estabelecem acordo para a escolha de um único valor proposto, e **aprendizes**, que aprendem o valor escolhido [91]. O algoritmo garante que um único valor proposto por um proponente será escolhido por todos os aceitantes e aprendido por todos os aprendizes. No protocolo *inp* do BTS, o PAXOS é usado para garantir que o processo dentro da seção crítica (cuja entrada é controlada pelo algoritmo de exclusão mútua) e os servidores concordem a respeito da tupla sendo removida.

Usando estas duas abstrações e a operação *rdp* (definida na seção anterior), a operação *inp* pode ser implementada de forma muito elegante, como apresentado no algoritmo 5.

No protocolo, o PAXOS BIZANTINO é usado através da invocação das funções *paxos* passando cinco argumentos: o primeiro processo a propor um valor, o conjunto de proponentes P, o conjunto

Algoritmo 5 Operação *inp* do BTS (processo p e servidor s).

```
{Cliente}
procedure inp(t̄)
 1: repeat
 2:    enter()
 3:    t ← rdp(t̄)
 4:    if t = ⊥ then
 5:       exit()
 6:       return ⊥
 7:    end if
 8:    d ← paxos(p,P,A,L,t)
 9:    exit()
10: until d = t
11: return t
```

```
{Servidor}
upon hasLock(p)
12: d ← paxos(p,P,A,L,⊥)
13: if d ≠ ⊥ then
14:    if d ∉ T_s then
15:       R_s ← R_s ∪ {d}
16:    end if
17:    T_s ← T_s \ {d}
18: end if
```

de aceitantes A, o conjunto de aprendizes L e o valor proposto. O valor de retorno desta função é o valor escolhido pelos aceitantes e aprendido pelos aprendizes, que chamamos de valor decidido. Os primeiros 4 parâmetros nas chamadas do PAXOS são os mesmos tanto nos clientes quanto nos servidores (linhas 8 e 12): p (o primeiro a propor é o processo na seção crítica), $P \triangleq \{p\} \cup U$ (os proponentes são p e os servidores), $A \triangleq U$ (os servidores são os aceitantes) e $L \triangleq \{p\} \cup U$ (os aprendizes são p e os servidores). Os valores propostos por p (o processo na seção crítica) e os servidores são diferentes, e serão explicados mais adiante. Note que o cliente que tenta remover a tupla **não** é incluído no conjunto de aceitantes. Desta forma, independentemente do cliente ser falho ou não, o limite $|A| = n \geq 3f + 1$ é mantido [91].

No protocolo do algoritmo 5, o processo p, que tenta remover uma tupla t que combina com o molde $\bar{t}$, inicia a operação tentando obter acesso exclusivo ao espaço de tuplas, i.e. acessar sua seção crítica (linha 2). A seguir, p lê t (linha 3) e, supondo que esta tupla exista, executa o algoritmo PAXOS visando removê-la (linha 8). Quando o PAXOS termina, p e todos os servidores aprendem sobre a remoção (ou não) de t. Após aprender o resultado, p sai de sua seção crítica (linha 9). Se o valor decidido pelo algoritmo PAXOS é t, então a operação termina e t é a tupla removida (linha 11). Caso contrário, p executa o algoritmo novamente (laço das linha 1-10). Se não existe uma tupla no espaço que combine com $\bar{t}$, p sai de sua seção crítica e retorna $\perp$ (linhas 4-7).

Quando um servidor s dá permissão a um processo p para que este entre em sua seção crítica (p trava o espaço de tuplas – $hasLocked(p) = true$), s executa o PAXOS (linha 12) assumindo p como primeiro proponente e propondo $\perp$. Se o valor decidido é $t \neq \perp$ (linha 13) então t é a tupla a ser removida pelo processo p. Se t não existe na cópia local do espaço de tuplas, ela é adicionada ao conjunto de remoções pendentes R_s (linhas 14-16), garantindo que esta tupla não será removida novamente. Caso contrário, t é removida da cópia local do espaço de tuplas (linha 17).

Conforme já discutido, para que um processo p remova uma tupla t do espaço provido pelo BTS, o valor decidido no PAXOS precisa ser t. Abrindo a "caixa-preta" do PAXOS podemos perceber que isto só acontece quando t proposta por p chega em um número suficiente de servidores[8] antes que

[8]Pelo menos $\lceil \frac{n+f}{2} \rceil$ servidores, garantindo que a maioria dos processos corretos aceitou o valor proposto.

estes suspeitem de p e tentem eleger um novo líder. A correção do algoritmo é derivada diretamente dessa observação, do modelo de sistema adotado (sincronia terminal) e da repetição da execução do procedimento pelo cliente até que uma tupla seja removida ou ele descubra que não existe tupla que combine com seu molde no espaço.

Da forma como o algoritmo 5 está descrito, o protocolo para a operação *inp* pode ainda ficar bloqueado já que um processo pode executar a linha 8 e falhar antes de executar a linha 9, não liberando o espaço de tuplas travada para acesso de outros processos. Para resolver este problema temos de remover a liberação explícita do recurso (saída de sua seção crítica - linha 9) do cliente e incluí-la como uma liberação local em cada servidor depois deste completar sua parte no protocolo (incluindo o comando $unlock()$ após a linha 18). Com esta modificação simples, o algoritmo de exclusão mútua garante que um cliente que entra em sua seção crítica terá a oportunidade de propor uma tupla para remoção através de uma execução do PAXOS, saindo de sua seção crítica após aprender o valor decidido. Desta forma, outros processos terão chance de acessar este tipo de tupla no espaço tão logo a execução do PAXOS termine, e as operações *inp* executadas no espaço ocorrerão na mesma ordem em todos os servidores corretos.

Outra modificação que pode ser feita no algoritmo visando sua otimização, é a execução da operação de leitura (linha 3) juntamente com a primitiva *enter* do protocolo de exclusão mútua (linha 2). Um cliente p pode enviar o molde $\bar{t}$ para os servidores juntamente com a requisição para entrada em sua seção crítica e cada servidor s pode retornar o conjunto $T_s^{\bar{t}}$ juntamente com a permissão para que p entre em sua seção crítica (quando essa permissão for dada). Esta otimização elimina um acesso ao sistema de quóruns, diminuindo a quantidade de mensagens executadas no protocolo e, conseqüentemente, o tempo requerido para execução da operação *inp*. As mensagens concedendo permissão ao cliente e enviando as tuplas que combinam com o molde passado devem ser assinadas pelos servidores. Desta forma um cliente pode montar uma **justificativa** para provar aos servidores que *(i.)* ele recebeu permissão para acessar sua seção crítica e *(ii.)* a tupla proposta para remoção foi lida do espaço de tuplas. Este conjunto de mensagens recebido é enviado juntamente com a tupla proposta na linha 8.

A figura 4.2 mostra uma execução favorável do protocolo *inp*.

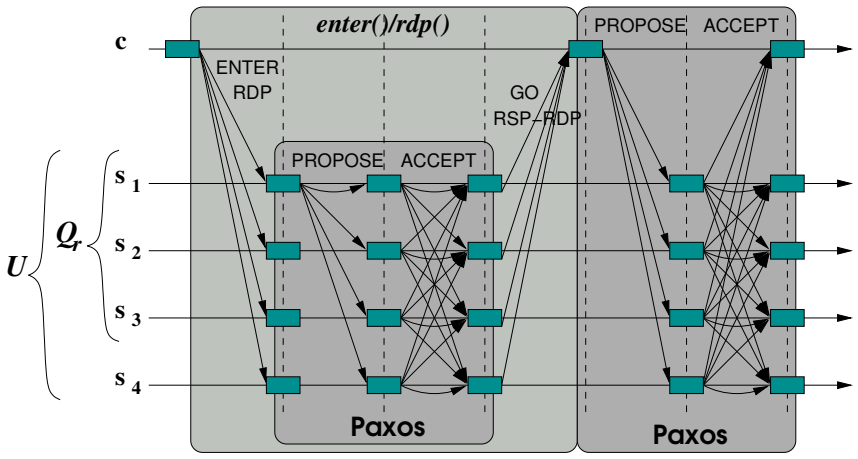


Figura 4.2: Execução favorável do protocolo *inp* ($n = 4$ e $f = 1$).

A figura 4.2 mostra que o protocolo *inp* também é não confirmável (como o protocolo *out* apresentado na seção 4.3.2): um processo p que remove a tupla t aprende que ela será removida apenas por ele e por nenhum outro processo devido as propriedades de segurança do PAXOS [91], porém, p não sabe quando todos os servidores corretos vão remover t de seus espaços de tuplas locais, já que isso acontece apenas após a aprendizagem de t (linhas 14-17).

4.3.5 Correção do BTS

Nesta seção discutiremos informalmente a correção dos protocolos do BTS. A segurança (*safety*) dos protocolos é garantida pelo fato de que uma tupla t não pode ser **sempre** lida no BTS se *(i.)* sua escrita no espaço de tuplas não está completa (satisfazendo condições 1 e 3 descritas na seção 4.2) ou *(ii.)* sua remoção está completa (satisfazendo condições 2 e 4 descritas na mesma seção). *(i.)* é decorrente do fato de que, para que t seja lida, ela precisa ser retornada por pelo menos $f+1$ servidores (linha 7 do algoritmo 4) de um quórum de leitura. Como cada operação *out* completada insere t em um quórum de escrita (e conseqüentemente, em um quórum de leitura composto apenas por servidores corretos), t sempre estará disponível em q_r servidores do sistema, e portanto sempre será lida. Desta forma, só podemos afirmar que uma tupla pode ser lida se sua operação de escrita tiver sido completada. A condição *(ii.)* é resultado do fato de que quando uma remoção é completada, todos os servidores corretos removem t de suas cópias locais do espaço de tuplas (ou adicionam t ao conjunto de remoções pendentes). Conseqüentemente, se t está em processo de remoção (e isso já foi confirmado para o cliente que executa a remoção) não podemos garantir que existirão $f+1$ servidores que retornarão t durante a execução de uma operação *rdp*.

Em termos da vivacidade do BTS, todas as operações implementadas pelo protocolos apresentados satisfazem terminação livre de espera. A operação *out* sempre termina em processos corretos uma vez que assumimos canais confiáveis e o protocolo do lado cliente não requer nenhuma resposta (não existe um comando **wait** no algoritmo). A operação *rdp* é livre de espera já que a propriedade de disponibilidade dos sistemas de quóruns assimétricos garante que sempre existe um quórum de leitura disponível, que vai responder a requisição de leitura enviada pelo cliente, causando o término da operação (finalizando o laço das linhas 3-6). A terminação da operação *inp* é conseqüência direta da terminação de todas as primitivas usadas na construção do protocolo correspondente: a exclusão mútua satisfaz progresso justo (ver seção 2.3.3), portanto a primitiva $enter()$ tem terminação garantida, a operação *rdp* termina (como mostrado acima), e o PAXOS sempre termina em nosso modelo de sistema.

Note que os protocolos apresentados não satisfazem a propriedade de **linearização** [69]. Isto se deve ao fato do protocolo de leitura permitir que uma leitura concorrente com uma escrita (*out* ou *inp*) retorne a tupla sendo inserida/removida (assumindo que essa tupla combina com o molde passado como argumento) ou outra tupla que já se encontrava no espaço antes da operação de escrita ser iniciada. Desta forma, as propriedades 1-3 da seção 4.2 só valem quando consideramos operações executadas seqüencialmente. A propriedade 4 é sempre respeitada, já que as execuções de *inp* ocorrem sempre uma após a outra, controladas por um algoritmo de exclusão mútua.

4.3.6 Avaliação

A tabela 4.1 apresenta o custo dos protocolos para as operações do BTS considerando duas métricas: **complexidade de mensagens** e **número passos de comunicação**. A complexidade de mensagens mede o número máximo de mensagens trocadas entre os processos. Conseqüentemente, esta métrica nos dá uma idéia sobre o uso do sistema de comunicação. Isto reflete diretamente na escalabilidade do algoritmo. Os passos de comunicação contam o número de transmissões de mensagens seqüenciais entre os processos, sendo o fator preponderante no tempo requerido para o algoritmo terminar.

As duas primeiras colunas da tabela apresentam os custos, em termos das duas métricas descritas, para as três operações suportadas pelo BTS. Para fins de comparação, as últimas duas colunas desta mesma tabela apresentam os custos dos protocolos para as mesmas operações (com as mesmas semânticas e garantias) se implementadas com replicação Máquina de Estados (usando difusão com ordem total baseada no PAXOS). A estratégia para implementação destas operações usando esta técnica é a seguinte:

- $out(t)$: O cliente envia uma requisição aos servidores com ordem total visando incluir a tupla no espaço replicado. Os servidores recebem a mensagem e incluem a tupla no espaço;
- $rdp(\bar{t})$: Como esta operação não altera o estado do espaço replicado, ela pode ser realizada da mesma forma que no sistema de quóruns (algoritmo 4);
- $inp(\bar{t})$: O cliente envia uma requisição com ordem total aos servidores. Estes recebem a requisição e escolhem de forma determinista uma tupla a ser removida e retornada ao cliente. O cliente espera $f+1$ respostas com a mesma tupla para defini-la como resposta.

Note que esta implementação resulta em um protocolo para *out* não confirmável, como no BTS. Tendo em vista que o BTS não satisfaz linearização, podemos relaxar nossa replicação Máquina de Estados de tal forma que ela também não satisfaça esta propriedade. Desta maneira, não é necessário que as leituras sejam enviadas aos servidores com ordem total, e o mesmo protocolo de leitura usado pelo BTS pode ser usado nesta implementação. Nesta seção este protocolo será chamado de WSMR-TS (*Weak State Machine Replication Tuple Space*).

Operação	BTS		WSMR-TS	
	Complex. Mens.	Passos Com.	Complex. Mens.	Passos Com.
out	$O(n)$	1	$O(n^2)$	3
rdp	$O(n)$	2	$O(n)$	2
inp	$O(n^2)$	6	$O(n^2)$	4

Tabela 4.1: Custo dos protocolos para espaço de tuplas: BTS e WSMR-TS.

Os valores da tabela 4.1 consideram apenas execuções favoráveis do sistema. Nesta tabela, os protocolos para as operações *out* e *rdp* do BTS são muito simples e por isso mesmo têm um custo reduzido em termos das duas métricas consideradas quando comparados com o protocolo para a

operação *inp*. Esta operação têm seu custo dominado pelo algoritmo PAXOS BIZANTINO, que é executado duas vezes. Desta forma, este protocolo requer vários passos de comunicação extra e têm uma complexidade de mensagens não-linear.

Observando o custo das operações em um espaço de tuplas implementado com replicação máquina de estados, podemos notar que a grande desvantagem desta abordagem, quando comparado com os protocolos baseados em quóruns do BTS, é o fato da operação *out* se tornar tão complexa quanto a operação *inp*. Esta complexidade se deve ao fato de que essas duas operações alteram o estado do espaço de tuplas, e portanto requerem o uso de difusão com ordem total.

4.4 LBTS

O BTS corresponde ao projeto de um espaço de tuplas o mais simples possível baseado na idéia de sistemas de quóruns, explorando a filosofia de que operações complexas (que não podem ser implementadas por sistemas de quóruns) usam protocolos mais elaborados baseados em algoritmos de acordo enquanto as operações simples são executadas através de algoritmos rápidos e simples. O objetivo principal nesta construção é garantir que tuplas inseridas no espaço por clientes corretos estejam disponíveis no espaço para leitura ou remoção enquanto menos de um terço dos servidores forem falhos. Apesar de cumprir este objetivo, o BTS apresenta uma série de limitações:

1. **Não confirmável**: o protocolo para a operação *out* é não confirmável e dá uma interpretação não ordenada a esta operação, tornando o modelo de coordenação não *Turing-powerful*;
2. **Não linearizável**: o BTS apresenta semântica não linearizável, i.e. operações executadas concorrentemente não podem ser ordenadas sem ferir alguma das propriedades 1-4 descritas na seção 4.2;
3. **Ineficiência da operação** *inp*: o protocolo para a operação *inp* do BTS requer mais passos de comunicação que uma difusão com ordem total (usada na replicação Máquina de Estados), mesmo em execuções favoráveis. Além disso, este protocolo sempre requer o uso de criptografia assimétrica para assinatura de mensagens, o que impõem uma severa penalidade no desempenho do protocolo [22]. Isto limita as vantagens do uso do BTS em detrimento a uma solução baseada em replicação Máquina de Estados.

O LBTS (*Linearizable Byzantine Tuple Space*), apresentado nesta seção, é uma construção de espaço de tuplas baseada na mesma filosofia do BTS, que no entanto não sofre de nenhuma das três limitações listadas acima.

4.4.1 Premissas dos Algoritmos

O LBTS considera um conjunto infinito de clientes e $n \geq 4f + 1$ servidores organizados como um sistema de quóruns bizantino f-mascaramento (descrito na seção 2.4.2). Um número ilimitado de clientes e até f servidores podem falhar arbitrariamente.

Todos os algoritmos do LBTS assumem que cada servidor s do sistema contém um cópia local do espaço de tuplas, denotada por T_s, um contador de tuplas removidas do conjunto T_s, denotado por r_s, e um conjunto onde são armazenadas as tuplas removidas R_s. Este conjunto é usado para evitar que uma tupla seja removida mais de uma vez. Note que o conjunto R_s aqui tem uma função diferente da que exerce no BTS. Além disso, existe um conjunto L_s que contém os clientes que estão realizando uma leitura no espaço. Este conjunto é usada para garantir a terminação da operação de leitura e será explicado na seção que apresenta o protocolo correspondente a esta operação.

Assumimos várias simplificações para deixar os algoritmos mais legíveis. Em primeiro lugar, assim como no BTS, assumimos que tuplas idênticas não existem. Desta forma podemos utilizar operadores de conjuntos para manipular T_s e R_s. Assumimos também que qualquer mensagem enviada por um servidor s que não esteja corretamente assinada por s é simplesmente descartada. Outra premissa é que as mensagens carregam *nonces* para evitar ataques de *replay*. Assim como no BTS, assumimos também que as camadas de controle de acesso em nível de espaço e tuplas descritas na seção 3.5.3 também são utilizadas no LBTS. Finalmente, assume-se que as reações dos servidores ao recebimento de uma mensagem são atômicas, nunca sendo preemptados (ex. linhas 3-6 no algoritmo 6).

4.4.2 Inserção de tuplas - *out*

O protocolo para a operação *out* consiste apenas em um acesso simples ao sistema de quóruns. O algoritmo 6 apresenta este protocolo.

Algoritmo 6 Operação *out* do LBTS (cliente p e servidor s).

	{SERVIDOR}
{CLIENTE}	**upon** $receive(p, \langle \text{OUT}, t \rangle)$
procedure $out(t)$	3: **if** $t \notin R_s$ **then**
1: $\forall s \in U$, $send(s, \langle \text{OUT}, t \rangle)$	4: $\quad T_s \leftarrow T_s \cup \{t\}$
2: **wait until** $\exists Q \in \mathcal{Q}, \forall s \in Q$, $receive(s, \langle \text{ACK-OUT} \rangle)$	5: **end if**
	6: $send(p, \langle \text{ACK-OUT} \rangle)$

Quando um cliente p quer inserir a tupla t no espaço de tuplas, ele deve primeiro enviar t a todos os servidores (linha 1) e então esperar por confirmações de um quórum completo de servidores (linha 2). Do lado dos servidores, se a tupla presente na requisição de inserção não tiver sido removida previamente (linha 3), ela é inserida na cópia local do espaço de tuplas (linha 4) e uma confirmação é enviada ao cliente (linha 6).

Assim como no BTS, um cliente falho pode inserir uma tupla em um sub-conjunto dos servidores que não constitui um quórum completo. Neste caso, dizemos que a tupla inserida é uma **tupla fantasma**. As operações de leitura – *rdp* e *inp* – são capazes de ler/remover esta tupla se ela for inserida em pelo menos $f+1$ servidores (ver próximas seções).

Este protocolo apresenta várias características interessantes. Em primeiro lugar, ele é sempre **rápido** (termina em dois passos de comunicação) [53]. Adicionalmente, o protocolo é **confirmável** [93], i.e. um processo executando *out* sabe quando a operação termina. Conseqüentemente, ele provê

uma semântica ordenada para a operação *out*, o que faz com que a linguagem de coordenação provida pelo LBTS seja *Turing-powerful* [27]. Esta última característica cobre a limitação 1 do BTS.

4.4.3 Leitura de tuplas - *rdp*

A operação *rdp* é implementada pelo protocolo apresentado no algoritmo 7. Este protocolo é mais complexo que o de inserção por duas razões. Primeiro, ele é baseado no padrão de projeto *Listener* [58], usado aqui de maneira similar a introduzida em [92]. Neste padrão de comunicação, os processos leitores se registram nos servidores e são avisados sempre que o estado do objeto de interesse é alterado. Em segundo lugar, se uma tupla que combina com o molde passado na leitura é encontrada, o processo leitor pode ter de escrevê-la novamente no espaço de tuplas (*write back*) para garantir que ela poderá ser lida em operações de leitura posteriores. Este é o mecanismo fundamental utilizado para que LBTS satisfaça a condição de linearização.

Algoritmo 7 Operação *rdp* do LBTS (cliente p e servidor s).

{CLIENTE}
procedure $rdp(\bar{t})$
1: $\forall s \in U$, $send(s, \langle \text{RDP}, \bar{t} \rangle)$
2: $\forall x$, $Replies[x][s_1..s_n] \leftarrow \perp$
3: **repeat**
4: **wait until** $receive(s, \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r_s \rangle_{\sigma_s})$
5: $Replies[r_s][s] \leftarrow \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r_s \rangle_{\sigma_s}$
6: **until** $\exists r, \{s : Replies[r][s] \neq \perp\} \in \mathcal{Q}$
7: $\forall s \in U$, $send(s, \langle \text{RDP-COMPLETE}, \bar{t} \rangle)$
8: **if** $\exists t, count\_tuple(t, r, Replies[r]) \geq q$ **then**
9: **return** t
10: **else if** $\exists t, count\_tuple(t, r, Replies[r]) \geq f + 1$ **then**
11: $\forall s \in U$, $send(s, \langle \text{WRITEBACK}, t, r, Replies[r] \rangle)$
12: **wait until** $\exists Q \in \mathcal{Q}, \forall s \in Q$, $receive(s, \langle \text{ACK-WB} \rangle)$
13: **return** t
14: **else**
15: **return** $\perp$
16: **end if**

{SERVIDOR}
upon $receive(p, \langle \text{RDP}, \bar{t} \rangle)$
17: $L_s \leftarrow L_s \cup \{\langle p, \bar{t} \rangle\}$
18: $T_s^{\bar{t}} \leftarrow \{t \in T_s : m(t, \bar{t})\}$
19: $send(p, \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r_s \rangle_{\sigma_s})$
upon $receive(p, \langle \text{RDP-COMPLETE}, \bar{t} \rangle)$
20: $L_s \leftarrow L_s \setminus \{\langle p, \bar{t} \rangle\}$
upon $receive(p, \langle \text{WRITEBACK}, t, r, proof \rangle)$
21: **if** $count\_tuple(t, r, proof) \geq f + 1$ **then**
22: **if** $t \notin R_s$ **then**
23: $T_s \leftarrow T_s \cup \{t\}$
24: **end if**
25: $send(p, \langle \text{ACK-WB} \rangle)$
26: **end if**
upon remoção de t de T_s ou inserção de t em T_s
27: **for all** $\langle p, \bar{t} \rangle \in L_s : m(t, \bar{t})$ **do**
28: $T_s^{\bar{t}} \leftarrow \{t' \in T_s : m(t, \bar{t})\}$
29: $send(p, \langle \text{REP-RDP}, T_s^{\bar{t}}, r_s \rangle_{\sigma_s})$
30: **end for**

Predicado: $count\_tuple(t, r, msgs) \triangleq |\{s \in U : msgs[s] = \langle \text{REP-RDP}, s, T_s^{\bar{t}}, r \rangle_{\sigma_s} \wedge t \in T_s^{\bar{t}}\}|$

Quando a operação $rdp(\bar{t})$ é invocada, o cliente p envia o molde $\bar{t}$ para todos os servidores (linha 1). Quando um servidor s recebe esta mensagem, ele registra p como um *listener* (adicionando-o ao conjunto L_s) e envia uma resposta a requisição contendo todas as tuplas em T_s que combinam com $\bar{t}$ e o número atual de tuplas removidas r_s (linhas 17-19). Enquanto p está registrado como um *listener*, sempre que uma tupla é adicionada ou removida do espaço de tuplas, o conjunto com as tuplas que combinam com o molde passado por p é enviado a ele (linhas 27-30).

O processo cliente p coleta as respostas assinadas dos servidores, colocando-as na matriz esparça *Replies*, até que consiga obter um conjunto de respostas de um quórum de servidores reportando o mesmo número de remoções r (linhas 2-6). Depois disso, uma mensagem RDP-COMPLETE é enviada aos servidores (linha 7).

O resultado da operação depende da linha r da matriz *Replies* já que, ela contém dados provenientes de pelo menos um quórum de servidores que, no momento do envio das respostas, haviam executado um mesmo número r de remoções de tuplas (operações *inp*). Se existe alguma tupla t em $Replies[r]$ que aparece nas respostas de um quórum de servidores, então t é o resultado da operação (linhas 8-9). Isto é possível porque este quórum garante que uma tupla pode ser o resultado de qualquer leitura subseqüente a esta. Caso contrário, se não existe uma tupla retornada por um quórum completo, porém existe uma tupla t retornada por mais de f servidores[9] reportando um mesmo valor de r, então t é **reescrita** nos servidores (linhas 10-12). O propósito da reescrita é garantir que, se t não foi removida até r, então ela poderá ser lida em operações $rdp(\bar{t})$ subseqüentes ($m(t,\bar{t})$) executadas por qualquer cliente, até que t seja removida. Esta fase de reescrita é fundamental para o tratamento de tuplas fantasmas (ver seção 4.5) – que agora podem ter suas escritas completadas – e garantir a propriedade de linearização (ver seção 4.4.5). Isto cobre a limitação de número 2 do BTS.

Quando da recepção de uma mensagem de reescrita $\langle \text{WRITEBACK}, t, r, proof \rangle$, o servidor s verifica se a reescrita é **justificada**, isto é, se *proof* inclui pelo menos $f+1$ mensagens REP-RDP corretamente assinadas de diferentes servidores contendo r e t (linha 21). Uma reescrita não justificada é ignorada pelos servidores corretos. Após esta verificação, se t não está presente em T_s e nem em R_s (não foi removida), então o servidor s insere t em sua cópia local do espaço de tuplas (linhas 22-23). Finalmente, s envia a mensagem ACK-WB ao cliente (linha 25), que espera estas respostas de um quórum de servidores para então retornar t como o resultado da operação (linhas 12-13).

Otimizações na leitura

O protocolo do algoritmo 7 usualmente não requer a fase de reescrita quando em execuções livres de falhas e quando existem muitas tuplas no espaço que combinam com o molde utilizado na leitura. Neste caso, é muito provável que alguma tupla apareça nos conjuntos retornados por todos os servidores em um quórum, o que permite o fim da operação sem a necessidade de reescrita. Esta terminação rápida evita a necessidade da verificação se a reescrita é justificada, algo que implica na verificação das assinaturas no conjunto de mensagens REP-RDP. Assim, as assinaturas nestas mensagens não têm utilidade quando o protocolo termina em um único acesso aos servidores.

Criptografia de chave pública tem sido mostrada como o maior gargalo em sistemas tolerantes a faltas bizantinas práticos [36, 108], especialmente em redes locais e outras redes de alta velocidade [22]. Visando evitar o uso de assinaturas em casos onde elas não seriam necessárias (que esperamos serem bastante freqüentes), a seguinte otimização pode ser utilizada no protocolo: no início da execução da operação *rdp*, o cliente acessa um quórum de servidores pedindo as tuplas que combinam com seu molde, sem requisitar mensagens REP-RDP assinadas ou o uso de padrão *listener*. Se alguma tupla t for retornada por todos os servidores do quórum consultado, então a operação termina e t é o resultado. Se não existe tupla retornada por mais de f servidores, então a operação termina e o resultado é $\perp$. Se alguma tupla foi retornada por pelo menos $f+1$ servidores, o cliente sabe que

[9] Se a tupla é retornada por menos de $f+1$ servidores ela pode não ter sido inserida nos servidores, e pode estar sendo criada por uma coalisão de servidores maliciosos.

existe resposta para sua leitura, porém uma reescrita pode ser necessária, assim o protocolo normal definido pelo algoritmo 7 é usado.

Esta otimização permite leituras rápidas (em 2 passos de comunicação) em invocações sem remoções concorrentes e livres de falha. Como essas leituras não requerem o uso de criptografia de chave pública, elas terão latência extremamente baixa em redes de alta velocidade. Esta otimização é especialmente interessante quando o sistema é instalado em uma rede local, onde experimentos demonstram que a quantidade de escritas concorrentes é negligenciável, mesmo que o sistema esteja sujeito a uma carga elevada de requisições [22].

Uma outra otimização que deve ser usada neste protocolo objetivando diminuir o número de mensagens trocadas entre clientes e servidores é o envio da mensagem RDP-COMPLETE de "carona" (*piggybacked*) com a mensagem WRITEBACK, quando a fase de reescrita for necessária. Esta otimização permite que os servidores processem essas duas ações de uma vez só.

4.4.4 Leitura destrutiva de tuplas - *inp*

Os protocolos anteriores do LBTS foram implementados usando apenas técnicas comuns em protocolos para sistemas de quóruns, como o padrão *listener* [92] e reescritas [87]. O protocolo para a operação *inp*, por outro lado, requer o emprego de abstrações mais fortes. Isto é uma conseqüência direta da semântica do espaço de tuplas que não permite que uma mesma tupla seja removida duas vezes, i.e. cada tupla deve ser o resultado de no máximo uma operação *inp* (propriedade 4 definida na seção 4.2).

A fim de construir um protocolo para *inp* mais eficiente que aquele usado no BTS, que requer um elevado número de passos de comunicação para ser completado, além do uso de criptografia assimétrica, utilizamos uma abordagem diferente da usada naquela construção para implementar esta operação. Ao invés de dividir a operação em duas sub-tarefas para garantir que as remoções ocorram com ordem total em todos os servidores (travar o espaço e depois remover a tupla), como feito no BTS, o protocolo de leitura destrutiva do LBTS utiliza uma versão modificada do protocolo de difusão com ordem total baseado no PAXOS.

A ordenação total via PAXOS funciona da seguinte forma (uma descrição mais completa deste protocolo pode ser encontrada na seção 2.3.2). Quando um cliente quer difundir uma mensagem m com ordem total, ele envia uma mensagem a todos os servidores. Um dos servidores receptores, digamos s, é chamado líder. Quando s recebe m ele dá a essa mensagem um número de seqüência i e envia o par $\langle \mathsf{H}(m), i \rangle$ aos outros servidores. Se um servidor s' recebe $\langle \mathsf{H}(m), i \rangle$ do líder, e ele recebeu m do cliente, e também não aceitou nenhuma outra mensagem com número de seqüência i, s' **aceita** m como a i-ésima mensagem a ser entregue. Quando isso acontece, s' se engaja em dois *rounds* de troca de mensagens com os outros servidores para chegar a um acordo a respeito da associação m ao número de seqüência i. Quando o acordo é alcançado, todo servidor confirma que m será a i-ésima mensagem a ser entregue. Se algum servidor detecta que o líder é falho (ex. porque ele deixou espaços entre os números de seqüência de mensagens), ele pode tentar eleger um novo líder.

O protocolo *inp* é implementado através do uso de uma versão modificada do protocolo descrito acima. A idéa básica é fazer com que o servidor líder no protocolo de ordenação total envie, juntamente com o número de seqüência da mensagem (requisição *inp*), uma tupla candidata a resultado da operação. O principal benefício neste caso é que toda a operação de remoção é realizada em apenas uma execução do PAXOS, tendo portanto a mesma eficiência de uma implementação para esta operação na replicação máquina de estados. No entanto, para que esta estratégia funcione, uma série de modificações no protocolo básico de difusão com ordem total devem ser realizadas:

1. Quando um líder s recebe a requisição $inp(\bar{t})$ (i.e. uma mensagem $\langle \text{INP}, p, \bar{t} \rangle$) enviada por um cliente p, ele envia aos outros servidores não apenas um número de seqüência para a mensagem, mas também uma tupla $t_{\bar{t}} \in T_s$ que combina com o molde $\bar{t}$ passado pelo cliente. Caso não exista uma tupla que combine com o molde, o líder propõem $\perp$;

2. Um servidor s' aceita remover uma tupla $t_{\bar{t}}$ recebida de um líder s na mensagem se *(i.)* as condições usuais referentes à mensagem e ao número de seqüência do algoritmo de difusão com ordem total são satisfeitas; *(ii.)* s' não aceitou a remoção de $t_{\bar{t}}$ anteriormente; *(iii.)* $m(t_{\bar{t}}, \bar{t})$; e *(iv.)* $t_{\bar{t}}$ não é forjada, i.e. $t_{\bar{t}} \in T_{s'}$ ou s' recebe $f+1$ mensagens assinadas por diferentes servidores garantindo que eles têm $t_{\bar{t}}$ em seus espaços de tuplas locais. Esta última condição garante que a tupla t pode ser removida se e somente se ela pode ser lida;

3. Quando um novo líder l' é eleito, cada servidor envia a l' seu estado relativo ao protocolo (como no protocolo original para ordenação total baseado no PAXOS – ver seções 2.3.1 e 2.3.2) e um conjunto assinado com as tuplas em sua cópia do espaço de tuplas local que combinam com $\bar{t}$. Esta informação é usada por l' para construir a justificativa da proposta contendo a tupla t (em caso de t ser informada por mais de $f+1$ servidores). Caso não exista uma tupla retornada por $f+1$ servidores, este conjunto de tuplas serve para justificar a proposta de $\perp$ como o resultado da operação.

O protocolo para a operação *inp*, usando a versão modificada do protocolo de ordem total é apresentada no algoritmo 8.

Para um cliente p, o algoritmo para invocação de $inp(\bar{t})$ funciona exatamente como se estivesse acessando um espaço de tuplas construído usando replicação Máquina de Estados (ver seção 3.5.2): p difunde uma requisição para todos os servidores usando uma primitiva de difusão com ordem total e espera por $f+1$ respostas contendo resultados idênticos advindas de diferentes servidores, para então retorná-lo como o resultado da operação (linhas 1-3).

Do lado servidor, as requisições para execução de *inp* recebidas são inseridas em um conjunto de mensagens pendentes P_s. Quando este conjunto é não vazio, o código nas linhas 4-13 é executado pelo líder (o predicado $paxos\_leader(s)$ é *true* se e somente se s é o líder atual na replicação). Cada uma destas requisições pendentes é proposta em uma instância do PAXOS cujo identificador corresponde ao número de seqüência proposto para a mensagem. A seguir o líder escolhe uma tupla de seu espaço que combina com $\bar{t}$ e não esteja marcada para remoção (linhas 6-7), e a marca para remoção (linha 8). O procedimento $mark(i,t)$ marca a tupla, enquanto o predicado $marked(t)$ verifica se uma tupla

Algoritmo 8 Operação *inp* do LBTS (cliente p e servidor s).

{CLIENTE}
procedure $inp(\bar{t})$
1: $TO\text{-}multicast(U, \langle \text{INP}, p, \bar{t} \rangle)$
2: **wait until** receba a mensagem $\langle \text{REP-INP}, t_{\bar{t}} \rangle$ de $f+1$ servidores de U
3: **return** $t_{\bar{t}}$

{SERVIDOR}
upon $paxos\_leader(s) \wedge P_s \neq \varnothing$
4: **for all** $\langle \text{INP}, p, \bar{t} \rangle \in P_s$ **do**
5: $i \leftarrow i+1$
6: **if** $\exists t \in T_s : m(t, \bar{t}) \wedge \neg marked(t)$ **then**
7: $t_{\bar{t}} \leftarrow t$
8: $mark(i, t)$
9: **else**
10: $t_{\bar{t}} \leftarrow \perp$
11: **end if**
12: $paxos\_propose(i, \langle t_{\bar{t}}, \mathsf{H}(\langle \text{INP}, p, \bar{t} \rangle) \rangle)$
13: **end for**

upon $paxos\_deliver(\langle t_{\bar{t}}, \langle \text{INP}, p, \bar{t} \rangle \rangle)$
14: $unmark(i)$
15: $P_s \leftarrow P_s \setminus \{\langle \text{INP}, p, \bar{t} \rangle\}$
16: **if** $t_{\bar{t}} \neq \perp$ **then**
17: **if** $t_{\bar{t}} \in T_s$ **then**
18: $T_s \leftarrow T_s \setminus \{t_{\bar{t}}\}$
19: **end if**
20: $R_s \leftarrow R_s \cup \{t_{\bar{t}}\}$
21: $r_s \leftarrow r_s + 1$
22: **end if**
23: $send(p, \langle \text{REP-INP}, t_{\bar{t}} \rangle)$

está marcada. Caso não exista nenhuma tupla não marcada que combine com $\bar{t}$ na cópia local do espaço do servidor líder, o valor $\perp$ é proposto como resultado da operação, sendo enviado aos outros servidores (linhas 10 e 12). O código nas linhas 4-13 corresponde à primeira modificação descrita para o algoritmo PAXOS. As modificações 2 e 3 não aparecem explicitamente no código já que são razoavelmente simples, e não alteram severamente o algoritmo original.

Quando a requisição de INP é ordenada e os servidores estabelecem acordo sobre o valor de resposta para a operação, a requisição e sua resposta são entregues a camada de aplicação através de uma chamada ao procedimento *paxos_deliver*, quando o código do lado direito do algoritmo é executado[10] (linhas 14-23). Neste código, cada servidor s desmarca qualquer tupla marcada para remoção com este número de seqüência e remove a requisição de P_s (linhas 14-15). Depois disso, se o resultado da operação é uma tupla valida $t_{\bar{t}}$, o servidor verifica se ela existe em sua cópia local do espaço de tuplas T_s (linha 17), removendo-a de T_s em caso afirmativo (linha 18). Finalmente, $t_{\bar{t}}$ é incluída no conjunto das tuplas removidas R_s, o contador de remoções r_s é incrementado e o resultado da operação é enviado ao processo cliente (linhas 20-23).

É interessante notar que o PAXOS BIZANTINO não usa criptografia de chave pública já que todas as assinaturas requeridas no protocolos são feitas através de autenticadores, que nada mais são que vetores de códigos de autenticação de mensagens [36]. No entanto, as modificações 2 e 3 requeridas na versão modificada do algoritmo requerem uso de assinaturas baseadas em criptografia de chave pública no conjunto de tuplas enviado pelos servidores para um novo líder quando de sua eleição. Assim, o protocolo *inp* do LBTS está sujeito à latência adicional decorrente deste tipo de criptografia, porém apenas quando uma operação não puder ser resolvida no primeiro *round* de execução do algoritmo.

[10] Não usamos a primitiva *TO-deliver* devido ao fato do algoritmo em questão fazer mais do que ordenar mensagens.

Problemas e Soluções do PAXOS BIZANTINO Modificado

O algoritmo de difusão com ordem total baseado no PAXOS garante que todas as requisições serão executadas na mesma ordem em todos os servidores do sistema, mesmo com falhas em um número ilimitado de clientes e em até f servidores. No entanto, as modificações introduzidas dão margem para alguns ataques e situações indesejadas que não eram consideradas no algoritmo de ordenação total original. Mais especificamente, três novos problemas relacionados com a tupla proposta pelo líder podem aparecer:

1. **Um líder falho envia diferentes tuplas para diferentes servidores**. Este problema é resolvido pelo próprio algoritmo PAXOS, já que ele garante acordo entre os servidores no que diz respeito ao valor proposto (número de seqüência, requisição enviada pelo cliente e tupla a ser removida);

2. **Um líder falho envia uma tupla $t_{\bar{t}}$ não presente no espaço de outros servidores**. Uma tupla só pode ser removida se pelo menos $\lceil \frac{n+f}{2} \rceil$ servidores aceitarem a proposta do líder. Isto só acontece se *(i.)* estes servidores têm essa tupla em suas cópias locais do espaço, ou *(ii.)* o líder apresentou uma justificativa contendo mensagens assinadas de pelo menos $f+1$ servidores reportando ter $t_{\bar{t}}$. Em ambos os casos, algum processo correto precisa reportar ter a tupla em seu espaço local, o que é claramente impossível para tuplas não inseridas ou já removidas;

3. **Um líder falho propõem uma tupla fantasma** $t_{\bar{t}}$. O protocolo apresentado permite que uma tupla presente em apenas $f+1$ servidores seja removida, já que uma tupla proposta e corretamente justificada pelo líder é aceita por um servidor correto mesmo que ele não tenha esta tupla em sua cópia do espaço local (veja modificação 2). Um sub-problema específico é que cada servidor pode ter diferentes tuplas que combinam com o molde passado em seus espaços de tuplas locais, e todas essas tuplas podem estar presentes em f ou menos servidores. Conseqüentemente, nenhum líder correto conseguirá propor uma tupla que será aceita para remoção. Quando isto acontece (o que é esperado ser raro), o líder atual será suspeito e um novo líder será eleito para para um novo *round* do algoritmo. O novo líder, se correto, vai propor $\perp$, já que ele não encontrará nenhuma tupla nas mensagens enviadas pelos servidores para justificar outro tipo de proposta, i.e. não existe nenhuma tupla presente nos espaços locais de $f+1$ servidores.

4.4.5 Prova de Correção do LBTS

Nesta seção é apresentada a prova formal de que o LBTS implementa um espaço de tuplas que satisfaz as propriedades de linearização e liberdade de espera, juntamente com as condições de correção estipuladas na seção 4.2. Vale lembrar que o LBTS assume $|U| = n \geq 4f+1$ servidores e que o tamanho de um quórum é $q = \lceil \frac{n+2f+1}{2} \rceil$.

Antes, porém, algumas definições devem ser apresentadas. Dizemos que uma tupla t **existe** em um servidor s, ou que um servidor s **tem** t, quando $t \in T_s$. Outra definição importante é a noção de tupla legível.

Definição 1 *Uma tupla t é **legível** se ela será o resultado de uma operação $rdp(\bar{t})$, com $m(t,\bar{t})$, executada isoladamente (sem concorrência) em um espaço de tuplas contendo apenas a tupla t.*

A definição diz que uma tupla é legível se ela será o resultado de alguma operação *rdp* independentemente do quórum acessado e do número de faltas no sistema (desde que o limite máximo de t servidores falhos seja respeitado). A partir do algoritmo que implementa *rdp* no LBTS, podemos inferir que uma tupla será legível se ela aparecer em $f+1$ servidores corretos de qualquer quórum do sistema. Isto implica que a tupla precisa existir em $n-q+f+1$ servidores corretos[11] ($f+1$ servidores do quórum mais os $n-q$ que não pertencem ao quórum).

Dada esta definição, o primeiro lema mostra que se uma tupla é legível, então ela pode ser removida do espaço de tuplas.

Lema 1 *Se após uma operação op uma tupla t é legível, então t será o resultado de uma operação $inp(\bar{t})$, $m(t,\bar{t})$, executada logo após op.*

Prova: Assuma que após *op* o número de tuplas removidas no sistema é r. Assuma também que um cliente correto p invoca $inp(\bar{t})$. Se t é legível após r remoções, então existe um conjunto de pelo menos $n-q+f+1$ servidores corretos que têm t. Nesta condição, temos de provar que uma execução de $inp(\bar{t})$ retorna t. Considerando que o líder atual na execução da ordenação total com o PAXOS BIZANTINO é o servidor l, temos de verificar dois casos:

1. l **é correto** – novamente, dois casos devem ser considerados:

   (a) $t \in T_l$: neste caso l irá propor t como o resultado da operação. Como existem pelo menos $n-q+f+1$ servidores corretos que têm t, eles aceitam esta proposta e a tupla será o resultado da operação.

   (b) $t \notin T_l$: neste caso l vai propor $\perp$. Este valor não será aceito já que t existe em $n-q+f+1$ servidores corretos e portanto no máximo $q-f-1$ aceitarão $\perp$. Como $q-f-1 < \lceil \frac{n+f}{2} \rceil$ o valor não será decidido neste *round* e um novo líder será eleito[12]. Como nenhuma tupla foi aceita para remoção por $n-q+f+1$ servidores, o novo líder poderá escolher t como o resultado da operação e este valor será decidido ($n-q+f+1 > \lceil \frac{n+f}{2} \rceil$). Qualquer outra proposta não será aceita pelos servidores que têm t, o que levará o sistema a escolher um novo líder l'. Se l' for correto, então ele receberá os estados dos servidores e perceberá que a única tupla que pode ser proposta é t. Assim, este valor será proposto juntamente com sua justificativa da proposta, o que levará todos os servidores corretos do sistema a aceitá-la. Se l' for falho, aplica-se o caso 2.

2. l **é falho** – neste caso, l pode propor $\perp$ ou algum $t' \neq t$. Se $\perp$ for proposto, ele não será aceito por nenhum dos $n-q+f+1$ servidores corretos que têm t (já que $m(t,\bar{t})$). Se t' for proposto,

[11] Para melhor compreensão das provas apresentadas nesta seção, é sugerido que se tome o valor limite de $n = 4f+1$ e $q = 3f+1$.

[12] Lembrar que, no PAXOS BIZANTINO, um valor só é decidido se $\lceil \frac{n+f}{2} \rceil$ processos o aceitam [36, 130].

ele não será aceito por mais de f servidores, e portanto não será decidido como o resultado da operação. A razão para isto é que, pela definição de legível, não existe outra tupla no espaço que combina com $\bar{t}$, e portanto nenhum servidor correto terá t' em seu espaço local (ou t' e $\bar{t}$ não combinam). Em ambos os casos, no máximo $n-q+f$ servidores aceitarão o resultado proposto pelo líder e a decisão não será alcançada neste *round*, o que causará a eleição de um novo líder. Dependendo se este novo líder será correto ou não, os casos 1 ou 2 podem ser aplicados novamente.

Em ambos os casos, o resultado de $inp(\bar{t})$ será t. ■

O próximo lema prova que uma tupla não pode ser lida antes de ser inserida no espaço de tuplas. Este lema é importante porque demonstra que os servidores falhos não podem "inventar" tuplas.

Lema 2 *Antes de $\langle out(t)/ack\rangle$, t não é legível.*

Prova: Uma tupla será considerada para leitura se pelo menos $f+1$ servidores a enviarem em resposta a requisição de leitura. Isto significa que pelo menos um servidor correto precisa retornar t. Como nenhum servidor correto fará isso antes de ter t (o que só ocorre após o início de $out(t)$), t não pode ser legível. ■

O próximo lema prova que uma tupla inserida no espaço permance legível enquanto sua remoção não acontece.

Lema 3 *Após a operação $\langle out(t)/ack\rangle$ e antes da operação $\langle inp(\bar{t})/t\rangle$, t é legível.*

Prova: A definição de legível considera que t é a única tupla no espaço que combina com $\bar{t}$. Se t foi inserida por uma operação $out(t)$ então existe um quórum de servidores Q_1 que têm t. Considere uma leitura que ocorre após essa inserção e antes da remoção de t. Temos de provar que t sempre será o resultado da leitura executada através da invocação de $rdp(\bar{t})$, supondo $m(t,\bar{t})$. Após a linha 6 do protocolo $rdp(\bar{t})$, sabemos que um quórum de servidores Q_2 retornaram suas tuplas que combinam com $\bar{t}$ após r remoções de tuplas. Cada servidor correto de Q_2 que é membro de Q_1 irá responder t. Como a intersecção de cada dois quóruns em um sistema de quórum de disseminação tem pelo menos $2f+1$ servidores, dos quais no máximo f são falhos, t estará presente na resposta de pelo menos $f+1$ servidores consultados para leitura, e portanto será considerada como resultado para a leitura (já que não existem outras tuplas que combinam com $\bar{t}$). Assim, t é legível. ■

O lema a seguir prova que uma tupla que é resultado de alguma operação de leitura permanece legível até sua remoção.

Lema 4 *Após a operação $\langle rdp(\bar{t})/t\rangle$ e antes da operação $\langle inp(\bar{t})/t\rangle$, t é legível.*

Prova: Sabemos que um quórum de servidores têm t após o término de uma operação $rdp(\bar{t})$ que tem como resultado esta tupla (isso graças a fase de reescrita). Conseqüentemente, seguindo o mesmo raciocínio usado na prova do lema 3, podemos ver que t é legível. ■

Os próximos dois lemas mostram que não é possível remover a mesma tupla duas vezes.

Lema 5 *Após a operação $\langle inp(\bar{t})/t\rangle$, t não é legível.*

Prova: Assuma que t foi a r-ésima tupla removida do espaço de tuplas. Assuma também, por motivos de simplificação da prova, que não existem outras remoções no sistema. Este lema será provado por contradição. Suponha que uma operação $rdp(\bar{t})$ executada após r remoções retorna t. Para que isto aconteça, t foi reportado por pelo menos $f+1$ servidores que removeram r tuplas. Isto é claramente impossível já que nenhum servidor correto retornará t após sua remoção. Conseqüentemente, o lema esta provado. ■

Lema 6 *Uma tupla t não pode ser removida mais de uma vez.*

Prova: Devido as propriedades de segurança do algoritmo de ordem total, todas as remoções são executadas na mesma ordem em todos os servidores. Pelo lema 5, uma tupla removida não é mais legível. Além disso, pelo lema 1, uma tupla não legível não pode ser removida. Conseqüentemente, se t foi removida, ela não poderá ser o resultado de outra operação *inp*. ■

Os próximos lemas demonstram que as três operações providas pelo LBTS satisfazem a propriedade de terminação livre de espera [67], ou seja, elas sempre terminam em nosso modelo de sistema.

Lema 7 *A operação out é livre de espera.*

Prova: O único ponto em que um cliente executando o algoritmo 6 (*out*) pode ficar bloqueado é na espera de respostas de um quórum de servidores (q servidores do sistema). Como $q \leq n-f$ (propriedade de disponibilidade, enunciada na seção 2.4.2), sempre existirá um quórum capaz de responder a requisição de inserção, e portanto a operação termina. ■

Lema 8 *A operação rdp é livre de espera.*

Prova: A primeira fase do protocolo de leitura consiste no cliente p esperar respostas de um quórum de servidores que removeram r tuplas. Devido as propriedades de acordo do PAXOS BIZANTINO, se um servidor correto remover r tuplas, então todos os servidores corretos terminarão por remover r tuplas. A utilização do padrão *listener* para fazer com que cada servidor notifique os clientes leitores (incluindo p) sobre as tuplas que combinam com seus moldes de interesse após cada nova remoção, faz com que termine por existir algum r tal que todos os servidores corretos responderam a uma leitura após r remoções. E portanto, a primeira fase termina.

A fase de reescrita do protocolo, quando necessária, também satisfaz a terminação livre de espera já que o requisito para desbloqueio do cliente é o recebimento de confirmações de um quórum de servidores do sistema. Como sempre existe um quórum de servidores corretos (já que $q \leq n-f$) e estes sempre confirma uma reescrita corretamente justificada, não é possível que o cliente fique bloqueado. Isto caracteriza a operação como livre de espera. ■

Lema 9 *A operação inp é livre de espera.*

Prova: A vivacidade desta operação é diretamente dependente da vivacidade do algoritmo PAXOS BIZANTINO modificado. Este algoritmo é construído de tal forma que um novo líder seja eleito até que algum líder correto consiga propor um valor e este seja decidido em um *round* "síncrono" (onde todas as mensagens são trocadas em tempo e nenhum processo é suspeito) [36]. Neste *round*, o líder propõem um numero de seqüência e um resultado t para uma operação $inp(\bar{t})$ sendo invocada. As propriedades do algoritmo de ordenação total garantem que o número de seqüência será válido [36]. Já o resultado t, será aceito pelos outros servidores se ele é aceito por $\lceil \frac{n+f}{2} \rceil$ servidores. Isto sempre ocorre se o protocolo para $out(t)$ foi corretamente executado pelo cliente ou se a proposta de t está acompanhada por uma justificativa válida. Tendo em vista isso, temos de considerar três casos:

1. t **foi completamente inserida.** Neste caso, $q = \lceil \frac{n+2f+1}{2} \rceil$ servidores têm t e pelo menos $q-f$ (os corretos) vão aceitar a proposta. Como $q - f \geq \lceil \frac{n+f}{2} \rceil$ é sempre verdadeiro quando $n \geq 4f+1$, esta proposta terminará por ser decidida.

2. t **foi parcialmente inserida.** Se t existir em $\lceil \frac{n+f}{2} \rceil$ servidores, ela será aceita e decidida como resultado da operação. Se t for proposta acompanhada de uma justificativa mostrando que ela existe em pelo menos $f+1$ servidores, esta será aceita e t terminará por ser decidida. Se nem t não existe em $\lceil \frac{n+f}{2} \rceil$ servidores e nem tem sua proposta justificada, este valor não será aceito por um número de servidores suficientes e um novo líder terminará por ser eleito. Este líder receberá as tuplas que combinam com o molde passado na operação sendo executada e, supondo que ele seja correto, escolherá um resultado que possa ser justificado.

3. $t = \perp$**.** Este valor será aceito pelos servidores se não existirem tuplas que combinam com molde passado como parâmetro na operação ou o valor é justificado por um conjunto de mensagens mostrando que não existe nenhuma tupla presente em pelo menos $f+1$ servidores. Se este valor é aceito por $\lceil \frac{n+f}{2} \rceil$ servidores, ele terminará por ser decidido.

Em todos estes casos, t será aceito, e a operação termina. ■

Usando todos estes lemas podemos provar que o LBTS implementa um espaço de tuplas que satisfaz linearização e liberdade de espera.

Teorema 1 *LBTS é um espaço de tuplas linearizável cujas operações são livres de espera.*

Prova: A terminação livre de espera das operações providas pelo espaço é provada nos lemas 7, 8 e 9. Assim, resta-nos provar a linearização.

Considere qualquer história H de um sistema onde os processos interagem unicamente através do LBTS. Considere que H é completa, i.e. todas as invocações em H têm uma resposta associada[13]. Nessa história, um conjunto de tuplas T_H são manipuladas (inseridas, lidas e removidas). Como no ponto de vista de uma história completa, não existe interferência entre operações que manipulam diferentes tuplas, podemos considerar que cada tupla t é um objeto diferente. Assim, para toda tupla $t \in T_H$, denotamos por $H|t$ a sub-história de H que contém apenas operações que manipulam t.

[13] Se ela não for completa, podemos estendê-la com as respostas faltando.

Para cada $t \in T_H$, a premissa de que as tuplas são únicas garante que vai existir apenas uma operação $\langle out(t)/ack \rangle \in H|t$ e o lema 6 garante que não existirá mais de uma remoção para t nesta sub-história. Desta forma, $H|t$ vai conter uma operação *out*, zero ou mais leituras de t e no máximo uma operação *inp* cujo resultado é t.

Para provar que o LBTS é um espaço de tuplas linearizável devemos construir a prova em três passos. Primeiramente devemos construir uma história seqüencial $H'|t$ para cada tupla $t \in T_H$ com todas as operações seqüenciais de $H|t$ preservando sua ordem original. O segundo passo consiste em ordenar as operações concorrentes de acordo com as propriedades do LBTS (estipuladas pelos lemas 1-6). A seguir, mostraremos que $H'|t$ está de acordo com a especificação seqüencial de um espaço de tuplas em que apenas um tupla é manipulada. Finalmente, usaremos a propriedade de localidade da linearização (Teorema 1 em [69]): se para todo $t \in T_H$, $H|t$ é linearizável, então H é linearizável.

1. Para operações seqüenciais, para toda tupla $t \in T_H$, os lemas 2 e 3 mostram que $\langle rdp(\bar{t})/t \rangle$ só pode ocorrer em $H|t$ após sua inserção. Já os lemas 3, 4 e 5 mostram que todas as operações $\langle rdp(\bar{t})/t \rangle$ acontecem antes da remoção de t;

2. Para cada tupla $t \in T_H$, vamos ordenar as operações concorrentes em $H|t$ obtendo $H'|t$ contendo todas as operações de $H|t$. Esta ordenação deve ser feita da seguinte forma: todas as operações $\langle rdp(\bar{t})/t \rangle$ são colocadas após $\langle out(t)/ack \rangle$ (o lema 2 prova que uma tupla t não pode ser lida antes de sua inserção, e o lema 3 mostra que t pode ser lida após sua inserção) e antes de $\langle inp(\bar{t})/t \rangle$ (o lema 3 prova que t pode ser lida antes de sua remoção e o lema 5 prova que t não pode ser lida após sua remoção). A ordem entre diferentes operações de leitura que retornam t pode ser escolhida arbitrariamente já que, pelo lema 4, se t é lida uma vez, ela sempre será lida até sua remoção;

3. Após ordenar as operações da forma descrita, toda história $H'|t$, para toda tupla $t \in T_H$, começa com uma operação $\langle out(t)/ack \rangle$, seguida de zero ou mais operações de leitura $\langle rdp(\bar{t})/t \rangle$, e pode terminar com uma operação $\langle inp(\bar{t})/t \rangle$. Claramente, estas histórias satisfazem as quatro condições de correção apresentadas na seção 4.2, mesmo quando operações são executadas concorrentemente no sistema. Portanto, as histórias $H|t$ são linearizáveis.

4. Usando a propriedade de localidade da linearização citada anteriormente, temos que, para todo $t \in T_H$, $H|t$ é linearizável, o que implica na linearização de H.

Isto significa que todas as histórias em que um conjunto de processos se comunicam usando unicamente o LBTS são linearizáveis. Conseqüentemente, o LBTS é linearizável. ■

4.4.6 Avaliação

Esta seção apresenta uma avaliação comparando o LBTS com uma implementação do espaço de tuplas baseada em replicação máquina de estados provendo operações com semântica equivalente.

Esta avaliação é semelhante a apresentada na seção 4.3.6, e também considera as métricas de complexidade de mensagens e número de passos de comunicação.

O LBTS provê um espaço de tuplas linearizável e com semântica ordenada para a operação *out*. Estas características são exatamente as mesmas providas pelo espaço de tuplas com segurança de funcionamento baseado em replicação Máquina de Estados construído no capítulo 3, e portanto tal construção será usada aqui para fins de comparação[14]. Nesta seção chamaremos esta implementação de SMR-TS (*State Machine Replication Tuple Space*). Vale lembrar que ambas as implementações (LBTS e SMR-TS) utilizam o PAXOS BIZANTINO como algoritmo de difusão com ordem total.

Operação	LBTS		SMR-TS	
	Complex. de Mens.	Passos de Com.	Complex. de Mens.	Passos de Com.
out	$\mathbf{O(n)}$	**2**	$\mathbf{O(n^2)}$	**4**
rdp	$\mathbf{O(n)}$	$\mathbf{2}/6$	$\mathbf{O(n)}/O(n^2)$	$\mathbf{2}/6$
inp	$\mathbf{O(n^2)}$	$\mathbf{4}/7$	$\mathbf{O(n^2)}$	**4**

Tabela 4.2: Custo dos protocolos para espaço de tuplas: LBTS e SMR-TS.

A tabela 4.2 apresenta uma avaliação dos protocolos para as operações não bloqueantes dos espaços de tuplas considerando execuções favoráveis (ver seção 2.1.3). Os custos dos protocolos do LBTS são apresentados na segunda e terceira colunas da tabela. As duas últimas colunas da tabela apresentam os custos correspondentes ao SMR-TS. O protocolo do LBTS para a operação *out* é mais eficiente que o do SMR-TS em ambas as métricas já que sua implementação consiste apenas em um acesso aos servidores. O protocolo de leitura, por outro lado, têm os mesmos custos em execuções favoráveis em que não ocorrem inserções ou remoções de tuplas que combinam com o molde sendo lido concorrentemente à operação *rdp*. Isto ocorre devido a otimização na leitura, que só executa a difusão com ordem total no caso da leitura não terminar com uma consulta simples ao sistema (ver seção 3.5.2). Os primeiros valores na linha referente a operação *rdp* apresentam as métricas neste caso (complexidade de mensagens $O(n)$ e 2 passos de comunicação requeridos). Quando a leitura não puder ser resolvida nesta fase otimista, ambos os protocolos requerem 6 passos de comunicação, porém o LBTS tem complexidade de mensagens linear, enquanto o SMR-TS requer troca de mensagens entre os servidores. O protocolo para a operação *inp* requer uma única execução do PAXOS em ambas as implementações. Entretanto, se houverem muitas tuplas não completamente escritas (o que na replicação Máquina de Estados é impossível já que temos difusão com ordem total na escrita), o LBTS pode não decidir no primeiro *round* do algoritmo mesmo tendo um líder correto (veja o item 3 da seção 4.4.4). Esperamos que estes casos com muitas tuplas fantasmas sejam muito raros em execuções reais do sistema.

A tabela nos permite concluir que a maior vantagem do LBTS em relação ao SMR-TS reside no fato de que, neste último, todas as operações requerem protocolos com complexidade de mensagens $O(n^2)$, tornando operações simples como *rdp* e *out* tão complexas como *inp*. Outra vantagem do LBTS é o fato de que operações simples como *out* e *rdp* sempre terminarem em poucos passos de comunicação enquanto que no SMR-TS podemos ter certeza que as operações terminam em 4 passos apenas em execuções favoráveis [91, 130].

[14]Considerando somente a camada de replicação (seção 3.5.2).

Assim como na avaliação do BTS, a tabela comparativa não apresenta métricas para execuções não favoráveis. Neste caso, todas as operações baseadas no PAXOS BIZANTINO são atrasadas até que exista sincronismo suficiente no sistema para que o protocolo termine (instante $u > \text{GST}$). Este problema é especialmente relevante em sistemas operando em redes de larga escala e ambientes sujeitos a faltas bizantinas, onde um atacante pode atrasar as comunicações (ex. através de um ataque de negação de serviço) em momentos específicos da execução do protocolo com o propósito de atrasar sua terminação.

Finalmente, existe uma outra vantagem do LBTS sobre o SMR-TS (e do BTS sobre o WSMR-TS) que as métricas apresentadas na tabela não revelam: o LBTS executa as operações concorrentes em todos os servidores enquanto o SMR-TS ordena todas as operações concorrentes antes de executá-las. Esta diferença pode causar um impacto no *throughput* do sistema em situações de alta carga.

4.5 O Problema das Tuplas Fantasmas

Os protocolos para as operação *out* do BTS e do LBTS consistem basicamente na difusão da tupla a ser inserida em um conjunto de servidores. Tendo em vista que nosso modelo de sistema prevê falhas nos clientes, é possível que um cliente executando *out* não envie a tupla sendo inserida ao número requerido de servidores devido a uma falha de parada ou mesmo ao seu comportamento malicioso. Quando isto acontece temos a inserção incompleta de uma tupla no espaço, e tal tupla é chamada de **tupla fantasma**.

Formalmente, um tupla fantasma é uma tupla que está presente em pelo menos um servidor correto mas não em um número suficiente que a torne legível. Se ela existe em pelo menos um servidor correto, ela pode ser lida eventualmente, já que, segundo os algoritmos de leitura do BTS e do LBTS, uma tupla é válida para leitura se ela existe em pelo menos $f + 1$ servidores (linha 7 do algoritmo 4 e linha 10 do algoritmo 7). O problema com este tipo de tupla é que ela pode ser lida em algumas execuções da operação *rdp* (por exemplo, se ela existe em alguns servidores corretos, os f maliciosos podem reportar a posse da tupla, o que permitira sua leitura) mas não em todas (por exemplo, se o quórum acessado for composto apenas por servidores corretos, e a tupla existir em menos de $f + 1$ destes servidores).

Do ponto de vista da formalização baseada em histórias introduzida na seção 4.2, podemos explicar a existência deste tipo de tuplas através da ocorrência uma operação *out* que só termina quando a tupla é removida do espaço. Isto significa que todas as operações de leitura e remoção de uma tupla fantasma são vistas como operações concorrentes a sua inserção.

Existem três abordagens para lidar com este problema. A primeira é usar algum tipo de **algoritmo de coleta de lixo distribuído** para remover das réplicas este tipo de tupla. Infelizmente, não se pode utilizar algoritmos convencionais de coleta de lixo distribuída, como os apresentados em [2], uma vez que a noção de dado inacessível não se aplica diretamente a espaços de tuplas (uma tupla não tem "referência"). Até onde sabemos, não existem algoritmos distribuídos que possam ser aplicados diretamente na resolução deste problema.

Recentemente, alguns trabalhos têm sido realizados de tal forma a remover tuplas indesejadas de um espaço de tuplas [29, 94], entretanto estes trabalhos se concentram na remoção de tuplas expiradas (considerando que cada tupla tem um prazo de validade), o que não é o caso no problema em questão. Dada a inexistência de uma abordagem satisfatória para este problema, a remoção de tuplas fantasmas em espaço de tuplas replicado deverá ser alvo de futuras pesquisas, talvez aprimorando o conceito de **apagamento** [94].

A segunda abordagem consiste em fazer com que os servidores que recebem uma tupla a repassem aos outros servidores, desta forma garantindo que uma tupla recebida por um servidor correto acabe por ser recebida por todos os outros. O problema com esta abordagem reside no fato dela incorrer em um grande número de mensagens trocadas, acarretando uma complexidade de mensagens quadrática também na operação de inserção. Esta abordagem é equivalente ao uso de uma primitiva de **difusão confiável** [25, 66] para a inserção de tuplas.

A terceira abordagem para lidar com este problema é **não fazer absolutamente nada**! Como estamos lidando com clientes falhos, qualquer comportamento relacionado às operações realizadas por processos falhos é admissível, afinal, o que existe é uma tupla cuja inserção demora um tempo muito grande para ser completada. Desta forma, no caso do BTS, ela pode ser lida intermitentemente (não garante linearização), enquanto para o LBTS, no momento da primeira leitura ela será reescrita e deixará de ser uma tupla fantasma. Esta abordagem, apesar de não resolver realmente o problema, é a adotada para as construções apresentadas neste capítulo. Na seção 4.6.6 apresentamos uma técnica para limitar a quantidade de tuplas fantasmas que um cliente malicioso pode inserir no espaço de tuplas.

A operação *inp* do BTS também pode deixar uma tupla visível por um tempo, entretanto, a propriedade de terminação do PAXOS no modelo de sistema adotado garante que ela termina por desaparecer dos espaços locais dos processos corretos. Assim, o problema neste caso não é tão grave. Esta capacidade de tuplas removidas permanecerem no espaço já foi explorada em [110], onde o objetivo é melhorar a implementação de estruturas de dados compostas por várias tuplas em um espaço. O conceito de tuplas fantasma apresentado nesta seção é um efeito colateral da estratégia de replicação utilizada e tem pouca relação com o apresentado em [110], entretanto, a semântica de consistência assumida nos dois espaços é equivalente.

4.6 Incrementando o BTS e o LBTS

Existem várias limitações do BTS e do LBTS que podem ser eliminadas através da introdução de pequenas modificações nos algoritmos. Esta seção apresenta estas modificações e discute quais as implicações de seu uso no BTS e no LBTS.

4.6.1 Uso do *inp* do LBTS no BTS

O protocolo para a operação *inp* do LBTS é muito mais eficiente que seu equivalente no BTS. Desta forma, a pergunta que surge é se este protocolo mais eficiente poderia ser utilizado no BTS. A resposta é afirmativa. O protocolo pode ser empregado diretamente sem nenhuma alteração em nenhum dos outros protocolos do BTS.

4.6.2 Múltiplos Espaços de Tuplas

Os protocolos apresentados para o BTS e o LBTS consideram que cada servidor $s \in U$ tem uma cópia T_s do espaço de tuplas T. Dado que a utilização de várias instâncias de espaços de tuplas não interfere na semântica de cada um dos espaços individualmente, a modificação das construções para suportar vários espaços de tuplas é trivial. A solução é colocar uma cópia de cada espaço utilizado em cada servidor e executar os protocolos sempre no escopo de um dos espaços, colocando o identificador do espaço em cada uma das requisições enviadas aos servidores. Note que desta forma as operações *inp* executadas em diferentes espaços podem ser executadas em paralelo (já que tratam de conjuntos de tuplas diferentes).

4.6.3 Remoções em Paralelo no Espaço de Tuplas

Conforme descrito na seção 3.3, o tipo de uma tupla é definido como a seqüência dos tipos dos campos da tupla. De acordo com a regra de combinação definida nesta mesma seção, se assumirmos que um campo com valor '$*$' em um molde pode ser considerando de qualquer tipo para fins de combinação, uma tupla e um molde só combinam se eles têm o mesmo tipo. Usando este artifício, é possível particionar um espaço de tuplas de acordo com os tipos das tuplas presentes neste, e executar operações *inp* que tentam remover tuplas de tipos diferentes em paralelo, como se estas "partições" fossem espaços diferentes. Esta otimização é aplicável tanto no BTS quanto no LBTS, e pode melhor o *throughput* do sistema em cenários de elevada carga.

4.6.4 Suporte a Notificação e Operações Bloqueantes

Até o momento, foram apresentados protocolos apenas para as operações não bloqueantes do BTS e do LBTS (*out*, *rdp* e *inp*). Nesta seção apresentamos protocolos para implementação das operações bloqueantes (*in* e *rd*). Para implementar estas operações eficientemente usamos um mecanismo de **notificação** como o usualmente provido pela maioria das implementações comerciais de espaços de tuplas [120, 121]. Neste mecanismo, processos clientes podem registrar seu interesse em um tipo particular de tupla, representado pelo molde $\bar{t}$, executando uma operação *register*($\bar{t}$). O protocolo para esta operação é não confirmável [93] e consiste no envio da mensagem $\langle \text{REGISTER}, \bar{t} \rangle$ para todos os servidores de U pelo processo interessado p. Quando um servidor recebe esta mensagem, ele armazena o par $\langle p, \bar{t} \rangle$ em seu conjunto *ToNotify*.

Para cada tupla $t \in T_s$ que combina com algum $\bar{t}$ de $\langle p,\bar{t}\rangle \in ToNotify_s$, o servidor s envia a mensagem[15] $\langle \text{NOTIFY},t\rangle$ para p. Esta verificação deve ser feita após cada inserção de tupla em T_s. O cliente p armazena mensagens deste tipo recebidas de servidores e **consolida** uma notificação para uma tupla t se e somente se ele recebe mensagens $\langle \text{NOTIFY},t\rangle$ de $n-q+f+1$ servidores diferentes.

Este quórum de servidores de $n-q+f+1$ é necessário para garantir que a tupla cuja notificação foi consolidada é legível (de acordo com a definição 1) no LBTS. No caso do BTS, onde não precisamos satisfazer linearização, a notificação é consolidada desde que $f+1$ mensagens deste tipo provenientes de diferentes servidores sejam recebidas. Note que, no entanto, isto não garante que uma tupla t cuja notificação foi consolidada por p possa ser lida por ele. Isto ocorre, por exemplo, porque na leitura, p pode acessar outros servidores do sistema, que não os que lhe enviaram as mensagens de notificação, não obtendo portanto t de $f+1$ servidores. Mesmo que o quórum requerido para consolidação no BTS fosse de $2f+1$ servidores, ou de forma mais geral, $n-q_r+f+1$ servidores, uma operação de leitura subseqüente poderia não retornar t já que até f servidores podem mentir durante a operação de leitura, negando a existência da tupla cuja notificação foi consolidada. Esta característica intrigante do BTS, de que é **impossível** construir um mecanismo de notificação que **garanta** a existência de uma tupla, está relacionada ao fato de sua construção usar protocolos não confirmáveis e prover apenas semântica não ordenada para a operação *out*. Se a construção desse mecanismo fosse possível, a operação *out* poderia ser construída com semântica confirmável no BTS [28], o que não é verdade, dadas as características dos protocolos.

Um cliente p que deseja parar de receber notificações dos servidores deve executar a operação $unregister(\bar{t})$. Esta operação se baseia também em um protocolo não confirmável onde p envia uma mensagem $\langle \text{UNREGISTER},\bar{t}\rangle$ a todos os servidores, que ao receberem-na, simplesmente descartam o par $\langle p,\bar{t}\rangle$ de seu conjunto *ToNotify*.

A introdução do suporte de notificações provê **desacoplamento de bloqueio** (ver seção 3.2) ao modelo de coordenação generativo, i.e. os clientes que buscam informações no espaço de tuplas não precisam realizar acessos bloqueantes ao espaço (os clientes não precisam parar suas linhas de execução enquanto acessam o espaço de tuplas). Apesar de oferecer poucas vantagens em termos qualitativos ao modelo de coordenação[16], a adição deste tipo de desacoplamento torna o espaço de tuplas tão poderoso, em termos de desacoplamento, quanto o modelo de comunicação *publish/subscribe* [55].

Utilizando este suporte de notificação, as operações bloqueantes podem ser implementadas quase que diretamente. O algoritmo a ser seguido por um cliente p que executa a operação $rd(\bar{t})$ (resp. $in(\bar{t})$) é o seguinte: p começa executando $register(\bar{t})$ para registrar seu interesse em tuplas que combinam com $\bar{t}$. Quando a notificação de uma tupla t ($m(t,\bar{t})$) é consolidada por p, ele executa $rdp(\bar{t})$ (resp. $inp(\bar{t})$) e, se bem sucedido (resultado $\neq \perp$), executa $unregister(\bar{t})$ e a operação termina tendo o resultado de $rdp(\bar{t})$ (resp. $inp(\bar{t})$) como seu resultado. Caso contrário (resultado da leitura $= \perp$), p permanece esperando outras notificações para então executar novamente a versão não bloqueante

[15] Se existirem várias tuplas que combinam com o molde registrado, o servidor pode mandá-las todas na mesma mensagem.

[16] Na opinião do autor, este tipo de desacoplamento apenas torna o modelo mais complexo e menos elegante.

do protocolo. Note que as operações bloqueantes só terminam quando as operações não bloqueantes conseguem obter uma tupla no espaço. Esta característica é importante uma vez que garante que as mesmas propriedades de *safety* (ex. linearização no LBTS) providas pelas operações não bloqueantes valem para as operações bloqueantes.

4.6.5 Limitando a Memória Utilizada no LBTS

Na descrição do LBTS, foi mencionado que o conjunto R_s armazena todas as tuplas removidas do espaço. Em execuções de longa duração, onde um número grande de tuplas é incluído e removido do espaço de tuplas, este conjunto tende a crescer indefinidamente. Apesar de na prática podermos limpar este conjunto periodicamente, já que inserções e remoções não podem demorar um tempo infinito[17], em termos teóricos, isto exige memória infinita dos servidores.

Esta seção apresenta uma estratégia para remoção de tuplas dos conjuntos R quando estas já forem completamente removidas dos servidores. O esquema se baseia em duas modificações básicas nos protocolos visando implementar controles específicos que substituam a memória infinita em cada uma de suas necessidades:

- O primeiro uso da memória infinita é evitar que clientes executando um leitura reescrevam uma tupla que está sendo removida concorrentemente nos servidores que já a removeram (linha 22 do algoritmo 7). Desta forma, o requisito aqui é evitar que tuplas sendo removidas sejam re-escritas. Isto pode ser implementado fazendo-se com que cada servidor armazene uma tupla removida t enquanto houverem clientes realizando leituras que se iniciaram antes da remoção de t no servidor s. Isto implica que, uma tupla removida t fica no conjunto R_s enquanto houverem clientes em L_s (conjunto dos *listeners*) que iniciaram suas leituras em s (bloco com as linhas 1-16 do algoritmo 7) antes da remoção de t de s (bloco com as linhas 14-23 do algoritmo 8). Desta forma, o conjunto R_s pode ser "limpo" sempre que s processar uma mensagem RDP-COMPLETE (linha 20 do algoritmo 7).

- O segundo uso para memória infinita no LBTS é o impedimento de que tuplas sendo inseridas sejam removidas mais de uma vez. Esta é a mesma motivação para o conjunto de tuplas a serem removidas do BTS, e a mesma solução deve ser empregada aqui. A idéia é fazer com que uma tupla removida que não estava presente no espaço local de um servidor s quando de sua remoção, **só possa ser removida** do conjunto R_s quando não houverem leituras concorrentes à remoção sendo executadas e quando a tupla for recebida pelo servidor (ver algoritmo 3).

Estas duas modificações tornam a quantidade de tuplas presentes no LBTS dependente da quantidade de escritas concorrentes às remoções sendo executadas e da quantidade de tuplas fantasma do sistema.

[17] Lembrar que as tuplas removidas são armazenadas apenas com o intuito de que elas não sejam removidas mais de uma vez durante sua inserção e/ou remoção (ver seção 4.4.1).

4.6.6 Limitando Operações *out* de Clientes Faltosos

Os protocolos do BTS e do LBTS descritos neste capítulo permitem que clientes maliciosos insiram um número ilimitado de tuplas fantasma no espaço. No caso do BTS não existe alteração possível no protocolo que elimine esta limitação e mantenha as mesmas complexidade de mensagens e passos de comunicação do protocolo *out* (ver seção 4.5). Para o LBTS, no entanto, podemos utilizar **justificativas de escritas**, de forma similar a usada em [80], para modificar o protocolo *out* visando limitar em 1 o número de inserções incompletas que um cliente pode fazer.

O ponto central desta modificação é o uso de justificativas na inserção, de maneira similar à usada nas reescritas. Uma justificativa de inserção é um conjunto contendo mensagens ACK-OUT assinadas provenientes de um quórum de servidores. Estas mensagens correspondem as respostas de uma requisição OUT para inserção de uma tupla. Cada inserção deve ter um número de seqüência provido pelo cliente. Cada servidor armazena o maior número de seqüência enviado por cada cliente em um vetor de contadores out_count, com uma entrada para cada cliente. Para inserir uma nova tupla, o cliente precisa enviar uma mensagem OUT com um número de seqüência maior que o último usado por ele, juntamente com uma justificativa de escrita contendo as respostas da última escrita completada (se essa não for a primeira). Um servidor aceita uma inserção (e executa o bloco com as linhas 3-6 do algoritmo 6) se o número de seqüência da requisição do cliente c for for maior que $out\_count[c]$ e c apresentou uma justificativa contendo mensagens assinadas de um quórum de servidores para a inserção anterior (de número de seqüência $out\_count[c]$). Na primeira inserção, o cliente deve enviar um número de seqüência 1 e não precisa apresentar justificativa de inserção.

A principal implicação desta modificação é que ela requer que todas as mensagens ACK-OUT sejam assinadas pelos servidores. O uso de criptografia de chave pública nestas assinaturas poderia elevar em muito a latência da operação de escrita, especialmente em redes locais [22]. No entanto, podemos utilizar vetores de MACs da mesma forma que em [36] para implementar estas assinaturas. Desta forma, a justificativa de inserção é válida para um servidor s se e somente se ela contém $q - f$ mensagens corretamente autenticadas para esse servidor (o campo com o MAC correspondente a este servidor está correto). Esta modificação praticamente elimina a latência das assinaturas.

Outra implicação desta modificação é o requisito de canais FIFO entre os clientes e os servidores, o que não era necessário na versão original do LBTS.

Note a modificação para limitação das escritas de clientes maliciosos apresentada nesta seção pode ser implementada em uma camada subjacente do algoritmo de escrita do LBTS, não requerendo portanto nenhuma alteração no protocolo do algoritmo 6.

4.7 Trabalhos Relacionados

Existem duas técnicas de replicação para a construção de serviços tolerantes a faltas bizantinas: sistemas de quóruns bizantinos [86] (ver seção 2.4.2) e replicação máquina de estados [36, 114] (ver seção 2.4.1). A primeira é uma abordagem centrada em dados e baseada na execução de diferentes

operações em diferentes conjuntos de servidores que se intersectam (chamados quóruns), enquanto a segunda é baseada na manutenção de um estado consistente entre as várias réplicas do sistema através da execução da mesma seqüência de operações. Uma vantagem dos sistemas de quóruns em relação à replicação Máquina de Estados reside no fato de que suas operações não precisam ser executadas na mesma ordem em todos os servidores, não requerendo portanto a resolução do problema de consenso. Adicionalmente, os protocolos para quóruns tendem a ser muito mais escaláveis devido a oportunidade para concorrência na execução das operações e a mudança do trabalho de processamento mais pesado dos servidores para os processos clientes.

As construções apresentadas neste capítulo utilizam sistemas de quóruns bizantinos e provêem protocolos específicos para cada uma das operações do objeto de memória compartilhada espaço de tuplas. Apenas uma das operações providas por estas construções, *inp*, requer a resolução de consenso em seus protocolos, necessitando portanto de mais trocas de mensagens entre os servidores e premissas de sincronismo para sua terminação (sincronia terminal - ver seção 2.1.3). Este requisito se justifica pelo fato do espaço de tuplas ter número de consenso 2 [117] e portanto não poder ser implementado de maneira determinista satisfazendo terminação livre de espera em sistemas completamente assíncronos. Até onde sabemos, apenas o trabalho FT-SCALABILITY [1] faz uso de sistemas de quóruns bizantinos na implementação de objetos mais poderosos que registradores, de forma semelhante a usada na implementação do BTS e do LBTS. Este trabalho propõem um esquema de replicação genérico (pode ser utilizado para implementar qualquer serviço) para sistemas assíncronos sujeitos a faltas bizantinas que se baseia inteiramente em protocolos para sistemas de quóruns. Como isto não pode ser implementado satisfazendo liberdade de espera, o FT-SCALABILITY sacrifica a vivacidade: as operações satisfazem apenas **liberdade de obstrução** [68], i.e. uma operação é garantida de terminar apenas se não existem outras operações sendo executadas concorrentemente a ela. Desta forma, o FT-SCALABILITY não provê operações livres de espera, mas permite a implementação de objetos mais fortes que registradores em sistemas assíncronos. O (L)BTS, por outro lado, garante a liberdade de espera em todas as operações assumindo sincronia parcial (que acreditamos ser um modelo bastante realista). Quando comparado com a nossa abordagem , o FT-SCALABILITY apresenta pelo menos duas desvantagens: *(i.)* ele satisfaz apenas liberdade de obstrução, o que em ambientes sujeitos a faltas bizantinas pode ser muito perigoso, já que clientes maliciosos podem invocar operações continuamente no sistema, causando negação de serviço facilmente; e *(ii.)* ele requer $n \geq 5f + 1$ servidores, uma resistência sub-ótima que tem um alto impacto no custo do sistema quando levamos em consideração a provisão de independência de falhas através de diversidade [99].

A grande maioria dos trabalhos sobre sistemas de quóruns bizantinos se concentra na implementação de operações de leitura e escrita para registradores (ex. [35, 64, 80, 86, 87, 92, 93]). No entanto, existem alguns outros trabalhos nesta linha que propõem objetos de memória compartilhada diferentes. Em [13] é proposto um objeto que fornece *non-skipping timestamps*. Este objeto pode ser utilizado para eliminar uma conhecida vulnerabilidade dos protocolos para sistemas de quóruns bizantinos. Este tipo de objeto é equivalente a um registrador *fetch&add*, que sabidamente tem número de consenso 2 [67]. Entretanto, para implementar este objeto usando sistemas de quóruns, sua especificação foi enfraquecida de tal forma que o objeto resultante tenha número de consenso 1 (como um registrador). O sistema para autoridade certificadora COCA [129] provê uma operação para atualização de

chaves públicas associadas a certificados usando apenas protocolos para sistemas de quóruns bizantinos. Sabendo-se que as operações de atualização (*read-write*) no sentido da hierarquia livre de espera têm número de consenso maior ou igual a 2 [67], pode-se pensar que esta operação dá ao serviço implementado pelo COCA um número de consenso maior que 1. No entanto, a operação de atualização do COCA é na verdade uma operação de escrita (onde a nova chave pública representa o valor) em um certificado (que funciona como um registrador), e portanto, o serviço provido é equivalente a um registrador. Alguns trabalhos propõem objetos de consenso baseados em registradores usando aleatoriedade (ex. [88]) ou detectores de falhas (ex. [3]). Estes trabalhos diferem fundamentalmente do apresentado neste capítulo por usarem objetos baseados em sistemas de quóruns (registradores) para implementar consenso enquanto neste capítulo usamos consenso (protocolo PAXOS BIZANTINO) na implementação de objetos mais elaborados (espaço de tuplas). Além disso, os algoritmos de consenso providos nestes trabalhos não são uniformes, i.e. os processos precisam conhecer uns aos outros [9], um problema quando consideramos sistemas abertos.

Conforme apresentado na seção 3.6, existem vários trabalhos que exploram replicação para construção de espaços de tuplas tolerantes a faltas. Alguns deles são baseados em replicação máquina de estados (ex. [11]) enquanto outros usam sistemas de quóruns (ex. [128]). Entretanto, nenhuma destas propostas consideram a ocorrência de falhas bizantinas, foco principal do BTS e do LBTS.

4.8 Considerações Finais

Este capítulo apresentou duas construções para espaços de tuplas tolerantes a faltas bizantinas baseados em sistemas de quóruns bizantinos: o BTS e o LBTS. Estas construções se baseiam no uso de protocolos leves (baixa complexidade de mensagens e poucos passos de comunicação) para sistemas de quóruns na implementação de operações simples de leitura e escrita (*rdp* e *out*) e no uso do algoritmo de consenso PAXOS BIZANTINO para implementação de operações de leitura-escrita (*inp*). Devido a esta filosofia de desenvolvimento, estes protocolos apresentam importantes vantagens em termos de métricas comumente utilizadas na avaliação de algoritmos distribuídos se comparados a construções equivalentes baseadas exclusivamente em replicação Máquina de Estados.

As principais contribuições apresentadas neste capítulo são: **(1)** o BTS é o primeiro espaço de tuplas tolerante a faltas bizantinas publicado [21] e, mais importante, **(2)** estes protocolos compreendem as primeiras construções de um objeto de memória compartilhada estritamente mais forte (em termos da hierarquia livre de espera [67]) que um registrador [117] que satisfazem terminação livre de espera implementadas a partir de sistemas de quóruns.

Capítulo 5

Confidencialidade em Espaços de Tuplas Tolerantes a Faltas Bizantinas

Nos capítulos anteriores foram introduzidas diversas construções para espaços de tuplas tolerante a faltas bizantinas. Estas construções tratam basicamente da manutenção da integridade e da disponibilidade do espaço de tuplas enquanto serviço. Neste capítulo consideramos o problema da segurança e apresentamos um esquema de confidencialidade para espaços de tuplas tolerantes a intrusões baseado em compartilhamento de segredo. Este esquema garante que os dados armazenados no espaço de tuplas não sejam revelados para partes não autorizadas mesmo que algumas réplicas do espaço sejam faltosas. Além disso, são discutidas formas para integração deste esquema nos espaços de tuplas tolerantes a faltas maliciosas previamente apresentados. A validação do esquema é feita através da implementação de um protótipo e a realização de alguns experimentos para medição do impacto do mecanismo de confidencialidade na latência de uma das construções introduzidas neste trabalho.

5.1 Contexto e Motivação

A utilização de replicação para manter a disponibilidade e a integridade de um sistema é muitas vezes vista como um impedimento para a implementação de confidencialidade no mesmo. A razão para isto é simples: se uma informação secreta não é armazenada em um, mas em vários servidores, é possivelmente mais fácil para um atacante obtê-la (não mais difícil). Implementar confidencialidade em um espaço de tuplas replicado não é simples por diversas razões. A primeira, e mais importante, é que não podemos confiar nos servidores individualmente para garantir a confidencialidade das tuplas, uma vez que até f servidores podem ser maliciosos, revelando as tuplas armazenadas a processos não autorizados. A segunda complicação advém da necessidade de combinação entre tuplas e moldes. Campos cifrados geralmente não podem ser comparados com campos de um molde. Além disso, algumas políticas de acesso de granularidade fina [18] (ver próximo capítulo) podem impor limites nos campos que podem ser cifrados. Várias soluções aparentemente simples não funcionam neste cenário ou são simplesmente inaceitáveis quando pensamos no modelo de coordenação baseado em espaço de tuplas. Duas das mais intuitivas destas soluções seriam:

- **Fazer com que cada cliente cifrasse os campos das tuplas inseridos por ele usando uma chave secreta compartilhada ou sua chave privada**: o problema desta solução está no fato de que ela requer que todos os clientes que queiram ler e/ou remover uma tupla cifrada conheçam a chave secreta ou as chaves pública (para que campos cifrados possam ser decifrados) e privada (para cifrar campos de moldes e as regras de combinação possam ser usadas). Além do problema prático da dificuldade em se realizar a distribuição destas chaves, esta solução acaba com a propriedade de anonimato do modelo de coordenação baseado em espaço de tuplas, que define que os clientes não precisam conhecer uns aos outros para interagirem;
- **Cifrar a comunicação cliente-servidores e deixar que estes últimos cifrem os campos das tuplas com suas próprias chaves**: Esta solução não é aceitável em nosso modelo de sistema porque queremos manter a confidencialidade mesmo que alguns servidores sejam controlados pelo adversário, e estes servidores poderiam decifrar e revelar o conteúdo de campos confidenciais das tuplas.

Note que os requisitos inerentes a um espaço de tuplas tolerante a intrusões, especialmente a possibilidade de alguns servidores estarem completamente sobre controle de um adversário malicioso, torna todos os mecanismos de segurança de um espaço de tuplas propostos previamente [26, 127] de pouca utilidade, uma vez que, para o bom funcionamento destes é necessário que se assuma que os servidores são confiáveis.

Desta forma, a solução para a provisão de confidencialidade em espaço de tuplas tolerante a faltas bizantina passa pela utilização de uma primitiva de compartilhamento de segredo [24, 118]. A idéia é gerar fragmentos da tupla e armazenar cada um deles em um servidor diferente. A confidencialidade é garantida devido a propriedade de que a reconstrução da tupla só pode ser feita com a colaboração de $f+1$ servidores. Isto significa que mesmo que f servidores do sistema sejam controlados pelo adversário, é impossível que este consiga reconstruir uma tupla armazenada segundo o esquema de confidencialidade. Conseqüentemente, os mecanismos de controle de acesso definidos para o espaço de tuplas seriam respeitados, mesmo na existência de servidores falhos. Tendo em vista que os clientes também podem ser maliciosos, e que as tuplas são acessadas a partir de seu conteúdo, mecanismos adicionais são necessários, conforme será discutido no decorrer deste capítulo.

5.2 Criptografia de Limiar

Um problema bastante comum nas aplicações que usam métodos de criptografia é como manter a segurança das chaves criptográficas e outros segredos usadas pelas partes para se comunicar. Esta segurança se refere tanto à manutenção da confidencialidade (manutenção de chaves secretas ou privadas fora do alcance de partes não autorizadas) quanto da disponibilidade (impedindo que chaves sejam destruídas por partes maliciosas) destas chaves. A **criptografia de limiar** [24, 46, 118] minimiza este problema pois permite que uma chave (segredo) seja "dividida" entre várias partes.

Em um esquema de criptografia de limiar bastante simples, chamado **compartilhamento de segredo**, um segredo S é dividido entre n partes, cada uma recebendo um fragmento do segredo de tal

forma que S só pode ser reconstruído se um quórum mínimo de k partes ($0 < k \leq n$) participar com seus fragmentos do segredo. Desta forma, qualquer conjunto de $m < k$ partes (com suas frações do segredo) não consegue obter informação alguma sobre S. Um esquema deste tipo é dito um esquema de limiar (n,k).

Os primeiros esquemas de limiar propostos na literatura [24, 118] propunham esquemas de compartilhamento de segredo baseados em interpolação polinomial e geometria plana (intersecção de retas). Estes esquemas requerem um entregador honesto na inicialização da computação para gerar os fragmentos do segredo e distribuí-los aos n participantes envolvidos. Além disso, estes esquemas também requerem que as partes entreguem seus fragmentos dos segredos para um combinador honesto que vai recriar o segredo. Se esse segredo for, por exemplo, uma chave privada usada para assinatura de mensagens por um serviço replicado, algum processo receberá todos os fragmentos da chave e recuperará a mesma, assinando a mensagem requisitada. Este mesmo processo pode também assinar qualquer outra mensagem, uma vez que ele tem a chave combinada. Para contornar este problema foram criadas as funções de limiar [62], que permitem que cada possuidor de um fragmento do segredo distribuído faça um cálculo parcial, como por exemplo uma assinatura parcial, a partir de seu fragmento de tal forma que estes cálculos parciais possam ser combinados para gerar o objeto desejado (ex. a assinatura) sem que nenhum fragmento do segredo seja revelado. Este tipo de função é muito usada em esquemas de **assinatura de limiar** [47], em que k fragmentos de assinatura gerados a partir de diferentes pedaços da chave privada, e **criptografia de limiar** [90], onde o texto limpo associado a um texto cifrado só é revelado a partir de k decifragens parciais do texto cifrado realizadas com fragmentos diferentes da chave.

Para lidar com cenários onde as partes podem ser maliciosas e agir ativamente contra os protocolos (adversário ativo), foram criados os mecanismos de **compartilhamento de segredo verificável** (VSS), em que cada fragmento de um segredo ou computação fornecido ao combinador pode ter sua validade verificada [40]. Estes esquemas são fundamentais para que o resultado de uma computação em criptografia de limiar não seja alterado devido a resultados e valores incorretos fornecidos por partes maliciosas.

Um tipo especialmente interessante de esquema de compartilhamento de segredos é o **compartilhamento de segredo verificável publicamente** (*publicly verifiable secret sharing scheme* - PVSS) [116]. Este tipo de esquema funciona de forma semelhante ao VSS, porém, não apenas os n portadores de segredos podem verificar a validade dos mesmos, mas sim qualquer parte que interaja no protocolo (por isso o "verificável publicamente"). O PVSS é o tipo de esquema escolhido na concretização do mecanismo de confidencialidade apresentado neste capítulo.

5.3 Premissas e Propriedades do Mecanismo

O mecanismo proposto neste capítulo assume que as tuplas armazenadas no espaço de tuplas estão protegidas por um mecanismo de controle de acesso em nível de espaço de tuplas e em nível de tuplas. Este mecanismo é descrito na seção 3.5.3. Desta forma, cada tupla tem associada à ela um conjunto de credenciais que um cliente deve apresentar para ter acesso ao seu conteúdo.

A principal propriedade garantida pelo mecanismo é: uma tupla t com credenciais de leitura C_t, só terá seu conteúdo privado revelado para clientes que apresentem credencias que satisfaçam C_t. Esta propriedade é garantida mesmo na presença de até f servidores maliciosos.

5.3.1 Compartilhamento de Segredo Verificável Publicamente

O principal esquema criptográfico utilizado no mecanismo de confidencialidade proposto é o **compartilhamento de segredo verificável publicamente** (*publicly verifiable secret sharing scheme* - PVSS) [116]. Este esquema é utilizado para garantir a confidencialidade das tuplas armazenadas no espaço replicado mesmo em face a existência de servidores maliciosos. Em um esquema (n,k)-PVSS, um **distribuidor** (*dealer*) distribui fragmentos (*shares*) de um segredo para n partes distintas, chamadas **portadores** (*share holders*), usando um protocolo de distribuição. Posteriormente, um **combinador** (*combiner*) obtém pelo menos k fragmentos para reconstruir o segredo. O esquema é dito "verificável publicamente" porque qualquer parte pode verificar se os fragmentos estão corrompidos (não apenas os n portadores). Uma propriedade fundamental deste tipo de esquema é que nenhuma informação sobre o segredo compartilhado é obtida com menos de k fragmentos. Este esquema é baseado nas seguintes primitivas (denotamos por x_i e y_i, respectivamente, as chaves privadas e públicas de cada portador i):

- $\mathsf{share}(y_1,...,y_n,s)$ é usada para gerar os n fragmentos $\{s_1,...,s_n\}$ do segredo s e a prova criptográfica $PROOF_s$ de que estes segredos são corretos;
- $\mathsf{verifyD}(s_i,x_i,PROOF_s)$ é usada pelo portador i para verificar se o fragmento s_i distribuído pelo distribuidor é correto;
- $\mathsf{prove}(s_i,x_i,PROOF_s)$ é usado para gerar uma prova criptográfica $PROOF_s^i$, a qual é usada para provar que o fragmento s_i, apresentado pelo portador i para o combinador, é correto;
- $\mathsf{verifyS}(s_i,y_i,PROOF_s,PROOF_s^i)$ é usada para verificar se um fragmento s_i obtido do portador i é correto em relação ao segredo distribuído (representado por $PROOF_s$).
- $\mathsf{combine}(s_1,...,s_k)$ é usada para combinar um conjunto de k fragmentos válidos e recuperar o segredo s.

Para a implementação do esquema de confidencialidade no espaço de tuplas replicado, utilizamos o esquema criptográfico proposto em [116]. A segurança deste esquema se baseia nas premissas do problema Diffie-Helman [51] e ele é provado seguro no modelo dos oráculos aleatórios [14].

5.4 Visão Geral do Mecanismo de Confidencialidade

Conforme já discutido, a solução proposta neste capítulo segue a idéia de deixar os servidores cuidarem da confidencialidade, entretanto, ao invés de confiar nos servidores individualmente, confiamos em um **conjunto** de servidores, seguindo o paradigma de **confiança distribuída** [115]. Usamos

o protocolo de compartilhamento de segredo verificável publicamente apresentado na seção 5.3.1, considerando $k = f + 1$. Nele, cada servidor de espaço de tuplas s_i têm uma chave privada x_i e uma chave pública y_i. Os clientes conhecem as chaves públicas de todos os servidores e fazem o papel dos distribuidores do protocolo, obtendo um conjunto de fragmentos da tupla (função share). Todo campo da tupla pode ser decifrado com $k = f + 1$ fragmentos válidos através da função combine, portanto uma coalisão de servidores maliciosos não pode revelar o conteúdo de um campo confidencial (assumimos no máximo f servidores falhos).

Além de garantir a confidencialidade da tupla mesmo com uma parte dos servidores controlada pelo adversário, um esquema de confidencialidade para espaço de tuplas deve garantir o acesso por conteúdo às tuplas (através da regra de combinação), mesmo na existência de campos cifrados. Quando um cliente invoca a operação $out(t)$, ele escolhe um dos três tipos de proteção para cada um dos campos de t:

- **público**: o campo não é cifrado e portanto seu valor pode ser comparado arbitrariamente (p.ex. se ele pertence a um intervalo) e revelado a partes maliciosas[1];
- **comparável**: o campo é cifrado e um resumo criptográfico de seu valor é armazenado juntamente com a tupla de tal forma que as comparações de igualdade (requeridas na verificação de combinações) sejam possíveis;
- **privado**: o campo é cifrado e nenhuma informação derivada dele é armazenada, logo seu conteúdo não pode ser alvo de comparações.

Dada uma tupla t, definimos seu **vetor de tipos de proteção** v_t como um seqüência de tipos de proteção para cada um dos campos de t. Os possíveis valores para os campos de um vetor de tipos de proteção são PU, CO e PR, para indicar que o campo correspondente da tupla é público, comparável ou privado, respectivamente. Assim, cada campo de v_t descreve o tipo de proteção a ser aplicado para o campo correspondente da tupla t. Por exemplo, se um tupla $t = \langle 7, 8 \rangle$ tem um vetor de tipos de proteção $v_t = \langle CO, PR \rangle$, sabemos que o primeiro campo (valor 7) é comparável e o segundo (valor 8) é privado.

A idéia por trás dos campos marcados como comparáveis é a seguinte. Suponha que o cliente p_1 queira inserir uma tupla t no espaço com um único campo comparável f_1. Assumindo a existência de uma função de resumo resistente a colisão H, p_1 envia t cifrada juntamente com $\mathsf{H}(f_1)$ aos servidores. Suponha que mais tarde um cliente p_2 invoque $rd(\bar{t})$ e que o espaço de tuplas precise verificar se $\bar{t}$ combina com t. Assumindo que $\bar{t}$ tenha um único campo $\overline{f}_1$, p_2 calcula $\mathsf{H}(\overline{f}_1)$ e envia este valor aos servidores do espaço de tuplas que verificam se este resumo combina com $\mathsf{H}(f_1)$. Este mecanismo funciona para o regra de combinação do modelo generativo (comparações de igualdade – definida na seção 3.3), porém claramente não funciona com outras comparações como as inequalidades que podem ser necessárias na verificação de políticas[2] (ver seção 3.5.4 e próximo capítulo). Além disso, este

[1] Este tipo de campo pode ser necessário, por exemplo, para o emprego de políticas de acesso de granularidade fina (ver próximo capítulo).

[2] Para que funcionasse, seria necessária uma função de resumo que mantivesse o resultado da comparação entre valores quando seus resumos fossem comparados. Por exemplo, para inequalidades, se $v > v'$ então $\mathsf{H}(v) > \mathsf{H}(v')$.

mecanismo tem outra limitação: apesar das funções de resumo serem unidirecionais, se o domínio dos valores que um campo pode conter é conhecido e limitado, um ataque de força bruta pode revelar seu conteúdo. Por exemplo, suponha que um campo pode conter apenas valores de 32 bits. Um atacante pode simplesmente calcular os resumos de 2^{32} possíveis valores para descobrir qual o valor do campo armazenado. Devido a esta limitação, recomenda-se que o esquema seja usado apenas com tuplas não tipadas (todos os campos têm um mesmo tipo), ou com tuplas cujos campos sejam de tipos com domínio arbitrariamente grande (o tipo *string*). Adicionalmente, a limitação dos campos comparáveis é a razão pela qual definimos os campos privados: nenhum resumo é enviado aos servidores e portanto comparações são impossíveis, garantindo a confidencialidade destes campos.

5.5 Algoritmo Abstrato

A idéia principal do esquema de confidencialidade é fazer com que o processo que insere a tupla no espaço atue como um distribuidor no esquema PVSS, gerando n fragmentos da tupla e distribuíndo-os entre os servidores. Um processo que deseja ler uma tupla, coleta $f+1$ destes fragmentos e combina-os para obter a tupla. Desta forma, os processos clientes do espaço de tuplas representam os papéis de distribuidores e combinadores, enquanto os servidores são os portadores dos fragmentos dos segredos. É necessário um esquema de verificação pública para permitir que um cliente verifique se o fragmento retornado pelo servidor é correto (nos esquemas de compartilhamento de segredos verificável "normais" apenas os portadores, os servidores neste caso, podem verificar se um fragmento está corrompido). Isto é muito importante uma vez que estamos considerando um sistema aberto com um número ilimitado de clientes.

Antes de apresentarmos os algoritmos para inserção e leitura de tuplas, vamos definir o conceito de *fingerprint*. Dada uma tupla t, o *fingerprint* t_h de t é uma tupla contendo o mesmo número de campos de t, onde os valores de cada um destes campos corresponde ao valor do campo de t, sem porém revelar informações sobre campos comparáveis e privados. A função fingerprint calcula o *fingerprint* de uma tupla baseando-se na tupla e no vetor de tipos de proteção associado.

$$\mathsf{fingerprint}(\langle f_1,...,f_m\rangle,\langle pt_1,...,pt_m\rangle) \triangleq \langle \mathsf{F}(f_1),...,\mathsf{F}(f_m)\rangle, \text{ onde}$$

$$\mathsf{F}(f_i) = \begin{cases} * & , \text{se } f_i = * \\ f_i & , \text{se } \mathit{formal}(f_i) \text{ ou } pt_i = PU \\ \mathsf{H}(f_i) & , \text{se } pt_i = CO \\ \text{PR} & , \text{se } pt_i = PR \end{cases}$$

Dado um vetor de tipos de proteção v_t da tupla t conhecido (e usado) tanto pelos processos que inserem t quanto pelos seus leitores[3], a definição do *fingerprint* da tupla garante que se um molde $\bar{t}$ combina com a tupla t, i.e. $m(t,\bar{t})$, então o *fingerprint* de $\bar{t}$, $\bar{t}_h$ combina com o *fingerprint* de t, t_h, i.e. $m(t_h,\bar{t}_h)$. Lembrar que esta regra vale apenas se t_h e $\bar{t}_h$ são gerados usando o mesmo vetor de tipos de proteção.

[3]Este vetor seria definido no contexto da aplicação que faz uso do espaço de tuplas.

A partir da definição de *fingerprint* apresentada acima, é possível montar um mecanismo de confidencialidade que garanta que nenhum servidor conheça o conteúdo de um tupla (pelo menos não de seus campos privados e comparáveis), e mesmo assim possa aplicar a regra de combinação entre entradas e moldes, que permite acesso a dados por conteúdo.

Os algoritmos 9 e 10 descrevem os algoritmo de confidencialidade para a escrita e leitura de tuplas, respectivamente.

Algoritmo 9 Inserção da tupla t (cliente p e servidor s)

$\{Cliente\}$

1. p gera n fragmentos $\{ts_1,...,ts_n\}$ e a prova de correção $PROOF_t$, usando a função $\mathsf{share}(y_1,...,y_n,t)$.
2. p computa o *fingerprint* t_h da tupla t usando o vetor de proteção v_t através de $t_h = \mathsf{fingerprint}(t,v_t)$.
3. p envia uma mensagem cifrada $\langle ts_s,t_h,PROOF_t\rangle$ para cada servidor s do sistema[4].

$\{Servidor\}$

1. Um servidor s aceita a tupla t inserida por p, se $\mathsf{verifyD}(ts_s,x_s,PROOF_t)$ retornar *true*.
2. Após aceitar a tupla t, s calcula $PROOF_t^s = \mathsf{prove}(ts_i,x_s,PROOF_t)$ e armazena $\langle ts_s,t_h,PROOF_t,PROOF_t^s,p\rangle$.

O algoritmo de inserção da tupla consiste basicamente na geração do *fingerprint* da tupla e da geração e distribuição dos fragmentos da tupla a ser inserida. Um servidor só aceita a tupla se o fragmento recebido for validado de acordo com a prova enviada pelo cliente. Uma vez a tupla sendo aceita, o servidor gera uma segunda prova e armazena todo o conjunto de dados recebidos mais essa nova prova e o identificador do cliente que inseriu a tupla.

O algoritmo de leitura é também bastante simples: um cliente envia o *fingerprint* do molde usado na leitura aos servidores e coleta $f+1$ respostas com fragmentos válidos e os mesmos valores para t_h e $PROOF_t$. A partir desses $f+1$ fragmentos é possível reconstruir a tupla t. Se o *fingerprint* de t não for igual a t_h o cliente notifica os servidores de que esta tupla é inválida. Os servidores, ao verificarem que realmente t é inválida, colocam o processo que a inseriu em uma lista negra e não aceitam mais inserções advindas deste processo. Este algoritmo pode ser usado também para encontrar tuplas em operações de remoção (*in* e *inp*).

É importante notar que tanto no algoritmo de inserção quanto no algoritmo de leitura todas as mensagens que contém fragmentos da tupla sendo inserida/lida são cifradas usando uma chave compartilhada entre o cliente e o servidores. Isto evita que algum adversário malicioso intercepte as mensagens trocadas entre o cliente e os servidores de tal forma a obter $f+1$ fragmentos e ter acesso a uma tupla secreta, já que o modelo de sistema usado neste trabalho não prevê canais confidenciais (ver seção 2.1.4).

Existem vários pontos sobre o esquema de confidencialidade que merecem alguma discussão.

Algoritmo 10 Leitura da tupla t (cliente p e servidor s)

$\{Cliente\}$

1. p computa o *fingerprint* $\bar{t}_h$ do molde $\bar{t}$ usando o vetor de proteção v_t através de $\bar{t}_h = \mathsf{fingerprint}(\bar{t}_h, v_t)$.
2. p envia uma requisição de leitura aos servidores, utilizando $\bar{t}_h$ como molde.
3. p recebe as respostas assinadas e cifradas de um conjunto de servidores e escolhe uma tupla t que possui $f+1$ fragmentos corretos, i.e., $f+1$ servidores responderam os mesmos t_h e $PROOF_t$ e, para cada servidor s, sua prova satisfaz $\mathsf{verifyS}(ts_s, y_s, PROOF_t, PROOF_t^s)$.
4. O processo p combina os $f+1$ fragmentos corretos recebidos no passo anterior usando $\mathsf{combine}(ts_1, ..., ts_{f+1})$ e obtém a tupla t.
5. p verifica se $t_h = \mathsf{fingerprint}(\mathsf{t}, \mathsf{v_t})$. Caso esta igualdade se verifique, a operação termina e seu resultado é t.
6. Caso esta igualdade não se verifique, p deve avisar aos servidores que a tupla t foi inserida de forma inválida (seu *fingerprint* não foi calculado corretamente). Para isso, p envia uma mensagem invalidando a tupla para todos os servidores, com todos as mensagens assinadas coletadas durante a leitura, para provar que t está corrompida.
7. O processo p escolhe outra tupla para ser lida (ou retorna $\perp$ se não existir outra tupla). Esta escolha se dará através da re-execução da operação de leitura.

$\{Servidor\}$

1. Ao receber a requisição de leitura, cada servidor s retorna uma mensagem assinada e cifrada contendo seqüências de dados da forma $\langle ts_s, t_h, PROOF_t, PROOF_t^s \rangle$ para cada *fingerprint* t_h que combine com $\bar{t}_h$.
2. Ao receber a mensagem invalidando uma tupla t, s verifica se a invalidação é coerente: t_h não é o *fingerprint* de t e a tupla foi gerada a partir de $f+1$ fragmentos provenientes de respostas assinadas pelos servidores. Se sim, a tuple é removida (caso já não tenha sido) e o processo que a inseriu é colocado em uma "lista negra" (suas futuras inserções serão ignoradas).

Em primeiro lugar, a maior parte do processamento, como a geração e combinação dos fragmentos do segredo, é realizada no processo cliente. Deste modo, a utilização deste esquema não deve causar impactos severos na escalabilidade de um sistema.

O segundo ponto, também relacionado com a escalabilidade, é que não usamos nenhum protocolo com complexidade de troca de mensagens quadrática, os quais não apresentam uma boa escalabilidade. O mecanismo é baseado em protocolos que apresentam uma complexidade de mensagens $O(n)$ (o protocolo não requer que os servidores se comuniquem entre si).

5.6 Integrando o Esquema nos Espaços de Tuplas

Este seção descreve como os protocolos de escrita e leitura, anteriormente apresentados, podem ser integrados nos espaços de tuplas definidos nesta tese.

5.6.1 Replicação Máquina de Estados

A aplicação do esquema de confidencialidade em um espaço de tuplas com este tipo de replicação é relativamente simples e direta, no entanto, alguns cuidados a mais são necessários. Devido a necessidade de acordo no estado das réplicas, todas as requisições ao serviço devem ser feitas utilizando uma primitiva de difusão com ordem total [36, 115]. Desta forma, não é possível enviar diferentes versões de uma requisição para diferentes servidores (contendo apenas seu fragmento da tupla). A forma de usar o esquema neste modelo é fazer com que o cliente cifre cada um dos fragmentos com a chave secreta compartilhada entre ele e o servidor que vai armazenar esse fragmento (caso contrário, um servidor falho teria acesso a todos os fragmentos e poderia recuperar a tupla). Sendo assim, cada servidor terá acesso apenas ao fragmento a ele endereçado.

Como lidamos com clientes maliciosos e precisamos manter uma equivalência entre o estado das réplicas, mesmo que um servidor não aceite um fragmento recebido (passo 1 do servidor no Algoritmo 9) ele deve manter os dados referentes a inserção da tupla (como o molde e a prova). Isto é feito para evitar que os estados (conjunto de tuplas presentes no espaço) de diferentes réplicas corretas sejam não equivalentes, o que pode facilmente ocorrer se algumas réplicas descartam uma tupla sendo inserida (devido ao recebimento de um fragmento inválido) enquanto outras a mantém no espaço. Esta não equivalência permitiria a ocorrência de resultados inesperados nas operações de leitura/remoção.

Além das modificações descritas acima, três outras modificações menores se fazem necessárias. Em primeiro lugar, a função *extract_response* usada no algoritmo do espaço de tuplas replicado (algoritmo 2), agora deve tentar extrair as respostas conforme definido nos passos 3 e 4 do algoritmo 10, e não mais esperar $f+1$ respostas iguais. A segunda modificação diz respeito a aplicação dessa mesma estratégia na leitura otimizada (procedimento *execute_rd* do algoritmo 2): a tupla só é lida se são recebidas $n-f$ mensagens contendo o mesmo t_h e $PROOF_t$, com pelo menos $f+1$ fragmentos válidos da tupla. A última modificação diz respeito ao envio de mensagens invalidando tuplas. Estas mensagens devem ser enviadas a todos os servidores através de difusão com ordem total a fim de garantir que não existam inconsistências enquanto uma tupla inválida esteja sendo removida do espaço replicado.

Apesar de acrescentar um custo adicional ao protocolo de confidencialidade apresentado na seção anterior, tal custo é bastante aceitável se considerarmos que: *(i.)* apenas criptografia simétrica é usada, e esta é muito rápida se comparada as demais operações criptográficas do esquema; *(ii.)* o número de servidores n não é esperado ser muito grande (uma boa estimativa seria $n \leq 10$), e portanto o número de cifragens no cliente não será grande; e *(iii.)* o servidor realiza apenas uma operação criptográfica

a mais para recuperar seu fragmento da requisição, portanto a maior parte do trabalho extra fica no cliente, o que permite ao sistema atender um (potencial) grande número de clientes.

Como última nota a respeito da integração de confidencialidade ao espaço de tuplas baseado em replicação ativa é preciso dizer que é impossível garantir linearização irrestrita quando existem clientes maliciosos inserindo tuplas no espaço. A razão para isso é que uma tupla inserida por um processo malicioso pode ser inválida para algumas réplicas e válida para outras, já que tal processo pode enviar fragmentos corretos apenas para algumas réplicas. Isto significa que uma tupla nestas condições pode ser lida se o processo leitor obtiver $f+1$ fragmentos válidos, que podem ou não ser retornados nas primeiras $n-f$ respostas recebidas do sistema. Conseqüentemente, tal tupla pode ser lida eventualmente no sistema. Desta forma, o espaço de tuplas baseado em replicação ativa com confidencialidade garante linearização apenas no acesso às tuplas inseridas corretamente.

5.6.2 BTS

O BTS (ver seção 4.3) é a mais simples construção para espaço de tuplas baseada em sistemas de quóruns bizantinos. Seus algoritmos para inserção e leitura de tuplas (operações *out* e *rdp*, respectivamente) são bastante simples e eficientes. A integração dos algoritmos 9 e 10 do esquema de confidencialidade neste espaço é direta, praticamente nenhuma modificação é necessária. Isto ocorre por três motivos básicos: *(i.)* o BTS requer $n \geq 3f+1$, logo $n-f > f+1$ e portanto sempre existirão $f+1$ servidores corretos para retornar fragmentos válidos de uma tupla corretamente inserida; *(ii.)* a inserção de uma tupla no BTS é feita através de um protocolo trivial onde o cliente envia uma mensagem com a tupla a ser inserida para todos os servidores do sistema, desta forma basta que ele envie os diferentes fragmentos (e demais informações associadas a tupla) aos diferentes servidores para inserir uma tupla com confidencialidade; e *(iii.)* a leitura de uma tupla no BTS se dá quando a mesma está presente em pelo menos $f+1$ servidores, desta forma, com o esquema de confidencialidade, mudamos o algoritmo de leitura para que o cliente considere $f+1$ tuplas com os mesmos *fingerprint* e $PROOF_t$ e que tenham fragmentos válidos.

5.6.3 LBTS

O esquema de confidencialidade descrito neste capítulo não pode ser integrado ao LBTS. Isto ocorre devido a fase de reescrita do protocolo *rdp* desta construção. O problema nesta fase é que ela exige que uma tupla seja lida e posteriormente reescrita nos servidores. O esquema de confidencialidade proposto na seção anterior exige que os fragmentos do segredo sejam gerados todos ao mesmo tempo (durante a distribuição), desta forma, a reescrita só seria possível se o leitor combinasse o segredo para então gerar novos fragmentos do mesmo e redistribuí-los aos servidores. Esta abordagem não é factível no esquema de compartilhamento de segredos utilizado pois permite que clientes maliciosos reescrevam tuplas corretamente inseridas de tal forma a torná-las inválidas.

Tendo em vista este problema, é deixado como trabalho futuro a investigação de uma estratégia que permita o uso do esquema proposto quando reescritas forem necessárias (ver seção 7.4).

5.7 Implementação e Ambiente de Execução

Os algoritmos apresentados na seção anterior foram implementados utilizando o arcabouço para prototipação, simulação e execução de algoritmos distribuídos NEKO [124].

Os canais confiáveis e autenticados do sistema são implementados através do uso de *sockets* TCP e chaves de sessão baseadas no algoritmo *HmacSHA-1*. As assinaturas utilizadas nos protocolos são implementadas usando os algoritmos *SHA-1* e *RSA* (com 1024 bits, usada no algoritmo de replicação máquina de estados) para resumos e criptografia assimétrica, respectivamente. Toda implementação se baseia no uso da biblioteca de criptografia do Java 1.5 (*Java Cryptography Extensions*), com exceção do esquema de compartilhamento de segredo verificável publicamente, que foi completamente implementado utilizando a biblioteca de aritmética modular disponível no Java (especialmente a classe `BigInteger`) seguindo as especificações do trabalho original [116].

O ambiente de testes consta de 5 máquinas com a mesma configuração de *hardware* (AMD Athlon XP 1.9Ghz, 512MB de RAM, placa *ethernet* de 100MB/s) conectadas por um *switch* 1GB/s. O ambiente de *software* em todas as máquinas é também homogêneo: S.O. GNU/Linux *kernel* 2.6.12, máquina virtual Java da SUN versão 1.5.0_06.

5.8 Resultados e Análises

A fim de avaliar o esquema de confidencialidade proposto, realizamos alguns experimentos com a sua aplicação na replicação Máquina de Estados baseada no algoritmo PAXOS BIZANTINO. Os resultados destes experimentos são apresentados nesta seção. Todos os valores reportados aqui compreendem o tempo médio necessário (em milisegundos) para a execução de uma operação por um cliente do sistema, recolhido a partir de 500 execuções da operação e excluindo-se 5% dos valores com maior desvio da média. O tamanho da tupla utilizada nos testes é de 50 bytes, divididos em 4 campos, todos comparáveis. Os valores em tempo referidos no texto a seguir são aproximados.

A tabela 5.1 apresenta os custos da adição de confidencialidade no espaço de tuplas. Estes custos são apresentados considerando o espaço de tuplas replicado em 4, 7 e 10 servidores. Analisando esta tabela, podemos perceber que a maior parte do tempo despendido com o mecanismo de confidencialidade está relacionado com o cliente. Apena a operação de inserção (*out*) exige algum aumento de processamento no servidor. Este é um fato importante, pois indica a capacidade de escalar do sistema.

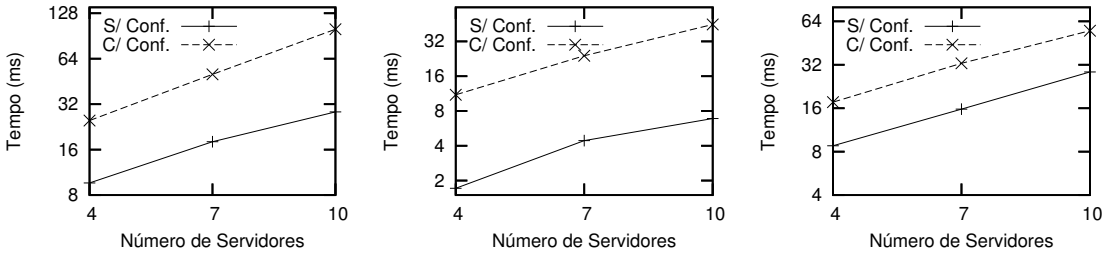
Nº de servidores	4			7			10		
Operação	*out*	*rdp*	*inp*	*out*	*rdp*	*inp*	*out*	*rdp*	*inp*
Custo no cliente	4,5	6,55	6,15	6,06	8,6	7,55	8,48	11,41	10,95
Custo no servidor	8,3	0	0	14,15	0	0	20,13	0	0
Aumentos no custo	12,8	6,55	6,15	20,21	8,6	7,55	28,61	11,41	10,95

Tabela 5.1: Custo do mecanismo de confidencialidade (valores em ms).

Os custos nas operações de inserção estão relacionados com o tempo gasto para gerar e assinar os fragmentos (cliente) e o tempo gasto para extrair e verificar o fragmento recebido, bem como construir

a prova de que o fragmento é correto (servidor). Nas operações de leitura e remoção, apenas os clientes têm aumento no custo de processamento, o qual está relacionado com o tempo necessário para verificar $f + 1$ fragmentos recebidos dos servidores e combinar estes fragmentos. Como esperado, com o aumento do número de servidores, os custos aumentaram, pois mais fragmentos precisam ser gerados, verificados e combinados.

A figura 5.1 apresenta a latência média para a execução das três operações com e sem confidencialidade, no protótipo de um espaço de tuplas tolerante a faltas bizantinas. A confidencialidade foi introduzida neste sistema da forma descrita na seção 5.6.1. Na figura 5.1(a), são apresentados os resultados referentes a operação de inserção. Como os testes foram realizados em 5 máquinas, nos cenários com 7 e 10 servidores, foi necessário executar mais de um processo na mesma máquina (o que aumenta a latência percebida pelo cliente). Caso os testes fossem realizados com um número maior de máquinas, certamente estes valores seriam menores. Como a operação de inserção é a que requer uma maior latência adicional e é a única que causa um aumento no processamento do servidor, os valores de latência apresentados, nos cenários com confidencialidade, cresceram mais do que o esperado com o aumento no número de servidores.



(a) OUT (b) RDP (c) INP

Figura 5.1: Desempenho na replicação máquina de estados.

Na figura 5.1(b) e 5.1(c), temos os resultados referentes as operações de leitura e remoção, respectivamente. Nestes cenários, o crescimento da latência foi linear e manteve um padrão, visto que nestas operações apenas o cliente tem um aumento no processamento.

5.9 Trabalhos Relacionados

Vários trabalhos têm proposto a integração de mecanismos de segurança ao modelo de coordenação por espaços de tuplas visando tanto controle de acessos em nível de espaço (quais processos podem executar operações sobre um espaço de tuplas) quanto em nível de tuplas (quais processos podem ler ou remover uma determinada tupla) [26, 44, 127]. A principal limitação destes trabalhos é assumir um espaço de tuplas confiável, i.e. que não vai revelar informações para partes não autorizadas. Quando consideramos um sistema tolerante a intrusões este problema se torna muito mais complexo devido a possibilidade de alguns dos servidores (que implementam o espaço replicado) serem controlados pelo adversário. Até onde sabemos este capítulo apresenta o primeiro trabalho a considerar confidencialidade em espaços de tuplas replicados.

Tomando em conta toda a literatura relacionada à manutenção de confidencialidade em sistemas de armazenamento tolerantes a faltas bizantinas, o esquema apresentado neste trabalho guarda alguma semelhança com alguns trabalhos. O trabalho seminal nessa área apresenta uma solução onde os dados são fragmentados e distribuídos por diversos servidores [57]. Mais tarde, uma série de outros trabalhos melhoraram este esquema tornando-o mais eficiente (ex. [48]). Em particular, o SECURE STORE usa um esquema de compartilhamento de segredo para dividir os dados entre servidores [75]. Para reconstruir os dados originais, a colaboração de pelo menos $f + 1$ servidores é requerida. O esquema não é muito eficiente e requer um grande número de servidores devido ao fato dos fragmentos criados serem replicados em diversos sítios. Além disso, o sistema não tolera clientes maliciosos. O sistema CODEX [90] utiliza protocolos baseados em sistemas de quóruns e esquemas de compartilhamento de segredo para prover armazenamento de segredos. Os segredos armazenados no CODEX são distribuídos pelos servidores (cada servidor recebe um fragmento) e não podem ser apagados ou sobrescritos. Cada um destes segredos é periodicamente regerado, de tal forma a fazer com que a confidencialidade dos segredos seja sempre mantida[5]. Um trabalho mais recente nessa área é apresentado em [34]. Este trabalho apresenta uma construção para armazenamento tolerante a faltas bizantinas que provê confidencialidade e otimização de espaço através do uso de um esquema de criptografia de limiar bastante custoso, executado nos servidores através de um protocolo de difusão confiável (onde os servidores trocam mensagens entre si). A otimização de espaço é obtida através do uso de códigos de apagamento [107]. Diferentemente destes trabalhos, o mecanismo proposto neste capítulo é baseado na eficiente primitiva de compartilhamento de segredo verificável publicamente proposta em [116], não requer nenhum tipo de comunicação entre os servidores e coloca a maior parte do processamento criptográfico do lado cliente, o que garante sua boa escalabilidade. Como o espaço de tuplas é muito mais um mecanismo de coordenação do que de armazenamento, as tuplas nele armazenada são usualmente pequenas e não tendem a ficar armazenadas por longos períodos, nosso mecanismo de confidencialidade não leva em consideração a otimização do espaço de armazenamento e nem o rejuvenescimento de fragmentos.

5.10 Conclusões

Este capítulo apresentou uma solução para manutenção de confidencialidade em espaços de tuplas tolerantes a faltas bizantinas. Esta solução se baseia no uso de um eficiente esquema de compartilhamento de segredo verificável publicamente, juntamente com resumos criptográficos e criptografia simétrica.

O esquema proposto foi implementado e alguns experimentos foram realizados, demonstrando que, apesar do esquema adicionar um custo adicional no tempo necessário para a execução de uma operação no espaço de tuplas replicado, este custo é aceitável se comparando a alta complexidade da qualidade de serviço sendo oferecida.

Com este esquema, a arquitetura para segurança de funcionamento em espaços de tuplas defi-

[5] Adversários não teriam tempo para atacar mais réplicas e obter fragmentos suficientes para reconstruir o segredo. Para mais detalhes veja [119].

nida no capítulo 3 está completa. No próximo capítulo será investigado o poder deste objeto na sincronização de processos sujeitos a faltas bizantinas.

Capítulo 6

Compartilhando Memória entre Processos Bizantinos usando Espaços de Tuplas

Este capítulo analisa as conseqüências de se usar um espaço de tuplas com segurança de funcionamento para coordenação de processos sujeitos a falhas bizantinas.

6.1 Contexto e Motivação

Apesar do grande número de algoritmos tolerante a faltas bizantinas para sistemas onde a comunicação se dá através de passagem de mensagens [33, 36, 42, 43, 73, 79, 88], apenas recentemente iniciou-se o estudo de algoritmos para a coordenação de processos sujeitos a falhas bizantinas que se comunicam através de memória compartilhada [4, 8, 85]. A motivação desta linha de pesquisa reside na disponibilidade atual de várias soluções para a emulação de objetos de memória compartilhada com segurança de funcionamento em sistemas distribuídos baseados em passagem de mensagens sujeitos a faltas bizantinas [21, 33, 36, 42, 73, 88]. Uma questão fundamental nesta linha de pesquisa é: qual o poder dos objetos de memória compartilhada na coordenação de processos que podem falhar arbitrariamente [85]? Esta questão é especialmente relevante já que estes tipos de falhas podem ser usadas para modelar o comportamento de adversários maliciosos como *hackers* e *malware* [10]. Resumindo, o objetivo é mascarar estas falhas usando objetos de memória compartilhada.

Os primeiros trabalhos nesta área resultaram em várias contribuições teóricas importantes. Eles mostraram que objetos simples como registradores e *sticky* bits[1] [105], quando combinados com listas de controle de acesso (ACLs), são suficiente para implementar consenso [4], que a resistência ótima

[1] Informalmente, um *sticky* bit b é um registrador que pode armazenar os valores 0, 1 ou $\perp$ (valor inicial, representando um valor "não definido") e suporta duas operações básicas: $b.\mathsf{write}(v)$ escreve um valor $v \in \{0,1\}$ no *sticky* bit se seu valor atual é $\perp$ (retornando *true*), ou não faz nada se o valor de b é diferente de $\perp$ (retornando *false*); e $b.\mathsf{read}()$, que retorna o valor armazenado em b. Em [105] é provado que um *sticky* bit pode ser usado para implementar consenso livre de espera entre um número arbitrário de processos.

do consenso neste modelo é $m \geq 3t + 1$ [4, 85] (t é a quantidade máxima de processos falhos e m é o número total de processos[2]), e que os *sticky* bits protegidos por ACLs são objetos universais, i.e., eles podem ser usados para implementar qualquer objeto de memória compartilhada [85], para citar apenas algumas destas contribuições.

Apesar da inegável importância destes resultados teóricos, do ponto de vista prático, estes trabalhos também demonstram a limitação da combinação de objetos simples como *sticky* bits com ACLs: o número de objetos requeridos e a quantidade de operações executadas nestes objetos é enorme, tornando os algoritmos baseados nesta combinação impráticaveis em sistemas reais. A razão para isto é que os algoritmos caem em um problema combinatorial: existem m processos e k objetos de memória compartilhada para os quais devemos definir ACLs associando processos aos objetos de tal forma que processos falhos não possam invalidar as ações de processos corretos.

Este capítulo contribui com essa área através da modificação deste modelo em dois aspectos. Em primeiro lugar, é proposto o uso de **políticas de segurança de granularidade fina** [96] para controlar o acesso aos objetos de memória compartilhada. Estas políticas nos permitem especificar se uma operação invocada em um objeto de memória compartilhada deve ser executada ou negada, dependendo de quem invoca a operação, dos parâmetros passados e do estado atual do objeto. Um objeto protegido por este tipo de política é chamado PEO (*Policy-Enforced Object*).

Em segundo lugar, os algoritmos apresentados neste capítulo usam apenas um objeto de memória compartilhada: o **espaço de tuplas aumentado** (*augmented tuple space*) [11, 117]. Este objeto, uma extensão do espaço de tuplas introduzido na linguagem LINDA [59], armazena estruturas de dados genéricas chamadas tuplas e provê operações para inclusão, remoção, leitura e inclusão condicional destas tuplas. No capítulo 3 apresentamos uma arquitetura que implementa este tipo de objeto.

Este capítulo mostra que os PEATS (***Policy-Enforced Augmented Tuple Spaces***) são uma solução atrativa para a coordenação de processos sujeitos a falhas bizantinas. Os algoritmos apresentados são muito mais simples que os algoritmos previamente apresentados na literatura (baseados em *sticky* bits e ACLs) [4, 85]. Eles são também mais eficientes em termos do número de bits e objetos necessários para resolução de certos problemas. Esta comparação de objetos aparentemente simples como *sticky* bits com objetos aparentemente complexos como os espaços de tuplas pode parecer injusta, porém, na realidade, a implementação de versões linearizáveis destes objetos requer protocolos com complexidades similares. Por exemplo, ambos podem ser implementados usando os sistemas tolerantes a faltas bizantinas baseados em replicação máquina de estados [33, 36, 42, 73].

As contribuições apresentadas neste capítulo são as seguintes:

- um novo modelo de computação onde objetos de memória compartilhada são protegidos por políticas de granularidade fina;
- um novo objeto de memória compartilhada, o espaço de tuplas aumentado protegido por políticas (PEATS);

[2]Não usamos n e f como nos capítulos anteriores devido ao fato de não estarmos mais tratando com os servidores que implementam o espaço de tuplas (n servidores, dos quais até f podem falhar arbitrariamente), mas sim com os m processos clientes (dos quais, até t podem falhar de forma arbitrária) que usam o espaço de tuplas para se coordenar.

- mostramos os benefícios deste modelo através da provisão de algoritmos de consenso simples e eficiente com resistência $m \geq 3t+1$, e provamos que esta é a resistência ótima para algoritmos de consenso binários em nosso modelo de sistema. Além disso, nosso consenso forte binário usa apenas $O((m+t)\log m)$ bits contra $(m+1)\binom{2t+1}{t}$ *sticky* bits requeridos pelo algoritmo apresentado em [4];

- provamos que os PEATS são universais [67], i.e. eles podem ser usados para implementar qualquer outro objeto de memória compartilhada, provendo duas construções universais (ver seção 6.5) baseadas em objetos deste tipo; uma destas corresponde a uma melhoria na construção t-resistente de [85] que reduz o número de bits de memória compartilhada requeridos de $O(i\,m(\log m)\binom{2t+1}{t})$ para $O(i(m+t)(\log m)^2))$, onde i é o número de operações executadas no objeto emulado.

6.2 Modelo de Sistema

Esta seção apresenta o modelo de sistema considerado neste capítulo[3].

Assumimos um conjunto de m processos $\mathscr{P} = \{p_1, p_2, ..., p_m\}$ que se comunicam através de um conjunto de k objetos de memória compartilhada $\mathscr{O} = \{o_1, ..., o_k\}$ (ex., registradores, *sticky* bits, espaços de tuplas). Cada processo pode ser **correto** ou **falho**. Um processo correto é limitado a obedecer sua especificação, enquanto um processo falho, também chamado processo **bizantino** [79], pode desviar de forma arbitrária de sua especificação. Assumimos que um processo malicioso não pode se passar por outro processo correto quando invocando uma operação num objeto de memória compartilhada. Esta limitação é importante em nosso modelo já que usamos um **monitor de referência** [6] para verificar políticas de acesso para objetos. O monitor precisa saber a identidade correta do processo invocando operações no objeto para conceder ou negar acesso para esta invocação.

Uma **configuração** de um sistema distribuído onde m processos se comunicam usando k objetos de memória compartilhada é um vetor $C = \langle q_1, ..., q_m, r_1, ..., r_k \rangle$ onde q_i é o estado do processo p_i e r_j é o estado do objeto o_j. Um **passo** de um processo é uma ação deste processo que altera a configuração (o estado de um processo e/ou objeto). Uma **execução** de um sistema distribuído é uma seqüência infinita $C_0, a_0, C_1, a_1, ...$ onde C_0 é uma configuração inicial e cada a_i é um passo que muda o sistema da configuração C_i para C_{i+1}.

Cada objeto de memória compartilhada é acessado através de um conjunto de operações disponibilizadas através de sua interface. Uma operação de um objeto executada por um processo tem início quando o processo faz uma **invocação** a esta operação. A operação termina quando o processo recebe uma **resposta** para a invocação correspondente. Uma operação que foi invocada mas que não recebeu sua resposta é chamada **operação pendente**. Assumimos que todos os processos (mesmo os falhos) invocam uma operação em um objeto de memória compartilhada apenas após receber a resposta de

[3]Definimos um novo modelo neste capítulo a fim de apresentar as contribuições de forma semelhante à literatura de memória compartilhada por processos bizantinos [4, 85]. No entanto, o modelo de sistema aqui descrito é equivalente ao definido no capítulo 2.

sua última invocação a este objeto. Esta condição é algumas vezes chamado **boa formação** [83] ou **interação correta** [9].

Assumimos que os objetos de memória compartilhada usados neste capítulo são confiáveis (não falham) e satisfazem a condição de **linearização** [69]: apesar de serem acessados concorrentemente, cada operação executada parece tomar efeito instantaneamente em algum ponto entre sua invocação e resposta, de uma forma tal que eles parecem ter sido acessados seqüencialmente.

Em termos de condições de vivacidade, assumimos que todas as operações providas pelos objetos de memória compartilhada usados neste capítulo satisfazem alguma das condições de terminação descritas a seguir (x é um objeto de memória compartilhada):

- **liberdade de bloqueio (*lock-freedom*)** [9]: uma operação x.op é livre de bloqueio se, quando invocada por um processo correto em qualquer ponto em uma execução em que existam operações pendentes invocadas por processos corretos, alguma operação (x.op ou outra operação pendente) será completada;
- **t-resistência (*t-resilience*)** [85]: uma operação x.op é t-resistente se, quando executada por um processo correto, ela termina por completar em qualquer execução em que pelo menos $m-t$ processos corretos permanecem tendo invocações pendentes para alguma operação de x;
- **t-limiar (*t-threshold*)** [85]: uma operação x.op é t-limiar se, quando executada por um processo correto, ela termina por completar em qualquer execução em que pelo menos $m-t$ processos corretos invocam x.op;
- **liberdade de espera (*wait-freedom*)** [67]: uma operação x.op é livre de espera se, quando executada por um processo correto, ela termina por completar em qualquer execução (independentemente da ocorrência de falhas em outros processos).

A principal diferença entre t-limiar e t-resistência, reside no fato de que, na primeira, uma operação completa se $m-t$ processos corretos invocam a **mesma** operação, enquanto na segunda, a operação é completada apenas se $m-t$ processos corretos permanecem invocando alguma operação do objeto. Note que t-limiar implica em t-resistência, porém, o contrário não se verifica.

Para qualquer uma destas condições de vivacidade, dizemos que um objeto satisfaz a condição se todas as suas operações satisfazem a condição.

6.3 Objetos Protegidos por Políticas

Os trabalhos anteriores sobre objetos compartilhados por processos sujeitos a faltas bizantinas consideram que as operações providas por estes objetos são protegidas por ACLs [4, 8, 85]. Neste modelo, cada operação provida por um objeto é associada a uma lista de processos que têm acesso a esta operação. Apenas processos que têm acesso a uma operação podem invocá-la. Note que

este modelo requer algum tipo de monitor de referência [6] para proteger o objeto de acessos não autorizados. A implementação deste monitor não é problemática já que, em geral, é assumido que os objetos de memória compartilhada são implementados por servidores replicados com capacidade de processamento [33, 36, 42, 73, 88].

Neste capítulo assumimos este tipo de implementação e estendemos a noção de proteção considerando políticas de segurança mais poderosas que o controle de acesso baseado em ACLs. Para isso, definimos um **objeto protegido por políticas** (*Policy-Enforced Object* - PEO) como um objeto cujo acesso é controlado por políticas de segurança de granularidade fina. Mais a frente, mostraremos que o uso desse tipo de política torna possível a implementação de algoritmos simples e eficientes para a resolução de importantes problemas em sistemas distribuídos, como por exemplo, consenso.

Um monitor de referência permite a execução de uma operação em um PEO se a invocação correspondente satisfaz a política de acesso do objeto. A **política de acesso** é composta por um conjunto de regras, sendo cada uma delas composta por um padrão de invocação de operação e uma expressão lógica. A execução de uma invocação de operação é permitida (denotada pelo predicado $execute(\text{op})$ com valor $true$) apenas se a expressão lógica associada é satisfeita pelo padrão de invocação. Seguindo o princípio de falhar de forma segura por padrão, qualquer invocação que não satisfaz a regra é sempre negada [113]. Um valor lógico *false* é retornado pela operação sempre que o acesso é negado.

O monitor de referência tem acesso a três tipos de informações para avaliar se uma invocação $invoke(p, \text{op})$ a um objeto protegido x pode ser executada:

- a identidade do processo invocador p;
- a operação invocada op e seus argumentos;
- o estado atual de x.

Um exemplo de PEO é o registrador atômico protegido por política r em que apenas os processos p_1, p_2 e p_3 podem escrever e os valores escritos devem sempre ser maiores que o valor atual de r. A política de acesso para esse PEO é apresentada na figura 6.1. Usamos o símbolo ' $:-$ ' oriundo da linguagem de programação PROLOG para denotar que o predicado do lado esquerdo da regra é verdadeiro se a condição da direita é verdadeira. O predicado *execute* (na esquerda) indica se a operação deve ou não ser executada, enquanto o predicado *invoke* (na direita) indica se a operação foi invocada.

Object State r
R_{read}: $execute(read()) \ :- \ invoke(p, read())$
R_{write}: $execute(write(v)) \ :- \ invoke(p, write(v)) \wedge p \in \{p_1, p_2, p_3\} \wedge v > r$

Figura 6.1: Um exemplo de política de acesso para um registrador atômico.

Na política de acesso da figura 6.1, inicialmente são definidos os elementos do estado do objeto que podem ser utilizados nas regras. Neste caso, o estado do registrador é representado pelo seu valor

atual, denotado por r. A seguir são apresentadas uma ou mais regras de acesso. A primeira regra, chamada R_{read}, define que todas as leituras do registrador são permitidas. A segunda regra, chamada R_{write}, define que uma operação $write(v)$ invocada pelo processo p, só pode ser executada se *(i.)* p é um dos processos do conjunto $\{p_1, p_2, p_3\}$ e *(ii.)* o valor v a ser escrito for maior que o valor atual r do registrador. Note que a condição *(i.)* não é nada mais que uma implementação direta de uma ACL em nosso modelo.

Todos os algoritmos apresentados neste capítulo fazem uso de um único **espaço de tuplas aumentado protegido por políticas** *ts* (*Policy-Enforced Augmented Tuple Space* - PEATS), portanto, $\mathscr{O} = \{ts\}$. Uma implementação livre de espera e linearizável deste tipo de objeto foi apresentada no capítulo 3. Vale lembrar que toda implementação tolerante a faltas do PEATS requer resolução de consenso. Isto advêm diretamente do fato do número de consenso deste objeto ser maior que 1. Se fosse possível implementar este objeto sem resolver consenso, por exemplo, usando sistemas de quóruns "puros", a impossibilidade FLP [56] não seria válida: um sistema distribuído que pode resolver consenso em sistemas assíncronos (a implementação tolerante a faltas do PEATS) seria construído sem resolver consenso, o que é completamente contraditório.

Uma outra observação se faz necessária aqui: as construções de espaço de tuplas baseadas na junção de algoritmos para sistemas de quóruns bizantinos e consenso apresentadas no capítulo 4 (BTS e LBTS) **não** podem ser utilizadas para implementar diretamente um PEATS. O problema fundamental é que estas construções admitem a execução de operações no espaço de tuplas mesmo quando os estados das réplicas divergem. Mais precisamente, este tipo de política só pode ser implementado se as réplicas do PEATS estiverem no mesmo estado durante a execução de cada operação (o que é garantido pela replicação Máquina de Estados), garantindo assim que os monitores de referência das réplicas corretas sempre aceitam/negam a execução de uma invocação de forma unânime. Isto implica que, apesar de toda implementação tolerante a faltas do PEATS requerer a resolução de consenso, nem toda implementação de espaço de tuplas (que necessariamente faz uso de consenso – ver capítulo 4), pode implementar o modelo de proteção requerido pelo PEATS. Além deste problema, tanto o BTS quanto o LBTS não suportam a operação *cas*.

6.4 Resolvendo Consenso

Nesta seção ilustramos os benefícios do uso do PEATS para resolver várias variantes do problema de consenso.

O problema de consenso diz respeito a um conjunto de processos propondo valores de um conjunto $\mathscr{V}$ de possíveis valores (domínio) e tentando chegar a um acordo sobre um único valor decidido. Um **objeto de consenso** é um objeto de memória compartilhada que encapsula um algoritmo de consenso. Este objeto fornece apenas uma operação *propose*, que recebe como argumento um valor proposto e retorna o valor decidido pelo algoritmo encapsulado no objeto. A seguir, apresentamos algoritmos para implementação de três objetos de consenso (ou, para resolver três variantes do problema):

- **Consenso Fraco** [85]: Um objeto de consenso fraco x é um objeto de memória compartilhada com uma única operação x.propose(v),onde $v \in \mathscr{V}$, satisfazendo duas propriedades: (**Acordo**) em qualquer execução, a operação x.propose retorna o mesmo valor, chamado **valor de consenso**, para qualquer invocação feita por um processo correto; (**Validade**) em qualquer execução finita em que todos os processos participantes são corretos, se o valor de consenso é v, então algum processo invocou x.propose(v).

- **Consensus Forte** [85]: Um objeto de consenso forte x é definido por uma propriedade de validade mais forte que a usada no consenso fraco: (**Validade Forte**) se o valor de consenso é v, então algum processo correto invocou x.propose(v).

Note que, no modelo de sistema de faltas de parada, consenso fraco e forte são equivalentes.

Outra variante do consenso que resolvemos neste capítulo é o consenso (multivalorado) padrão [43], uma versão mais fraca do consenso forte:

- **Consenso Padrão** [43]: Um objeto de consenso padrão x é definido por uma condição de validade mais fraca que a validade forte: (**Validade Forte Padrão**) o valor de consenso tem de satisfazer duas condições: *(i.)* se todos os processos corretos invocam x.propose(v), então v é o valor de consenso; *(ii.)* se o valor de consenso é v, então algum processo correto invocou x.propose(v) ou $v = \perp$.

A idéia por trás do consenso padrão é que o valor de consenso precisa ser o valor proposto por algum processo correto ou então um valor padrão $\perp \notin \mathscr{V}$ [43]. Esta idéia é de alguma forma relacionada com o problema do **consenso abortável** [45]. Neste problema, um processo pode decidir um valor "aborte" (*Q - quit*) quando alguma falha é detectada. No problema do consenso padrão, o valor padrão ($\perp$) pode ser decidido mesmo em execuções sem processos falhos, desde que nem todos os processos corretos proponham um mesmo valor.

Lembramos que todos estes objetos, como todos os objetos mencionados neste capítulo, precisam satisfazer alguma das condições de terminação definidas na seção 6.2.

6.4.1 Objeto de Consenso Fraco

Em um objeto de consenso fraco, o valor de consenso pode ser qualquer um dos valores propostos. Com essa condição de validade, é perfeitamente legal que um valor proposto por um processo falho se torne o valor de consenso.

O algoritmo 11 apresenta a implementação do objeto de consenso fraco usando PEATS. O algoritmo é muito simples: um processo tenta inserir sua proposta no PEATS invocando a operação *cas*. Ele é bem sucedido apenas se não existe uma tupla DECISION[4] no espaço. Se esta tupla já existir no espaço, ela deve conter o valor a ser decidido, que será retornado como resultado do consenso.

[4]Uma tupla cujo primeiro campo contém o rótulo DECISION. Neste capítulo, dizemos que uma tupla é uma tupla X, se o valor de seu primeiro campo é o rótulo X.

Algoritmo 11 Objeto de consenso fraco (processo p_i).

Variáveis compartilhadas:

1: $ts = \varnothing$ {PEATS}

procedure x.propose(v)

2: **if** $ts.cas(\langle \text{DECISION}, ?d \rangle, \langle \text{DECISION}, v \rangle)$ **then**

3: $d \leftarrow v$ {o valor inserido é o valor de decisão}

4: **end if**

5: **return** d

A política de acesso para o PEATS usado no algoritmo 11 é apresentada na figura 6.2. Nesta política, usamos o predicado $formal(x)$ para verificar se um campo x da tupla é formal. Esta política de acesso permite apenas execuções da operação *cas*. O molde presente no espaço precisa ter dois campos: o primeiro com a constante DECISION e o segundo com um campo formal. A entrada (segundo argumento da operação *cas*), precisa ser uma tupla DECISION, semelhante ao molde mas com o segundo campo definido.

Object State TS

R_{cas}: $execute(cas(\langle \text{DECISION}, x \rangle, \langle \text{DECISION}, y \rangle)) \ :-$

$invoke(p, cas(\langle \text{DECISION}, x \rangle, \langle \text{DECISION}, y \rangle)) \wedge formal(x)$

Figura 6.2: Política de acesso para o PEATS usado no algoritmo 11.

Além de sua simplicidade e elegância, este algoritmo apresenta várias propriedades interessantes: em primeiro lugar ele é **uniforme** [9], i.e., funciona para qualquer quantidade de processos e os processos não precisam conhecer uns aos outros para participar do consenso. Segundo, ele pode resolver **consenso multivalorado**, já que o domínio dos valores propostos é arbitrário. Finalmente, o algoritmo é **livre de espera**, i.e., ele sempre termina, mesmo com a ocorrência de falhas em outros processos que participam do consenso.

Teorema 2 *O algoritmo 11 implementa um objeto de consenso fraco livre de espera.*

Prova: A partir da política de acesso, é possível ver que a única maneira de inserir uma tupla no espaço é invocando a operação *cas*. Esta operação só pode ser bem sucedida em sua execução (retornando *true*) uma única vez pois a política não permite a remoção de tuplas do espaço (operações *in* e *inp* não permitidas). Desta forma, a propriedade de acordo é satisfeita já que o primeiro processo que executa com sucesso a operação *cas* vai inserir uma tupla DECISION com o valor de consenso v no espaço. Outros processos lerão v (através do campo formal $?d$) em suas execuções da operação *cas*, que retornarão sempre *false* (a tupla DECISION não será inserida no espaço). A propriedade de validade é satisfeita porque, em qualquer execução com apenas processos corretos, o valor de consenso só pode ter sido proposto por algum destes processos (que inseriu a tupla DECISION com sua proposta no espaço). O algoritmo é livre de espera porque a operação *cas* é livre de espera. ■

6.4.2 Objeto de Consenso Forte

Um objeto de consenso forte reforça a condição de validade requerendo que o valor de consenso seja proposto por processos corretos mesmo na presença de processos falhos. Esta condição mais restrita resulta num algoritmo mais complexo (porém, ainda simples se comparado aos trabalhos anteriores [4, 85]). Entretanto, este algoritmo não compartilha de alguns benefícios do algoritmo apresentado na seção anterior:

- **não uniforme**: o algoritmo de consenso forte não é uniforme pois um processo participante precisa saber quais são os outros para então ler suas propostas e decidir um valor proposto por algum processo correto;
- **consenso binário**: nosso algoritmo resolve apenas consenso binário. Esta limitação é também fruto do fato de que um processo precisa saber se um valor foi proposto por um processo correto antes de decidí-lo;
- **objeto t-limiar**: o algoritmo para consenso forte não é livre de espera pois são requeridas $m-t$ propostas para garantidamente o algoritmo terminar. Entretanto, o número de processos necessários em nosso algoritmo é ótimo: $m \geq 3t+1$ (veja corolário 1 na próxima seção).

Algoritmo 12 Objeto de consenso forte (processo p_i).

Variáveis compartilhadas:

1: $ts = \varnothing$ {PEATS}

procedure x.propose(v)

2: $ts.out(\langle \text{PROPOSE}, p_i, v \rangle)$
3: $S_0 \leftarrow \varnothing$ {conjunto dos processos que propuseram 0}
4: $S_1 \leftarrow \varnothing$ {conjunto dos processos que propuseram 1}
5: **while** $|S_0| < t+1 \wedge |S_1| < t+1$ **do**
6: **for all** $p_j \in \mathscr{P} \setminus (S_0 \cup S_1)$ **do**
7: **if** $ts.rdp(\langle \text{PROPOSE}, p_j, ?v \rangle)$ **then**
8: $S_v \leftarrow S_v \cup \{p_j\}$ {p_j propôs v}
9: **end if**
10: **end for**
11: **end while**
12: **if** $ts.cas(\langle \text{DECISION}, ?d, * \rangle, \langle \text{DECISION}, v, S_v \rangle)$ **then**
13: $d \leftarrow v$ {o valor inserido é o valor de decisão}
14: **end if**
15: **return** d

O algoritmo 12 apresenta o protocolo para consenso binário forte. Este algoritmo funciona da seguinte maneira: inicialmente, um processo p_i insere sua proposta no espaço de tuplas aumentado ts através da inserção da tupla PROPOSE (linha 2). A seguir, p_i acessa continuamente ts tentando ler as propostas (linha 7) até que ele encontre algum valor proposto por pelo menos $t+1$ processos (laço das linhas 5-11). O raciocínio por trás da espera por $t+1$ propostas iguais é que pelo menos um processo correto propôs um valor que aparece nessa quantidade de propostas, já que no máximo

t processos podem falhar. O primeiro valor que satisfizer esta condição é então inserido no espaço de tupla usando *cas*. Está fase de confirmação é importante porque diferentes processos podem coletar $t+1$ propostas para diferentes valores e é preciso garantir que uma única decisão seja definida. Todas as invocações posteriores à operação *cas* retornaram este valor (linhas 12-14).

Object State TS
R_{rdp}: $execute(rdp(t)) \ :- \ invoke(p, rdp(t))$
R_{out}: $execute(out(\langle \text{PROPOSE}, p, x\rangle)) \ :-$
$\quad invoke(p, out(\langle \text{PROPOSE}, p, x\rangle)) \wedge \nexists y : \langle \text{PROPOSE}, p, y\rangle \in TS$
R_{cas}: $execute(cas(\langle \text{DECISION}, x, *\rangle, \langle \text{DECISION}, v, S_v\rangle)) \ :-$
$\quad invoke(p, cas(\langle \text{DECISION}, x, *\rangle, \langle \text{DECISION}, v, S_v\rangle)) \wedge$
$\quad formal(x) \wedge |S_v| \geq t+1 \wedge (\forall q \in S_v, \langle \text{PROPOSE}, q, v\rangle \in TS)$

Figura 6.3: Política de acesso para o PEATS usado no algoritmo 12.

A política de acesso para o PEATS usado no algoritmo 12 é apresentada na figura 6.3. Esta política especifica que qualquer processo pode ler qualquer tupla (regra R_{rdp}); que cada processo pode introduzir apenas uma tupla PROPOSE no espaço (regra R_{out}); que o segundo campo do molde usado na operação *cas* precisa ser um campo formal (regra R_{cas}); e que o valor de decisão v deve aparecer na propostas de pelo menos $t+1$ processos. Estas regras simples, que podem facilmente ser implementadas na prática, limitam efetivamente o poder de processos bizantinos, permitindo a existência de um algoritmo bastante simples para a resolução de consenso forte.

O algoritmo 12 requer apenas $m(\lceil \log m \rceil + 1) + (1 + (t+1)\lceil \log m \rceil)$ bits no objeto PEATS[5] (m tuplas PROPOSE mais uma tupla DECISION). O algoritmo de consenso com a mesma resistência apresentado em [4] requer $(m+1)\binom{2t+1}{t}$ *sticky* bits[6].

Teorema 3 *O algoritmo 12 implementa um objeto de consenso binário forte t-limiar.*

Prova: A partir da política de acesso podemos ver que a única forma de inserir uma tupla DECISION no espaço é invocando a operação *cas*. Esta operação pode ser invocada com sucesso apenas uma vez, já que uma tupla inserida não pode ser removida (operações *in* e *inp* não são permitidas pela política), e duas tuplas de decisão não podem ser inseridas (o segundo campo do molde usado na operação *cas* precisa ser formal). Em qualquer execução do algoritmo, o primeiro processo que executa *cas* após ter lido $t+1$ tuplas PROPOSE com o mesmo valor v insere uma tupla DECISION com v (se sua invocação satisfaz a regra R_{cas}), fazendo com este se torne o valor de decisão (linhas 13 e 15). A propriedade de acordo é sempre satisfeita já que o valor v associado à tupla DECISION no espaço será lido por todos os processos corretos que não forem bem sucedidos em suas invocações da operação *cas*, i.e., todos que receberem *false* como resultado. Seus valores de decisão será v (linhas 12 e 15).

O algoritmo satisfaz também validade forte já que a tupla DECISION só pode ser inserida se seu valor v for justificado por $t+1$ tuplas PROPOSE contendo este valor presentes no espaço (pelo menos uma foi inserida por um processo correto). Esta condição é reforçada peloa regra R_{cas} da política de acesso.

[5]Ex., apenas 68 bits são requeridos para $t = 4$ e $m = 13$.

[6]O algoritmo de [4] exige uma quantidade enorme de memória. Por exemplo, se queremos tolerar $t = 4$ processos falhos, temos de ter $m = 13$ processos e usamos 1764 *sticky* bits.

Em termos de condições de terminação, o algoritmo é um protocolo t-limiar. Esta propriedade é satisfeita devido ao fato de que um processo só decide um valor v se este valor tiver sido proposto por $t+1$ processos. Assumindo $m \geq 3t+1$, podemos mostrar facilmente que se $m-t$ processos corretos (pelo menos $2t+1$) invocam x.propose com algum valor $v' \in \{0,1\}$, sempre existirá pelo menos $t+1$ tuplas PROPOSE para algum valor (0 ou 1) e um processo irá inserir uma tupla de decisão justificada no espaço. Dado que a operação *cas* é livre de espera, o algoritmo sempre termina. ■

6.4.3 Objeto de Consenso Multivalorado Forte

Um algoritmo para consenso multivalorado forte pode ser obtido com pequenas modificações no algoritmo de consenso binário forte, seguindo as mesmas idéias de [4]. De fato, se considerarmos o problema do **consenso k-valorado**, em que existem k possíveis valores para os processos proporem[7], i.e., $|\mathscr{V}| = k$ (consenso binário é um consenso 2-valorado), podemos usar o mesmo algoritmo da seção anterior e adaptá-lo para coletar diferentes valores de propostas em S_v, com $v \in \mathscr{V}$ e $|\mathscr{V}| = k$. O algoritmo funciona exatamente da mesma forma que o algoritmo 12: um processo propõe seu valor e permanece lendo valores propostos por outros processos até que existe um valor proposto por pelo menos $t+1$ processos. Este valor é tomado como uma possível decisão.

Infelizmente, este algoritmo requer mais processos para resistir a t processos falhos, como mostrado pelo seguinte teorema:

Teorema 4 *O algoritmo implementa um objeto de consenso k-valorado forte t-limiar se $m \geq (k+1)t+1$.*

Prova: As provas de acordo e validade são semelhantes as provas do teorema 3. Suponha a pior execução possível para um sistema executando o algoritmo acima: cada um dos k possíveis valores é proposto por t processos e t processos falhos não propõem (eles param ou ficam silenciosos durante toda a execução). Para garantir a terminação do algoritmo, precisamos que um processo desempate as t proposições para cada valor. Conseqüentemente, precisamos de $m \geq kt+t+1 = (k+1)t+1$ processos. ■

Uma conseqüência direta deste teorema é que o número de processos necessários para resolução de consenso k-valorado forte na presença de faltas bizantinas usando o algoritmo descrito é sempre $m > k$. Este resultado elimina a possibilidade do uso deste algoritmo em aplicações onde cada processo propõe algum identificador de processo no consenso. Exemplos de aplicações deste tipo são exclusão mútua baseada em consenso [88] e eleição de líder [9].

O algoritmo para consenso multivalorado forte requer $O(m(\log m + \log|\mathscr{V}|))$ bits de memória compartilhada.

O teorema a seguir prova que $m \geq (k+1)t+1$ é o número mínimo de processos requeridos para resolver o problema de consenso k-valorado forte tolerando t faltas.

[7] Note que este problema é completamente diferente do conhecido problema **consenso com conjunto-k**, onde o valor de consenso dos processos podem ser diferentes, mas dentro de um conjunto com k valores [39].

Teorema 5 *O problema de consenso k-valorado forte em sistemas assíncronos onde os processos se comunicam através de PEOs e até t processos podem apresentar faltas bizantinas só pode ser resolvido se o número de processos for* $m \geq (k+1)t+1$.

Prova: O teorema 4 prova que existe um algoritmo com essa resistência. Resta-nos provar então que não existe um algoritmo que implemente consenso k-valorado forte com $m \leq (k+1)t$. Assuma que existe um algoritmo A que resolve este problema com essa quantidade de processos. Vamos apresentar uma execução em que A não termina.

Seja $\mathscr{V} = \{v_1, ..., v_k\}$ o domínio dos valores. Suponha uma execução α de A onde nenhum processo falho participa do algoritmo (t processos ficam em silêncio) e cada um dos k valores de $\mathscr{V}$ é proposto por no máximo t processos corretos.

Independentemente do(s) objeto(s) de memória compartilhada e da(s) política(s) de acesso(s) utilizada(s) em A, para satisfazer a propriedade de validade do consenso forte, um processo correto só pode considerar um valor para decisão se este valor for proposto por pelo menos $t+1$ processos. Se isto não fosse verdade, seria fácil montar uma execução em que, devido à ausência de sincronismo no sistema, um processo correto poderia considerar para decisão um valor proposto apenas por processos falhos, violando a condição de validade.

Desta forma, em α o sistema chegará a uma configuração onde todos os processos corretos terão lido no máximo t propostas para cada um dos k valores e não conseguirão ler mais propostas (elas não existem, já que todos os corretos propuseram e os falhos estão em silêncio). Conseqüentemente, nenhum valor será proposto por $t+1$ processos e o algoritmo não terminará. Isto significa que A não pode existir. ■

Dado este teorema, podemos definir a resistência ótima para o consenso binário forte.

Corolário 1 *A resistência ótima para o problema de consenso binário forte em sistemas assíncronos onde os processos se comunicam através de PEOs é* $t = \lfloor \frac{m-1}{3} \rfloor$ *de m processos.*

Prova: A prova de que existe um algoritmo com esta resistência é dada pela existência do algoritmo 12. A prova de que esta resistência é ótima advêm diretamente do teorema 5 se fizermos $k = 2$. ■

6.4.4 Consenso Multivalorado Padrão

Um objeto de consenso multivalorado padrão pode ser obtido fazendo-se algumas modificações simples no objeto de consenso binário forte do algoritmo 12. O objetivo aqui é mostrar que podemos resolver consenso multivalorado mais forte que o consenso fraco com resistência ótima, i.e, com $m \geq 3t+1$ processos para tolerar t faltas.

As modificações requeridas no algoritmo 12 são as seguintes:

1. deve existir um conjunto S_v para cada um dos valores v nas tuplas $\langle PROPOSE, *, v\rangle$ contidas no PEATS, ao invés dos conjuntos S_0 e S_1 (como no algoritmo para consenso forte, descrito na seção anterior);

2. após a leitura de $m-t$ valores propostos, um processo coloca o valor $\perp$ na tupla DECISION, caso o não exista um valor v proposto por pelo menos $t+1$ processos (caso contrário, v vai na tupla);

3. se o valor colocado na tupla DECISION é $\perp$, o terceiro campo da tupla DECISION deve ser o conjunto de todos os S_v construídos durante a leitura das $m-t$ tuplas.

O objeto de consenso com estas modificações é apresentado no algoritmo 13.

Algoritmo 13 Objeto de consenso multivalorado padrão (processo p_i).

Variáveis compartilhadas:

1: $ts = \varnothing$ {PEATS}

procedure x.propose(v)

2: $ts.out(\langle \text{PROPOSE}, p_i, v\rangle)$
3: $\forall v \in \mathscr{V}, S_v \leftarrow \varnothing$
4: **while** $(\forall S_v, |S_v| < t+1) \wedge (|\bigcup S_v| < m-t)$ **do**
5: **for all** $p_j \in \mathscr{P} \setminus (\bigcup_{v\in\mathscr{V}} S_v)$ **do**
6: **if** $ts.rdp(\langle \text{PROPOSE}, p_j, ?v\rangle)$ **then**
7: $S_v \leftarrow S_v \cup \{p_j\}$ {p_j propôs v}
8: **end if**
9: **end for**
10: **end while**
11: **if** $|S_v| = t+1$ **then**
12: $P \leftarrow S_v$ {existe um valor proposto por um processo correto}
13: **else**
14: $v \leftarrow \perp$
15: $P \leftarrow \bigcup_{v\in\mathscr{V}} S_v$ {o valor padrão deve ser decidido}
16: **end if**
17: **if** $ts.cas(\langle \text{DECISION}, ?d, *\rangle, \langle \text{DECISION}, v, P\rangle)$ **then**
18: $d \leftarrow v$ {o valor inserido é o valor de decisão}
19: **end if**
20: **return** d

O algoritmo 13 é bastante semelhante ao algoritmo para consenso binário a não ser pelas modificações já descritas, que são implementadas da seguinte forma: os k conjuntos de processos são inicializados no início do algoritmo (linha 3), existe uma condição extra de saída no laço para quando forem lidas $m-t$ propostas diferentes (linha 4) e o valor padrão pode ser proposto para decisão se o último valor lido no laço não foi proposto por $t+1$ processos (linhas 11-16).

Para evitar a possibilidade de que processos maliciosos forcem o valor de consenso para $\perp$, temos de garantir que o valor padrão é colocado no PEATS (usando *cas*) por um processo p apenas se esse processo lê $m-t$ tuplas PROPOSE e nenhum dos valores lidos foi proposto por pelo menos $t+1$ processos. Esta condição é formalizada na política de acesso da figura 6.4.

Object State TS
R_{rdp}: $execute(rdp(t)) \ :- \ invoke(p, rdp(t))$
R_{out}: $execute(out(\langle \text{PROPOSE}, p, x\rangle)) \ :-$
$\quad invoke(p, out(\langle \text{PROPOSE}, p, x\rangle)) \wedge x \neq \perp \wedge \nexists y : \langle \text{PROPOSE}, p, y\rangle \in TS$
R_{cas}: $execute(cas(\langle \text{DECISION}, x, *\rangle, \langle \text{DECISION}, v, P\rangle)) \ :-$
$\quad invoke(p, cas(\langle \text{DECISION}, x, *\rangle, \langle \text{DECISION}, v, P\rangle)) \wedge formal(x) \wedge$
$\quad ((v \neq \perp \wedge |P| \geq t+1 \wedge (\forall q \in P, \langle \text{PROPOSE}, q, v\rangle \in TS)) \vee$
$\quad (v = \perp \wedge |\bigcup_{S_v \in P} S_v| \geq m - t \wedge (\forall S_v \in P, |S_v| \leq t) \wedge$
$\quad (\forall S_v \in P, \forall q \in S_v, \langle \text{PROPOSE}, q, v\rangle \in TS)))$

Figura 6.4: Política de acesso para o PEATS usado no consenso multivalorado padrão.

Existem duas diferenças básicas entre esta política e aquela usada para o consenso binário forte (figura 6.3). Primeiramente, todos os valores propostos precisam ser diferentes de $\perp$ (regra R_{out}). Em segundo lugar, a regra R_{cas} define que se p quer inserir a tupla DECISION no PEATS com $v = \perp$, ele precisa mostrar que não encontrou nenhum valor proposto por pelo menos $t+1$ processos. Mais precisamente, a regra define que se o segundo argumento da operação *cas* executada por p (algoritmo 13, linha 17) tem $v = \perp$, então o terceiro argumento deve conter um conjunto de conjuntos S_v satisfazendo as seguintes condições: *(i.)* a união de todos os conjuntos S_v precisa conter pelo menos $m - t$ processos; *(ii.)* nenhum conjunto S_v deve ter mais de t processos; e *(iii.)* para todo processo q pertencente a um conjunto S_v, deve haver uma tupla $\langle \text{PROPOSE}, q, v\rangle$ no PEATS.

Teorema 6 *O algoritmo 13 implementa um consenso multivalorado padrão t-limiar se $m \geq 3t+1$.*

Prova: A propriedade de acordo é satisfeita devido a política de acesso da figura 6.4, que não permite a inserção de mais de uma tupla DECISION no espaço. As duas condições da validade forte padrão são também satisfeitas:

1. Se todos os processos corretos invocam x.propose(v) com o mesmo valor v, então um valor diferente v' pode ser proposto por no máximo t outros processos. Desta forma, dada as regras da política de acesso, a operação *cas* não poderá ser executada para inserir uma tupla DECISION com v'.

2. Para que um valor $v \neq \perp$ seja decidido, ele precisa ter sido proposto por $t+1$ processos (pelo menos um deles é correto). ■

6.5 Construções Universais

Um problema fundamental em computação distribuída usando memória compartilhada é saber se um objeto X pode ser usado para implementar (ou **simular**) outro objeto Y. Esta seção prova que o PEATS é um **objeto universal** [67], i.e., ele pode ser usado para emular qualquer outro objeto de memória compartilhada. Em [67] é mostrado que um objeto é universal em um sistema com m processos se e somente se ele tem **número de consenso** m, i.e., se ele pode ser usado para resolver consenso entre m processos [67].

A prova de que o PEATS é universal é feita através da provisão de duas construções universais baseadas neste tipo de objeto. Uma **construção universal** é um algoritmo que usa um ou mais objetos universais para emular qualquer outro objeto de memória compartilhada [67]. Existem várias construções universais livre de espera para o modelo por faltas de parada, usando objetos de consenso [67], *sticky* bits [105], registradores *compare and swap* [9] e vários outros objetos universais. Uma construção universal para o modelo de falhas bizantinas usando *sticky* bits foi definida em [85]. Entretanto, esta construção não é livre de espera, mas sim t-resistente, uma condição de terminação mais apropriada para algoritmos tolerantes a faltas bizantinas. A possibilidade da existência de algoritmos com propriedades de vivacidade mais fortes é ainda um problema em aberto [85].

Para definir uma construção universal que emula qualquer objeto determinista o de um certo tipo, temos de começar definindo o que é o tipo de um objeto. Um tipo T é definido pela tupla $\langle STATE_T, S_T, INVOKE_T, REPLY_T, apply_T \rangle$ onde $STATE_T$ é o conjunto de possíveis estados dos objetos do tipo T, $S_T \in STATE_T$ é um estado inicial para os objetos deste tipo, $INVOKE_T$ é o conjunto de possíveis invocações de operações providas por objetos do tipo T, $REPLY_T$ é o conjunto de possíveis respostas a essas invocações, e $apply_T$ é uma função definida como:

$$apply_T : STATE_T \times INVOKE_T \rightarrow STATE_T \times REPLY_T.$$

A função $apply_T$ representa a transição de estados do objeto. Dado um estado S_i e uma invocação op, $apply_T(S_i, op)$ resulta em um novo estado S_j (o resultado da execução da operação op no estado S_i) e em uma resposta *rep* para a invocação. Esta definição é suficiente para mostrar a universalidade dos espaços de tuplas, no entanto, Malkhi et al. mostrou que uma generalização (trivial) é necessária para representar a emulação de tipos não deterministas e alguns objetos que satisfazem garantias mais fracas de vivacidade [85].

Nesta seção mostramos uma construção universal não-bloqueante bastante simples, que mostra o poder e a simplicidade dos algoritmos que usam PEATS. Adicionalmente, apresentamos modificações na construção universal t-resistente de [85], melhorando sua eficiência em termos de complexidade de bits.

6.5.1 Um Construção Universal Não-bloqueante Simples

A construção não-bloqueante apresentada nesta seção segue a mesma linha de construções prévias da literatura [9, 67, 85]. A idéia é fazer com que todos os processos corretos percebam e executem a mesma seqüência de operações invocadas no objeto emulado. Cada processo p_i mantém uma réplica do estado do objeto emulado S_i. Uma operação op é executada aplicando-se a função $apply_T(S_i, op)$ neste estado. O problema então fica reduzido essencialmente a definir uma ordem total para a execução das operações.

As operações a serem executadas no objeto emulado podem ser invocadas em qualquer um dos processos, portanto a definição de uma ordem para ela requer consenso entre os processos. Assim, precisamos de um objeto com número de consenso m, i.e., um objeto universal.

Tendo este objeto universal, a solução é fazer com que as operações a serem executadas no objeto sejam adicionadas a uma lista onde cada elemento tem um número de seqüência. O elemento com maior número de seqüência representa a última operação executada no objeto emulado. A consistência da lista, i.e. a propriedade de que cada um de seus elementos (cada operação) é seguida por no máximo um elemento, é garantida pelo objeto universal, o PEATS neste caso. Dada essa lista, cada processo executa as operações do objeto emulado em ordem crescente do número de seqüência.

Conforme já discutido, a lista de operações é implementada usando o PEATS. A idéia chave é representar cada operação como uma tupla SEQ com um campo posição e inserir essas tuplas no espaço usando a operação *cas*. Quando um processo quer executar uma operação *op* ele invoca *cas* da seguinte forma: se não existe uma tupla SEQ com o maior número de seqüência conhecido pelo processo no espaço, então ele insere uma tupla SEQ com esse número de seqüência e sua invocação à *op*. A figura 6.5 ilustra esta idéia.

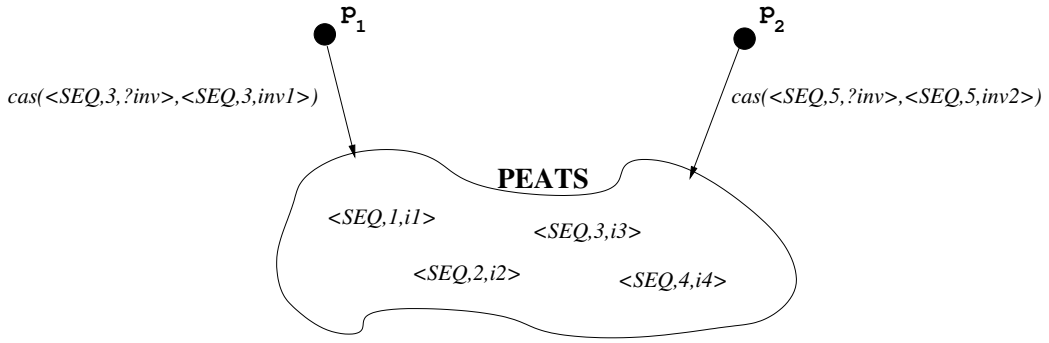


Figura 6.5: Construção universal usando PEATS.

Nesta figura, o processo p_1 tenta inserir uma tupla contendo a invocação com número de seqüência 3 no PEATS, enquanto o processo p_2 executa *cas* na esperança de inserir uma operação com número de seqüência 5. Dado que no PEATS já existem tuplas com números de seqüência de 1 a 4, o processo p_1 não conseguirá inserir sua tupla, enquanto o processo p_2 terá sucesso em sua inserção. O algoritmo 14 apresenta este objeto universal.

O algoritmo assume que cada processo p_i começa sua execução com o estado inicial do objeto emulado ($state = S_T$, linha 2) e conhecendo uma lista de operações vazia no PEATS ($pos = 0$, linha 3). Quando um processo p_i invoca uma operação *inv* no objeto emulado, ele itera através da lista de operações no PEATS atualizando sua variável $state$ (laço das linhas 4-11) e tentando inserir sua operação no final da lista usando a operação *cas* (linha 6). Se a operação *cas* é bem sucedida por p_i, a variável *state* de p_i é atualizada e a resposta de sua invocação é retornada (linhas 7 e 8).

O algoritmo é não-bloqueante devido a operação *cas*: quando dois processos tentam concorrentemente colocar uma tupla no final da lista, pelo menos um deles consegue. Entretanto, o algoritmo não é livre de espera porque algum processo pode ser bem sucedido em inserir suas operações na lista infinitas vezes, atrasando outros processos para sempre.

A política de acesso para a construção universal (apresentada na figura 6.6) define que uma tupla SEQ com um segundo campo *pos* só pode ser inserida no espaço (usando *cas*) se existe uma tupla com segundo campo com valor $pos - 1$ no espaço.

Algoritmo 14 Construção universal não-bloqueante (processo p_i).

Variáveis compartilhadas:

1: $ts = \varnothing$ {PEATS}

Variáveis locais:

2: $state = S_T$ {estado atual do objeto}
3: $pos = 0$ {posição do fim da lista de operações}

invoked *inv*

4: **loop**
5: $pos \leftarrow pos + 1$
6: **if** $ts.cas(\langle SEQ, pos, ?pos\_inv\rangle, \langle SEQ, pos, inv\rangle)$ **then**
7: $\langle state, reply\rangle \leftarrow apply_T(state, inv)$
8: **return** $reply$
9: **end if**
10: $\langle state, reply\rangle \leftarrow apply_T(state, pos\_inv)$
11: **end loop**

Object State TS
R_{cas}: $execute(cas(\langle SEQ, pos, x\rangle, \langle SEQ, pos, inv\rangle)) \ :-$
$invoke(p, cas(\langle SEQ, pos, x\rangle, \langle SEQ, pos, inv\rangle)) \wedge formal(x) \wedge$
$(pos = 1 \vee \exists y : \langle SEQ, pos - 1, y\rangle) \in TS)$

Figura 6.6: Política de acesso para o PEATS usado no algoritmo 14.

A prova de correção do algoritmo é baseada nos seguintes lemas:

Lema 10 *Em qualquer execução do sistema, as seguintes propriedades são invariantes do PEATS usado no algoritmo 14:*

1. *Para qualquer $pos \geq 1$, existe no máximo uma tupla $\langle SEQ, pos, inv\rangle$ no espaço de tuplas;*
2. *Para qualquer tupla $\langle SEQ, pos, inv\rangle$ no espaço de tuplas, com $pos > 1$, existe exatamente uma tupla $\langle SEQ, pos - 1, inv\rangle$ no espaço.*

Prova: Estas duas invariantes são conseqüências diretas do algoritmo 14 e de sua política de acesso (figura 6.6):

1. A partir da política de acesso pode-se verificar que uma tupla só pode ser inserida no espaço através de uma invocação da operação *cas*, onde o molde e a entrada passados como argumento devem ser tuplas SEQ com o mesmo número de seqüência *seq* e o campo da invocação do molde deve ser formal. Com essa propriedade e a definição do *cas*, é fácil ver que não pode haver duas tuplas SEQ com o mesmo número de seqüência no espaço de tuplas.
2. Novamente, a partir da política de acesso é possível ver que a operação *cas* só pode ser executada para inserir uma operação na posição *pos* se existe uma tupla no espaço com número de seqüência uma unidade menor. Isto garante que existe uma tupla $\langle SEQ, pos - 1, inv\rangle$ no espaço quando da inserção da tupla *pos*. A garantia de que não existe mais de uma tupla dessa é conseqüência direta da primeira parte do lema. ■

Lema 11 *A construção universal do algoritmo 14 é não-bloqueante.*

Prova: Este lema é provado por contradição. Considere, sem perda de generalidade, uma execução α onde apenas dois processos corretos p_1 e p_2 executam as invocações inv_1 e inv_2, respectivamente. Suponha que eles permanecem parados para sempre, não recebendo resposta para essas invocações. Vamos mostrar que α não existe. Uma inspeção no algoritmo mostra que os processos permanecem atualizando suas cópias do estado do objeto até que eles executem a mais recente operação inserida no espaço de tuplas (com campo posição igual à *pos*). Neste ponto, p_1 e p_2 vão tentar adicionar suas invocações no espaço na posição $pos+1$ através da execução da operação *cas* (linha 6). Como assumimos que o PEATS é linearizável, as duas invocações a operação *cas* devem acontecer uma após a outra, assim ou a tupla SEQ com inv_1 ou a com inv_2 será inserida no espaço na posição $pos+1$. O processo bem sucedido executando *cas* para esta posição vai inserir sua invocação para o objeto, e vai retornar o resultado desta operação (linhas 7 e 8). Este fato contradiz a definição de α. ■

Teorema 7 *O algoritmo 14 provê uma construção universal não-bloqueante.*

Prova: O resultado apresentado no lema 10 implica na existência de uma ordem total das operações executadas no objeto emulado. Através de uma inspeção no algoritmo, é fácil ver que um processo correto atualiza sua cópia do estado do objeto emulado aplicando a função determinista $apply_T$ em todas as tuplas SEQ na ordem definida pelos seus números de seqüência. Desta forma, todas as operações são executadas na mesma ordem por todos os processos corretos e esta ordem é condizente com especificação seqüencial do objeto, provida pela função $apply_T$. Isto é suficiente para provar que a construção universal satisfaz linearização. O lema 11 prova que a construção é não-bloqueante. ■

6.5.2 Melhorando a Construção Universal t-resistente de [85]

Existe apenas uma outra construção universal tolerante a faltas bizantinas na literatura [85]. Esta construção é t-resistente e se baseia no uso de *sticky* bits e objetos de consenso binário forte t-limiar. Este mesmo trabalho mostra que é possível emular qualquer objeto tolerante a faltas bizantinas em um sistema com memória compartilhada protegido por ACLs. Entretanto, a construção é complexa e requer uma grande quantidade de objetos universais. Mais especificamente, para executar i operações nesta construção, é necessário $i\lceil \log m \rceil$ objetos de consenso binário forte e $i\lceil \log |INVOKE| \rceil$ *sticky* bits. Usando apenas ACLs para implementar estes objetos de consenso, a resistência ótima ($m \geq 3t+1$) é atingida apenas se o algoritmo de consenso apresentado em [4] for usado. Este algoritmo requer $(m+1)\binom{2t+1}{t}$ *sticky* bits em sua implementação. Assumindo que $\log |INVOKE|$ é uma constante, este algoritmo de consenso aplicado à construção universal resulta em um algoritmo com complexidade de bits igual a $O(i\, m(\log m)\binom{2t+1}{t})$.

O algoritmo de consenso binário forte da seção 6.4.2 usa apenas $m(\lceil \log m \rceil + 1) + (1 + (t+1)\lceil \log m \rceil)$ bits (e apenas um objeto, o PEATS). Se usarmos este algoritmo na construção universal de [85] é possível obter uma construção universal com complexidade de bits igual a $O(i(n+t)(\log n)^2))$. Isto significa uma enorme melhoria no uso de memória. Adicionalmente, podemos emular os *sticky*

bits usados na construção universal usando o objeto de consenso fraco da seção 6.4.1. Desta forma, a construção completa pode ser implementada usando apenas um objeto de memória compartilhada, o PEATS.

6.6 Trabalhos Relacionados

Neste capítulo apresentamos vários algoritmos para memória compartilhada tolerante a faltas bizantinas usando um espaço de tuplas aumentado. Até onde sabemos, os únicos trabalhos anteriores que usam este tipo de objeto para resolver problemas fundamentais em computação distribuída são [11, 117]. Entretanto, estes trabalhos tratam apenas do problema do consenso livre de espera em sistema sujeitos a faltas por parada (sem falhas bizantinas nos processos). Nosso trabalho é o primeiro a fazer uso deste tipo de objeto como suporte para a coordenação de processos sujeitos a falhas bizantinas.

Sistemas assíncronos com memória compartilhada e processos sujeitos a faltas bizantinas foram inicialmente estudados independentemente por Attie [8] e Malkhi et al. [85]. O trabalho em [8] mostra que consenso fraco não pode ser resolvido usando apenas objetos "resetáveis"[8]. Este resultado implica no fato de que para resolver consenso nesse modelo, os algoritmos **têm** de utilizar algum tipo de objeto persistente (não-resetável), como *sticky* bits. O PEATS usado em nosso algoritmos pode ser visto como um objeto persistente já que as políticas de acesso especificadas não permitem que os processos reiniciem o estado do objeto.

O trabalho apresentado em [85] se baseia no uso de objetos de memória compartilhada protegidos por ACLs e define um algoritmo para consenso forte binário t-limiar e uma construção universal t-resistente. A primeira usa $2t+1$ *sticky* bits e requer $m \geq (t+1)(2t+1)$ processos. Este trabalho também mostra que não se pode construir objetos de consenso forte binários com $m \leq 3t$ processos neste modelo de computação.

Em um trabalho mais recente, Alon et al. [4] estendeu os resultados anteriores apresentando um algoritmo para consenso forte binário com resistência ótima ($m \geq 3t+1$) que usa um número exponencial de *sticky* bits e requer também um número exponencial de *rounds* (veja seção 6.5.2). Este trabalho prova vários limites inferiores no número de objetos requeridos para implementar consenso, incluindo um compromisso rígido caracterizando o número de objetos requeridos para implementação de consenso forte: um número polinomial de processos requer um número exponencial de objetos e vice-versa. Este resultado enfatiza o poder do uso de ACLs para limitar a ação de processos maliciosos, mas também mostra as limitações do modelo, especialmente em termos do grande número de objetos requeridos para se obter resistência ótima. A abordagem proposta neste capítulo usa um modelo diferente de computação, portanto esse compromisso não faz sentido. Neste capítulo introduzimos os objetos protegidos por políticas de granularidade final (PEOs) e mostramos que seu uso permite a construção de algoritmos muito mais simples, elegantes e eficientes que os baseados no modelo de proteção por ACLs, explorados nos trabalhos anteriores [4, 8, 85].

[8] Um objeto o é **resetável (*resettable*)** se, dado qualquer um de seus estados alcançáveis, existe uma seqüência de operações que podem trazer o objeto de volta ao seu estado inicial [8].

O tipo de controle de acesso por políticas usado na proteção do PEATS foi inspirado na abordagem LGI (*Law-Governed Interaction*) [96]. O uso desta abordagem para proteção de espaços de tuplas centralizados é apresentado em [95].

6.7 Considerações Finais

A proposta para computação distribuída com memória compartilhada acessada por processos sujeitos a falhas bizantinas apresentada neste capítulo difere das abordagens anteriores onde os objetos eram protegidos por listas de controle de acesso (ACLs). Nossa abordagem se baseia no uso de políticas de acesso com granularidade fina para especificar regras que permitem ou negam a invocação de uma operação em um objeto basendo-se nos argumentos da operação, na identidade de seu invocador e no estado do objeto. As construções apresentadas neste capítulo (objetos de consenso e construção universal) demonstram que esta abordagem pode ser usada para construção de algoritmos simples e elegantes, ao custo da definição de políticas de acesso para os objetos de memória usados nestes.

No que diz respeito a concretização do PEATS (ou qualquer outro PEO), a quantidade de recursos requeridos para implementação destes objetos é a mesma dos objetos protegidos por ACLs previamente usados para resolver problemas em nosso modelo: um monitor de referência local[9] instalado em cada réplica da implementação do objeto. O capítulo 3 apresentou uma implementação do PEATS.

Uma característica inerente da abordagem proposta é que sua utilidade é limitada quando usada na implementação de objetos persistentes simples, como *sticky* bits (seu uso é equivalente as ACLs). O verdadeiro potencial dos PEOs aparece apenas quando objetos mais elaborados, como espaços de tuplas aumentados, são considerados.

[9] No sentido de que as decisões sobre o acesso ou não a uma operação são tomadas localmente.

Capítulo 7

Conclusão

7.1 Visão Geral do Trabalho

Esta tese apresentou um estudo sobre implementação e uso do modelo de coordenação por espaços de tuplas em sistemas onde os processos estão sujeitos a faltas bizantinas.

O trabalho pode ser dividido em três partes básicas. A primeira compreende a arquitetura para espaço de tuplas com segurança de funcionamento e todos os mecanismos definidos nesta (descritos como camadas na seção 3.5). Dentre estes mecanismos, destaca-se a provisão de confidencialidade a partir do uso de um esquema de compartilhamento de segredo publicamente verificável e do uso de *fingerprints* de tuplas.

A segunda parte da tese apresenta duas construções para espaço de tuplas tolerante a faltas bizantinas. Estas construções introduzem o conceito de sistemas de quóruns ativos, em que o objetivo é mesclar protocolos para sistemas de quóruns com algoritmos de acordo visando obter protocolos mais leves para operações com semântica simples. As construções apresentadas, BTS e LBTS, apresentam importantes benefícios quando comparadas com construções similares baseadas em replicação Máquina de Estados.

A última parte da tese trata basicamente do uso de um espaço de tuplas aumentado protegido por políticas de segurança na coordenação de processos sujeitos a falhas bizantinas. Os algoritmos apresentados nesta seção mostram que esta abordagem resulta em protocolos muito mais eficientes, simples e elegantes, quando comparados a resultados anteriores sobre o problema de compartilhamento de memória entre processos bizantinos.

7.2 Revisão dos Objetivos

Esta seção revisa os objetivos da tese, enunciados na seção 1.2, e relembra quais foram as contribuições desta tese no sentido de atingir estes objetivos.

1. **Definição do modelo de coordenação generativa dentro de um arcabouço teórico de sistemas distribuídos.**

   Na seção 2.1.5 o modelo de sistema considerado nesta tese foi formalizado como uma composição de autômatos de I/O. Esta composição considera dois níveis de abstração: um conjunto de tamanho arbitrário de processos de aplicação que se coordenam através do espaço de tuplas com segurança de funcionamento (visto como um único autômato); e um conjunto de clientes de um algoritmo de replicação que acessam os servidores que implementam o espaço de tuplas. Os algoritmos executados nestes clientes e servidores são dependentes da construção para espaço de tuplas usada e juntos compõem o autômato do espaço de tuplas.

2. **Desenvolvimento de uma arquitetura para espaço de tuplas com segurança de funcionamento que contemple diversos mecanismos de tolerância a faltas e segurança.**

   Na seção 3.5 apresentamos tal arquitetura. Ela se baseia no uso de um algoritmo para replicação Máquina de Estados e na integração de uma série de mecanismos de segurança que oferecem proteção ao espaço contra diferentes tipos de ataques perpetrados por clientes e servidores maliciosos. Os mecanismos de segurança definidos para o espaço de tuplas são:

   - **controle de acesso**: permite a associação de um conjunto de credenciais requeridas para leitura e remoção à cada tupla inserida no espaço. Desta forma, um processo, ao armazenar uma tupla no espaço, define quais processos podem ler e quais processos podem remover a tupla. Este mecanismo permite também a definição de credenciais mínimas a serem apresentadas para inserção de uma tupla no espaço de tuplas.
   - **verificação de políticas**: o uso de políticas de granularidade fina associadas a um espaço de tuplas permite que o acesso ao espaço seja regulado por um conjunto de regras dependentes de aplicação que são usadas pelo monitor de referência para permitir ou negar o acesso ao espaço de acordo com *(i.)* a operação sendo invocada no espaço (incluindo seus argumentos); *(ii.)* a identidade do processo invocador; e *(iii.)* o estado do espaço de tuplas. Através do uso deste tipo de políticas é possível limitar de forma cabal o estrago causado por processos maliciosos.
   - **confidencialidade**: este mecanismo garante que nenhum servidor tem acesso total às informações contidas em uma tupla privada. O mecanismo se baseia no uso de um esquema de compartilhamento de segredo em que cada servidor recebe um fragmento da tupla. Para reconstrução da tupla é necessária a participação de pelo menos $f + 1$ servidores. Como por premissa assumimos que no máximo f servidores podem falhar, o mecanismo garante que uma tupla não pode ser lida sem a participação de servidores corretos. Como estes servidores respeitam as regras de controle de acesso definidas para a tupla, a confidencialidade é mantida, mesmo em caso de comprometimento dos servidores.

   Todos estes mecanismos foram definidos de forma a poderem ser integrados em uma estratificação única capaz de prover os principais atributos de segurança de funcionamento.

3. **Definição de algoritmos de replicação para espaços de tuplas tolerantes a faltas bizantinas baseados tanto no modelo de replicação Máquina de Estados quanto em sistemas de quóruns bizantinos.**

   Nesta tese apresentamos três construções para espaço de tuplas: uma baseada em replicação Máquina de Estados e duas baseadas na integração de protocolos para sistemas quóruns com algoritmos de acordo. As características destas construções são apresentadas na tabela 7.1.

Construção	Abordagem	Servidores	Linearização	Conf.	Políticas
SMR-TS	Máquina de Estados	$n \geq 3f+1$	sim	sim	sim
BTS	Quóruns + Acordo	$n \geq 3f+1$	não	sim	não
LBTS	Quóruns + Acordo	$n \geq 4f+1$	sim	não	não

Tabela 7.1: Construções para espaços de tuplas tolerantes a faltas bizantinas.

   As duas primeiras colunas apresentam o nome da construção e qual abordagem em que seus algoritmos se baseiam. Além das características de número de servidores necessários para que a construção tolere f faltas bizantinas (coluna 3) e se a construção satisfaz a condição de linearização (coluna 4), a tabela 7.1 também mostra que mecanismos de segurança definidos na arquitetura para espaço de tuplas com segurança de funcionamento a construção suporta[1]. Desta forma, é possível ver que nenhuma das construções baseadas em sistemas de quóruns suportam a verificação de políticas (coluna 6), e que o esquema de confidencialidade não é suportado pelo LBTS.

   A tabela 7.1 também mostra que a única construção que requer um número mínimo de servidores, satisfaz linearização e suporta a implementação de todos os mecanismos de segurança é o SMR-TS (espaço de tuplas baseado em replicação Máquina de Estados). Isto se deve ao fato de, do ponto de vista arquitetural, este espaço de tuplas emular perfeitamente um serviço centralizado, mantendo um estado equivalente em todas as réplicas do espaço. O LBTS, que também satisfaz a condição de linearização (e portanto, formalmente emula um objeto centralizado cujos acessos são atômicos), não suporta nenhum dos mecanismo de segurança apresentados na tabela (devido, principalmente, a possibilidade de divergência nos estados das réplicas). Por outro lado, conforme visto no capítulo 4, as construções baseadas em quóruns não requerem difusão com ordem total na implementação de todos os protocolos para as operações, e portanto são potencialmente mais eficientes que seus equivalentes baseados em replicação Máquina de Estados, conforme pode ser visto nas tabelas 4.1 e 4.2, apresentadas no capítulo 4.

4. **Desenvolvimento de um mecanismo para confidencialidade em espaço de tuplas tolerantes a faltas bizantinas.**

   O mecanismo de confidencialidade apresentado no capítulo 5 apresenta uma solução simples e eficaz para a manutenção de informações confidenciais no espaço de tuplas. Este mecanismo faz uso de um esquema de compartilhamento de segredo verificável publicamente e de funções de resumo criptográfico para permitir o acesso por conteúdo às tuplas (o que é inerente ao

[1] O controle de acesso não é apresentado na tabela pois é trivialmente suportado por todos as construções definidas nesta tese.

modelo generativo), mesmo este conteúdo estando distribuído entre os servidores que implementam o espaço.

5. **Investigação do poder do modelo de coordenação generativa como mecanismo de coordenação entre processos.**

   Nesta tese introduzimos os objetos protegidos por políticas (PEOs) e apresentamos uma série de algoritmos em que os processos se coordenam através de um PEO especial, o PEATS (*Policy-Enforced Augmented Tuple Space*). Apresentamos algoritmos que solucionam 4 variantes do problema de consenso e implementamos duas construções universais capazes de emular qualquer objeto de memória determinista. Todos esses algoritmos se baseiam no uso do PEATS. Estes resultados demonstram que um espaço de tuplas aumentado (que suporta a operação *cas*) protegido por políticas de granularidade fina é um objeto universal, e permite a criação de algoritmos extremamente eficientes (em termos da memória utilizada) e simples. Por outro lado, o custo da simplicidade destes algoritmos é compensado pelo trabalho extra em se definir as regras para a política de controle de acesso ao espaço de tuplas, que é específica para cada algoritmo.

7.3 Contribuições e Resultados desta Tese

Esta tese apresentou uma série de contribuições, tanto na área de coordenação quanto na área de algoritmos distribuídos. Nesta seção listamos algumas destas contribuições[2].

- Definição de espaço de tuplas com segurança de funcionamento e proposta de uma arquitetura que combina diferentes mecanismos de segurança e tolerância a faltas para construção de um espaço com esta qualidade de serviço. Os trabalhos prévios sobre segurança de funcionamento em espaços de tuplas se preocupavam com tolerância a faltas **ou** segurança. Além disso, a arquitetura de segurança de funcionamento definida nesta tese é a primeira a considerar tolerância a faltas bizantinas;

- Definição de uma estratégia de replicação baseada em sistemas de quóruns bizantinos que permite a implementação de abstrações mais fortes que registradores (de acordo com a hierarquia livre de espera [69]) satisfazendo terminação livre de espera. Apresentamos duas construções para espaços de tuplas que fazem uso de alguns protocolos baseados em sistemas de quóruns e outros baseados em primitivas de acordo. Estas construções apresentam importantes benefícios em termos de métricas de algoritmos distribuídos quando comparadas com a abordagem usualmente empregada para implementação de objetos mais fortes que registradores (replicação Máquina de Estados). Trabalhos anteriores para sistemas de quóruns bizantinos implementavam apenas registradores com operações de leitura e escrita ou suportavam a implementação de objetos mais fortes satisfazendo condições de terminação mais fracas que a terminação livre de espera;

[2]Cada capítulo desta tese cita as contribuições específicas no que tange o problema atacado no capítulo e a literatura relacionada.

- Definição de um novo modelo de computação para sistemas distribuídos em que os processos se comunicam através de memória compartilhada. Neste modelo, os objetos de memória compartilhada são protegidos por políticas mais refinadas que definem que alterações no estado do objeto são legais, de acordo com o algoritmo distribuído que faz uso do objeto. Este modelo representa uma ruptura radical com o modelo de proteção previamente usado em sistemas com memória compartilhada por processos sujeitos a falhas bizantinas, baseado em listas de controle de acesso.

Uma parte das contribuições listadas acima foram publicadas [16, 17, 18, 19, 20, 21, 22, 99].

7.4 Perspectivas Futuras

As contribuições apresentadas nesta tese abrem um amplo leque de possíveis temas para trabalhos futuros a serem explorados. Talvez o primeiro e mais imediato destes trabalhos seja a concretização de um *middleware* baseado na arquitetura definida nesta tese para coordenação entre processos sujeitos a falhas bizantinas. O suporte oferecido por este *middleware* deve contemplar todos os mecanismos definidos na arquitetura para espaço de tuplas com segurança de funcionamento sem sacrificar a elegância e simplicidade do modelo de programação provido pela coordenação generativa.

Um tipo de mecanismo para segurança de funcionamento não provido em nossa arquitetura de espaço de tuplas é o suporte a transações. Através deste mecanismo, seria possível garantir que, ou uma seqüência de operações no espaço de tuplas é executada, ou nenhuma das operações desta seqüência tomam efeito. A provisão desse mecanismo em um cenário onde os processos estão sujeitos a faltas bizantinas é um tanto quanto complicada até mesmo pelo conceito de protocolo de confirmação: se um processo não confirma a transação, então a transação não é confirmada por nenhum processo correto. Este requisito permite que processos maliciosos atrapalhem a vivacidade das aplicações através da não confirmação contínua de transações. Acreditamos que somente através de uma interpretação diferente do problema da confirmação atômica para cenários de faltas bizantinas este mecanismo possa ser implementado com sucesso. A definição desta interpretação e a concretização de um suporte para transações nos espaços de tuplas definidos nesta tese constituem-se em uma interessante perspectiva de trabalho futuro.

Um outro trabalho que não foi realizado nesta tese, mas que é muito interessante, é generalização da abordagem dos sistema de quóruns ativos. Nesta abordagem, os algoritmos para sistemas de quóruns e os algoritmos de acordo são usados na implementação de diferentes operações de um mesmo serviço replicado. A idéia foi introduzida nesta tese através da concepção de duas construções para espaços de tuplas baseados nesta abordagem de replicação (BTS e LBTS). Como trabalho futuro é esperada a generalização desta técnica através do desenvolvimento de um algoritmo de replicação para serviços deterministas que utilize protocolos de sistemas de quóruns para operações de leitura e escrita e o PAXOS BIZANTINO modificado (de forma semelhante ao protocolo para operação *inp* do LBTS) para operações de leitura-escrita.

Com relação ao mecanismo de confidencialidade, pelo menos duas melhorias podem ser feitas. Em primeiro lugar, seria de extremo interesse a integração do *fingerprint* de uma tupla ao esquema de compartilhamento de segredo verificável de tal forma que seja possível verificar através das funções de verificação providas pelo esquema se este *fingerprint* corresponde à tupla da qual o servidor armazena o fragmento. Esta integração permitiria que os servidores percebessem que uma tupla é inválida durante sua inserção e descartassem tuplas cujos *fingerprints* não possam ser verificados. Conjecturamos que esta integração pode ser feita através da inclusão do resumo criptográfico do *fingerprint* na computação do segredo, de tal forma que as verificações das provas possam informar se a tupla é inválida. Uma outra possibilidade de trabalho futuro no que tange ao mecanismo de confidencialidade diz respeito a definição de um novo mecanismo que possa ser usado no LBTS. Um mecanismo como esse deve permitir que novos fragmentos de uma tupla sejam gerados a partir da tupla limpa, sem invalidar os fragmentos antigos já distribuídos. Um possível ponto de partida para a resolução deste problema é a investigação de esquemas de compartilhamento de segredo pró-ativos como o proposto em [70], que oferecem protocolos de recuperação de segredos que podem ser usados para gerar fragmentos de um segredo sem invalidar os fragmentos já distribuídos.

Com relação ao modelo de computação com memória compartilhada protegida por políticas, vislumbramos também pelo menos duas possibilidades futuras a serem exploradas. No capítulo 6 apresentamos construções universais que satisfazem terminação livre de bloqueio e terminação t-limiar. No entanto, seria de grande interesse implementar uma construção universal baseada no PEATS que satisfizesse terminação livre de espera. Uma segunda perspectiva futura diz respeito a resposta da seguinte questão: seria possível transformar algoritmos livre de espera tolerante a faltas por parada em algoritmos tolerantes a faltas bizantinas através do uso de políticas para proteção da memória compartilhada? Se a resposta for afirmativa, resta-nos descobrir que classes de algoritmos podem ser transformadas e se essa transformação pode ser feita automaticamente.

A última perspectiva de trabalho futuros diz respeito as aplicações do espaço de tuplas com segurança de funcionamento. Acreditamos que abstrações como o PEATS podem ser utilizadas com sucesso na coordenação de processos em sistemas dinâmicos e heterogêneos, como grades computacionais, redes par a par e serviços Web colaborativos.

Referências Bibliográficas

[1] Abd-El-Malek, M., Ganger, G., Goodson, G., Reiter, M., and Wylie, J. (2005). Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles - SOSP'05*, pages 59–74.

[2] Abdullahi, S. E. and Ringwood, G. A. (1998). Garbage collecting the Internet: a survey of distributed garbage collection. *ACM Computing Surveys*, 30(3):330–373.

[3] Abraham, I., Chockler, G., Keidar, I., and Malkhi, D. (2006). Byzantine disk paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408.

[4] Alon, N., Merritt, M., Reingold, O., Taubenfeld, G., and Wright, R. (2005). Tight bounds for shared memory systems acessed by Byzantine processes. *Distributed Computing*, 18(2):99–109.

[5] Anderson, J. H., Kim, Y.-J., and Herman, T. (2003). Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2-3):75–110.

[6] Anderson, J. P. (1972). Computer security technology planning study. ESD-TR 73-51, U. S. Air Force Electronic Systems Division.

[7] Aspnes, J. (2003). Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175.

[8] Attie, P. C. (2002). Wait-free Byzantine consensus. *Information Processing Letters*, 83(4):221–227.

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

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

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

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

[13] Bazzi, R. A. and Ding, Y. (2004). Non-skipping timestamps for Byzantine data storage systems. In *Proceedings of 18th International Symposium on Distributed Computing - DISC 2004*, pages 405–419.

[14] Bellare, M. and Rogaway, P. (1993). Random oracles are practical: A paradigm for designing efficient protocols. In *Proc. of the 1st ACM Conference on Computer and Communications Security*, pages 62–73.

[15] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

[16] Bessani, A. N., Alchieri, E. A. P., Correia, M., da Silva Fraga, J., and Lung, L. C. (2006a). 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*.

[17] Bessani, A. N., Correia, M., da Silva Fraga, J., and Lung, L. C. (2006b). Brief announcement: Decoupled quorum-based Byzantine-resilient coordination in open distributed systems. In *Proceedings of 20th International Symposium on Distributed Computing - DISC 2006*.

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

[19] Bessani, A. N., Correia, M., da Silva Fraga, J., and Lung, L. C. (2006d). Towards a dependable tuple space. DI-FCUL TR 06–4, Department of Informatics, University of Lisbon.

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

[21] Bessani, A. N., da Silva Fraga, J., and Lung, L. C. (2006e). BTS: A Byzantine fault-tolerant tuple space. In *Proceedings of the 21st ACM Symposium on Applied Computing - SAC 2006*, pages 429–433.

[22] Bessani, A. N., Dantas, W. S., Alchieri, E. A. P., and da Silva Fraga, J. (2006f). Analisando o custo do armazenamento tolerante a faltas Bizantinas: PAXOS vs sistemas de quóruns. In *Anais do 7o Workshop de Testes e Tolerância a Faltas - WTF 2006*.

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

[24] Blakley, G. R. (1979). Safeguarding cryptographic keys. In *Proceedings of the 1979 AFIPS National Computer Conference*, volume 48, pages 313–317.

[25] Bracha, G. (1984). An asynchronous $\lfloor (n-1)/3 \rfloor$-resilient consensus protocol. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing - PODC'84*, pages 154–162.

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

[27] Busi, N., Gorrieri, R., and Zavattaro, G. (2000). On the expressiveness of Linda coordination primitives. *Information and Computation*, 156(1-2):90–121.

[28] Busi, N. and Zavattaro, G. (2000). Event notification in data-driven coordination languages: Comparing the ordered and unordered interpretations. In *Proceedings of the 15th ACM symposium on Applied computing - SAC 2000*, pages 233–239.

[29] Busi, N. and Zavattaro, G. (2003). Expired data collection in shared dataspaces. *Theoretical Computer Science*, 298(3):529–556.

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

[31] Cachin, C. (2001). Distributing trust on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks - DSN 2001*, pages 183–192.

[32] Cachin, C., Kursawe, K., and Shoup, V. (2000). Random oracles in constantinople: Practical asynchronous Byzantine agreement using criptography. In *Proceedings 19th ACM Symposium on Principles of Distributed Computing - PODC 2000*, pages 123–132.

[33] Cachin, C. and Poritz, J. A. (2002). Secure intrusion-tolerant replication on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks - DSN 2002*, pages 167–176.

[34] Cachin, C. and Tessaro, S. (2005). Asynchronous verifiable information dispersal. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems - SRDS 2005*, pages 191–202.

[35] Cachin, C. and Tessaro, S. (2006). Optimal resilience for erasure-coded Byzantine distributed storage. In *Proceedings of the International Conference on Dependable Systems and Networks - DSN 2006*, pages 115–124.

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

[37] Castro, M., Rodrigues, R., and Liskov, B. (2003). BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269.

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

[39] Chaudhuri, S. (1993). More choices allow more faults: Set consensus problems in totally asynchronous systems. *Information and Computation*, 105(1):132–158.

[40] Chor, B., Goldwasser, S., Micali, S., and Awerbush, B. (1985). Verifiable secret sharing and achieving simultaneity in the presence of faults. In *Proceedings of the 26th IEEE Symposium on Foundations of Computer Science - FOCS'85*, pages 383–395.

[41] Correia, M., Neves, N. F., Lung, L. C., and Veríssimo, P. (2005). Low complexity Byzantine-resilient consensus. *Distributed Computing*, 17(3):237–249.

[42] Correia, M., Neves, N. F., and Veríssimo, P. (2004). How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems - SRDS 2004*, pages 174–183.

[43] Correia, M., Neves, N. F., and Veríssimo, P. (2006). From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96.

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

[45] Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Hadzilacos, V., Kouznetsov, P., and Toueg, S. (2004). The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the 23rd annual ACM Symposium on Principles of Distributed Computing - PODC 2004*, pages 338–346.

[46] Desmedt, Y. and Frankel, Y. (1990). Threshold criptosystems. In *Proceedings of the 9th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'89*, pages 307–315.

[47] Desmedt, Y. and Frankel, Y. (1992). Shared generation of authenticators and signatures (extended abstract). In *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'92*, pages 457–469.

[48] Deswarte, Y., Blain, L., and Fabre, J.-C. (1991). Intrusion tolerance in distributed computing systems. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 110–121.

[49] Deswarte, Y., Kanoun, K., and Laprie, J.-C. (1998). Diversity against accidental and deliberate faults. In *Computer Security, Dependability, & Assurance: From Needs to Solutions - CSDA'98*, pages 171–181.

[50] Dierks, T. and Allen, C. (1999). The TLS Protocol Version 1.0 (RFC 2246). IETF Request For Comments.

[51] Diffie, W. and Hellman, M. E. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654.

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

[53] Dutta, P., Guerraoui, R., Levy, R. R., and Chakraborty, A. (2004). How fast can a distributed atomic read be? In *Proceedings of the 23rd annual ACM Symposium on Principles of Distributed Computing - PODC 2004*, pages 236–245.

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

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

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

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

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

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

[60] Gelernter, D. and Bernstein, A. J. (1982). Distributed communication via global buffer. In *Proceedings of the 1st Annual ACM Symposium on Principles of Distributed Computing - PODC'82*, pages 10–18.

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

[62] Gemmell, P. S. (1997). An introduction to threshold cryptography. *Cryptobytes—The Technical Newsletter of RSA Laboratories*, 2(3):7–12.

[63] Goldreich, O. (2001). *Fundamentals of Criptography: Basic Tools*, volume 1. Cambridge Press, Cambridge - Massachussets.

[64] Goodson, G. R., Wylie, J. J., Ganger, G. R., and Reiter, M. K. (2004). Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of Dependable Systems and Networks - DSN 2004*, pages 135–144.

[65] Graham S. at al. (2004). WS-Base Notification. Disponível em http://ifr.sap.com/ws-notification/WS-BaseNotification.pdf.

[66] Hadzilacos, V. and Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical Report TR 94-1425, Department of Computer Science, Cornell University, New York - USA.

[67] Herlihy, M. (1991). Wait-free synchronization. *ACM Transactions on Programing Languages and Systems*, 13(1):124–149.

[68] Herlihy, M., Lucangco, V., and Moir, M. (2003). Obstruction-free syncronization: double-ended queues as an example. In *Proceedings of 23th IEEE International Conference on Distributed Computing Systems - ICDCS 2003*, pages 522–529.

[69] Herlihy, M. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems*, 12(3):463–492.

[70] Herzberg, A., Jarecki, S., Krawczyk, H., and Yung, M. (1995). Proactive secret sharing or how to cope with perpetual leakage. In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'95*, pages 339–352.

[71] Jeong, K. and Shasha, D. (1994). PLinda 2.0: A transactional checkpointing approach to fault tolerant Linda. In *Proceedings of the 13th IEEE Symposium on Reliable Distributed Systems - SRDS'94*, pages 96–105.

[72] Junqueira, F. P. and Marzullo, K. (2003). Synchronous consensus for dependent process failures. In *Proceedings of 23th IEEE International Conference on Distributed Computing Systems - ICDCS 2003*, pages 274–283.

[73] Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. (2001). The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4):371–406.

[74] L. F. Cabrera et al. (2005). Web Services Coordination Specification - version 1.0. Disponível em http://www-128.ibm.com/developerworks/library/specification/ws-tx/.

[75] Lakshmanan, S., Ahamad, M., and Venkateswaran, H. (2003). Responsive security for stored data. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):818–828.

[76] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

[77] Lamport, L. (1998). The part-time parliament. *ACM Transactions Computer Systems*, 16(2):133–169.

[78] Lamport, L. (2001). Paxos made simple. *ACM SIGACT News*, 32(4):18–25.

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

[80] Liskov, B. and Rodrigues, R. (2006). Tolerating Byzantine faulty clients in a quorum system. In *Proceedings of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*.

[81] Littlewood, B. and Strigini, L. (2004). Redundancy and diversity in security. In *Proceedings of the 9th European Symposium on Research Computer Security - ESORICS 2004*, pages 423–438.

[82] Lucchi, R. and Zavattaro, G. (2004). WSSecSpaces: a secure data-driven coordination service for web services applications. In *Proceedings of the 19th ACM Symposium on Applied Computing - SAC 2004*, pages 487–491.

[83] Lynch, N. A. (1996). *Distributed Algorithms*. Morgan Kauffman.

[84] Lynch, N. A. and Tuttle, M. R. (1987). Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th annual ACM Symposium on Principles of Distributed Computing - PODC'87*, pages 137–151.

[85] Malkhi, D., Merritt, M., Reiter, M., and Taubenfeld, G. (2003). Objects shared by Byzantine processes. *Distributed Computing*, 16(1):37–48.

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

[87] Malkhi, D. and Reiter, M. (1998b). Secure and scalable replication in Phalanx. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems - SRDS'98*, pages 51–60.

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

[89] Mamei, M. and Zamboneli, F. (2004). Self-maintained distributed tuples for field-based coordination in dynamic networks. In *Proceedings of the 19th ACM Symposium on Applied Computing - SAC 2004*, pages 479–486.

[90] Marsh, M. A. and Schneider, F. B. (2004). CODEX: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47.

[91] Martin, J.-P. and Alvisi, L. (2005). Fast Byzantine consensus. In *Proceedings of the Dependable Systems and Networks - DSN 2005*, pages 402–411.

[92] Martin, J.-P., Alvisi, L., and Dahlin, M. (2002a). Minimal Byzantine storage. In *Proceedings of the 16th International Symposium on Distributed Computing - DISC 2002*, pages 311–325.

[93] Martin, J.-P., Alvisi, L., and Dahlin, M. (2002b). Small Byzantine quorum systems. In *Proceedings of the Dependable Systems and Networks - DSN 2002*, pages 374–388.

[94] Menezes, R. and Wood, A. (2006). The fading concept in tuple space systems. In *Proceedings of the 21st ACM Symposium on Applied Computing – SAC 2006*.

[95] Minsky, N. H., Minsky, Y. M., and Ungureanu, V. (2000). Making tuple-spaces safe for heterogeneous distributed systems. In *Proceedings of the 15th ACM Symposium on Applied Computing - SAC 2000*, pages 218–226.

[96] Minsky, N. H. and Ungureanu, V. (2000). Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305.

[97] Murphy, A., Picco, G., and Roman, G.-C. (2001). LIME: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems - ICDCS 2001*, pages 524–533.

[98] National Institute of Standards and Technology (2002). Secure Hash Standard. Federal Information Processing Standards Publication 180-2.

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

[100] Object Management Group (2002). The common object request broker architecture: Core specification v3.0. OMG Standart formal/02-12-06.

[101] Omicini, A. (1999). On the semantics of tuple-based coordination models. In *Proceedings of the 14th ACM Symposium on Applied Computing - SAC'99*, pages 175–182.

[102] Oppliger, R. (1998). Security at the internet layer. *IEEE Computer*, 31(9):43–47.

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

[104] Pease, M., Shostak, R., and Lamport, L. (1980). Reaching agreement in the presence of faults. *Journal of ACM*, 27(2):228–234.

[105] Plotkin, S. A. (1989). Sticky bits and universality of consensus. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing - PODC'89*, pages 159–175.

[106] Postel, J. (1981). Transmission control protocol (rfc 793). IETF Request For Comments.

[107] Rabin, M. (1989). Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348.

[108] Reiter, M. K. (1994). Secure agreement protocols: Reliable and atomic group multicast in rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, pages 68–80.

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

[110] Rowstron, A. (2000). Optimising the Linda *in* primitive: Understanding tupla-space run-times. In *Proceedings of the 15th ACM Symposium on Applied Computing - SAC 2000*, pages 227–232.

[111] Rowstron, A. (2003). Using mobile code to provide fault tolerance in tuple space based coordination languages. *Science of Computer Programming*, 46(1–2):137–162.

[112] Rowstron, A. and Wood, A. (1997). BONITA: A set of tuple space primitives for distributed coordination. In *Proceedings of the 30th IEEE Annual Hawaii International Conference on System Sciences*, pages 379–388.

[113] Saltzer, J. H. and Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308.

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

[115] Schneider, F. B. and Zhou, L. (2005). Implementing trustworthy services using replicate state machines. *IEEE Security & Privacy*, 3(5):34–43.

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

[117] Segall, E. J. (1995). Resilient distributed objects: Basic results and applications to shared spaces. In *Proceedings of the 7th Symposium on Parallel and Distributed Processing - SPDP'95*, pages 320–327.

[118] Shamir, A. (1979). How to share a secret. *Communications of ACM*, 22(11):612–613.

[119] Sousa, P., Neves, N. F., and Veríssimo, P. (2005). How resilient are distributed *f* fault/intrusion-tolerant systems? In *Proceedings of Dependable Systems and Networks – DSN 05*, pages 98–107.

[120] Sun Microsystems (2003). JavaSpaces service specification. Especificação disponível em *http://www.jini.org/standards*.

[121] Toby J. Lehman et al. (2001). Hitting the distributed computing sweet spot with TSpaces. *Computer Networks*, 35(4):457–472.

[122] Tolone, W., Ahn, G.-J., Pai, T., and Hong, S.-P. (2005). Access control in collaborative systems. *ACM Computing Surveys*, 37(1):29–41.

[123] Tsudik, G. (1992). Message authentication with one-way hash functions. *ACM Computer Communications Review*, 22(5):29–38.

[124] Urbán, P., Défago, X., and Schiper, A. (2002). Neko: A single environment to simulate and prototype distributed algorithms. *Journal of Information Science and Engineering*, 18(6):981–997.

[125] Veríssimo, P. and Lemos, R. (1989). Confiança no funcionamento: Proposta para uma terminologia em português. Publicação conjunta INESC e LCMI/UFSC.

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

[127] Vitek, J., Bryce, C., and Oriol, M. (2003). Coordination processes with Secure Spaces. *Science of Computer Programming*, 46(1-2):163–193.

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

[129] Zhou, L., Schneider, F., and Van Rennesse, R. (2002). COCA: A secure distributed online certification authority. *ACM Transactions Computer Systems*, 20(4):329–368.

[130] Zielinski, P. (2004). Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK.