Aldelir Fernando Luiz

Arquitetura para Replicação de Serviços Tolerantes a Faltas Bizantinas Baseada em Espaço de Tuplas

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para a obtenção do título de Mestre em Informática.

Curitiba PR
Fevereiro - 2009

Aldelir Fernando Luiz

Arquitetura para Replicação de Serviços Tolerantes a Faltas Bizantinas Baseada em Espaço de Tuplas

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica do Paraná como requisito parcial para a obtenção do título de Mestre em Informática.

Área de concentração: **Ciência da Computação**

Orientador: Luiz Augusto de Paula Lima Jr., Dr
Co-orientador: Lau Cheuk Lung, Dr

Curitiba PR
Fevereiro - 2009

Pontifícia Universidade Católica do Paraná
Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Informática

ATA DE DEFESA DE DISSERTAÇÃO DE MESTRADO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DEFESA DE DISSERTAÇÃO Nº 02/2009

Aos 20 dias do mês de fevereiro de 2009 realizou-se a sessão pública de Defesa da Dissertação **"Arquitetura para Replicação de Serviços Tolerantes a Faltas Bizantinas Baseadas em Espaço de Tuplas"**, apresentada pelo aluno **Aldelir Fernando Luis** como requisito parcial para a obtenção do título de Mestre em Informática, perante uma Banca Examinadora composta pelos seguintes membros:

Prof. Dr. Luiz Augusto de Paula Lima Jr PUCPR (Orientador)	(assinatura)	aprovado (aprov/reprov.)
Prof. Dr. Lau Cheuk Lung UFSC		Aprovado
Prof. Dr. Joni da Silva Fraga UFSC		aprovado
Prof. Dr. Luiz Nacamura Junior UTFPR		Aprovado

Conforme as normas regimentais do PPGIa e da PUCPR, o trabalho apresentado foi considerado aprovado (*aprovado/reprovado*), segundo avaliação da maioria dos membros desta Banca Examinadora. Este resultado está condicionado ao cumprimento integral das solicitações da Banca Examinadora registradas no Livro de Defesas do programa.

Prof. Dr. Mauro Sérgio Pereira Fonseca
Diretor do Programa de Pós-Graduação em Informática

Programa de Pós-Graduação em Informática
PPGIa
PUCPR

Rua Imaculada Conceição, 1155 Prado Velho CEP 80215 901 Curitiba Paraná Brasil
Tel.: (41) 3271 1669 Fax: (41) 3271 2121 www.ppgia.pucpr.br

Dados da Catalogação na Publicação
Pontifícia Universidade Católica do Paraná
Sistema Integrado de Bibliotecas – SIBI/PUCPR
Biblioteca Central

L953a
2009

Luiz, Aldelir Fernando
Arquitetura para replicação de serviços tolerantes a faltas bizantinas baseada em espaço de tuplas / Aldelir Fernando Luiz ; orientador, Luiz Augusto de Paula Lima Jr. ; co-orientador, Lau Cheuk Lung. – 2009.
xvi, 115 f. : il. ; 30 cm

Dissertação (mestrado) – Pontifícia Universidade Católica do Paraná, Curitiba, 2009
Bibliografia: f. 107-115

1. Computação. 2. Algoritmos. 3. Sistemas operacionais distribuídos (Computadores). I. Lima Júnior, Luiz Augusto de Paula. II. Lau, Cheuk Lung. III. Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática. IV. Título.

CDD 20. ed. – 001.64

À Lilian e à toda minha família, pelo constante incentivo.

Agradecimentos

Em primeiro lugar quero agradecer a DEUS, que com sua onipotência me concedeu a graça de vencer mais este importante desafio da vida, tenho a plena certeza de que sem suas ricas bênçãos eu não chegaria a lugar algum.

Agradeço especialmente a minha esposa Lilian pelo apoio nas horas mais difíceis e pelos incentivos (e puxões de orelha) mesmo nas horas onde força e inspiração pareciam me faltar, e a toda minha família (meus pais Juvenal e Vera, meus irmãos André, Fabiana e Adriana) pelo constante apoio que tenho recebido ao longo de minha vida.

Também quero fazer um agradecimento muito especial aos meus orientadores Professores Lau Cheuk Lung, Alysson Neves Bessani e Luiz Lima Jr., pelas diversas e preciosas contribuições que engrandeceram significativamente este trabalho, e pelo fato deles sempre estarem disponíveis mesmos nos momentos mais inoportunos (finais de semana e madrugadas pelo MSN), quando poderiam estar realizando outras atividades.

Não posso deixar de fazer uma menção especial à empresa TOTVS S/A que, além do apoio financeiro me possibilitou a realização deste curso através de horários flexíveis, permitindo que eu concluísse com êxito todas as etapas do mesmo.

Agradeço também a todos os professores de funcionários do PPGIa pelo elevado nível do curso, e em especial a secretária Cheila por sua prontidão e boa vontade em todos os momento em que precisei de sua ajuda.

Por fim, deixo meus sinceros agradecimentos a todos que contribuíram direta e indiretamente para a realização deste trabalho.

Sumário

Resumo **xv**

Abstract **xvi**

1 Introdução **1**

1.1 Motivação 1

1.2 Objetivos da Dissertação 3

1.3 Estrutura da Dissertação 4

2 Conceitos Básicos em Sistemas Distribuídos **6**

2.1 Modelo de Sistema 7

2.1.1 Processos 7

2.1.2 Relógios 8

2.1.3 Modelos de Falhas 9

2.1.4 Modelos de Sincronismo 11

2.1.5 Canais de Comunicação 14

2.2 Acordo em Sistemas Distribuídos 16

2.2.1 Consenso 16

2.2.2 Acordo Bizantino 19

2.2.3 Detecção de Falhas em Sistemas Distribuídos 21

2.3 Conclusões do Capítulo 25

3 Fundamentos de Replicação em Sistemas Distribuídos **27**

3.1 Técnicas de Replicação em Sistemas Distribuídos 28

3.1.1 A Abordagem de Replicação Primária-Backup 29

3.1.2 A Abordagem de Replicação Máquina de Estados 31

3.1.3 As Abordagens de Replicação Semi-Ativa e Semi-Passiva 33
3.1.3.1 Replicação Semi-Ativa 34
3.1.3.2 Replicação Semi-Passiva 34
3.1.3.3 Consideração sobre Replicação Semi-Ativa e Replicação Semi-Passiva 35
3.2 Técnicas de Replicação e o Modelo de Faltas Bizantinas 36
3.2.1 Protocolos para Replicação Tolerante a Faltas Bizantinas 36
3.2.1.1 PBFT - *Practical Byzantine Fault Tolerance* 37
3.2.1.2 Arquitetura de Separação do Acordo e Execução 41
3.2.1.3 Zyzzyva - *Speculative Byzantine Fault Tolerance* 45
3.2.1.4 A2M - *Attested Append-Only Memory* 50
3.3 Conclusões do Capítulo 52

4 Fundamentos em Espaço de Tuplas 54
4.1 Modelo de Coordenação Generativa 56
4.2 Espaço de Tuplas Aumentado e Protegido por Políticas 59
4.3 DepSpace: Uma Implementação do PEATS 62
4.4 Conclusões do Capítulo 65

5 RePEATS - Uma Arquitetura para Replicação de Serviços Tolerante a Faltas Bizantinas em Espaço de Tuplas 67
5.1 Contextualização e Motivação 68
5.2 RePEATS - *Replication Over Policy-Enforced Augmented Tuple Space* 69
5.2.1 Visão Geral do RePEATS 69
5.2.2 Propriedades 71
5.2.3 Controle de Acesso 71
5.2.4 Fila de Mensagens e Ordenação de Requisições 72
5.2.5 *Checkpointing* e Transferência de Estados 77
5.2.6 Mecanismo de *Logging* e Recuperação de Mensagens 81
5.2.7 Mecanismo de Coleta de Lixo 82
5.2.8 Correção do RePEATS 83
5.3 Conclusões do Capítulo 87

6 Aspectos de Implementação e Resultados **89**

6.1 Arquitetura e Protótipo de Implementação 89

6.1.1 Ambiente de Execução 91

6.1.2 Avaliação de Desempenho 92

6.1.2.1 Execuções Normais 94

6.1.2.2 Execuções Com Faltas/Falhas 97

6.2 Estudo de Caso: Implementação de um Sistema de Arquivos NFS Tolerante a Faltas Bizantinas 98

6.2.1 Avaliação de Desempenho 99

6.3 Conclusões do Capítulo 104

7 Conclusões e Perspectivas Futuras **105**

Lista de Figuras

2.1 Modelos de Falhas/Faltas 10
2.2 Interação em Sistemas Distribuídos [22] 14
2.3 Impossibilidade de Acordo Bizantino com $n = 3$ e $f = 1$ 20
2.4 Acordo Bizantino com $n \geq 3f + 1$ 21
2.5 Relação entre os modelos de falhas [52] 25

3.1 Replicação Passiva (primária-backup) 30
3.2 Replicação Máquina de Estados 32
3.3 Replicação Semi-Ativa 34
3.4 Replicação Semi-Passiva 35
3.5 Execução do PBFT 38
3.6 Arquiteturas de Replicação 42
3.7 Comparação de Protocolos de Replicação 46
3.8 Zyzzyva com Faltas 49
3.9 PBFT equipado com o A2M 51

4.1 Interação de Processos no Espaço de Tuplas 58
4.2 Política de acesso do PEATS usado no algoritmo 1 62
4.3 Interface do DepSpace 65

5.1 Modelo de execução do RePEATS 70
5.2 Política de acesso para o PEATS usado no algoritmo 2. 77
5.3 Política de acesso para o PEATS usado no algoritmo 3. 81

6.1 Arquitetura do Protótipo 91
6.2 *Throughput* das operações para os sistemas avaliados 95
6.3 Latência das operações para os sistemas avaliados 96

6.4 Latência das operações com falhas nos sistemas ($f = 1$) 97
6.5 *Throughput* das operações com falhas nos sistemas ($f = 1$) 98
6.6 Classe e Método de Acesso ao NFS 100
6.7 Latência de operações sobre objetos nos serviços NFS comparados: NFS nativo (API *java.io* + cliente NFS Linux), RePEATS, Arquitetura de Separação, Zyzzyva e PBFT. O servidor NFS usado em todos os experimentos é o jnfs. . . 102
6.8 Desempenho do RePEATS com diversos serviços em um único Espaço de Tuplas. 103

Lista de Tabelas

4.1 Classificação dos Modelos de Coordenação [14] 55

6.1 Comparação entre Algoritmos de Replicação BFT 103

Lista de Algoritmos

1 Consenso Binário entre N processos (processo p_i). 61
2 Algoritmo de Ordenação de Requisições (cliente p_i e servidor s_i). 75
3 Algoritmo de *Checkpoint* (processo servidor s_i) 79
4 Algoritmo de computação da latência 93
5 Algoritmo de computação do *throughput* 94

Lista de Abreviações

ACL	Access Control List
API	Application Programming Interface
BFT	Byzantine Fault Tolerance
FLP	Impossibilidade de acordo em sistemas assíncronos (Fischer, Lynch e Paterson)
MAC	Message Authentication Code
PBFT	Practical Byzantine Fault Tolerance
PEATS	Policy Enforced Augmented Tuple Space
RePEATS	Replication over Policy Enforced Augmented Tuple Space
RME	Replicação Máquina de Estados
TCB	Trusted Computing Base
TFB	Tolerância a Faltas Bizantinas
TI	Tolerância a Intrusões

Resumo

Diversas pesquisas para desenvolvimento de soluções práticas de suporte a aplicações distribuídas tolerantes a faltas bizantinas têm sido realizadas nos últimos anos. Na atualidade, um dos grandes desafios da computação distribuída consiste na especificação e proposição de modelos e protocolos capazes de prover robustez, confiabilidade e segurança para as aplicações distribuídas. Neste contexto, uma abordagem bastante atrativa é a replicação de componentes de software (Replicação Máquina de Estados), que em conjunto com mecanismos de segurança permite construir serviços tolerantes ao tipo de faltas mais severa, tal como a bizantina. Apesar das técnicas de replicação ser introduzidas há pelo menos vinte e cinco anos, na última década elas tiveram um grande destaque, o qual pôde ser observado com o surgimento de diversas soluções para Replicação Máquina de Estados (replicação ativa) tolerante a faltas bizantinas com foco à viabilidade prática, com a proposição de soluções para a construção de protocolos de replicação mais otimizados e robustos. Neste trabalho propomos o uso de uma abstração de memória compartilhada (um espaço de tuplas) como suporte de comunicação e coordenação para serviços replicados. Mais precisamente, consideramos o paradigma de coordenação na especificação de sistemas confiáveis, por meio de uma arquitetura para replicação de serviços baseada em espaço de tuplas. Este espaço é usado para o acordo/ordenação de mensagens e para agregar persistência ao esquema de replicação. A proposta é baseada no princípio da "separação das entidades de acordo das de execução", no entanto, tendo como camada de acordo um espaço de tuplas confiável e seguro, além de empregá-lo para a definição de um mecanismo de *logging* de mensagens persistente, e de um mecanismo de armazenamento estável de *checkpoints* compartilhados, visando permitir que as réplicas tenham seu estado reduzido e seu processo de recuperação mais eficiente. Além do mais, o espaço de tuplas nos permite concretizar o princípio da "separação", com a vantagem de termos algoritmos muito mais simples e modulares, com algo mais poderoso do que uma simples camada de acordo (uma "fila replicada"). Por fim, nossa proposta é $2f + 1$ resistente a faltas bizantinas e atende aos requisitos de confiabilidade, disponibilidade, integridade e certo nível de confidencialidade (provido pelo espaço de tuplas), mesmo que f réplicas falhem. A implementação da proposta em sistemas reais tem bons resultados quando comparado com os trabalhos prévios da literatura (sem o uso de um espaço de tuplas).

Palavras-chave: Replicação Máquina de Estados, Espaço de Tuplas, Tolerância a Faltas Bizantinas.

Abstract

Some researches aiming to design practical algorithms to support Byzantine Fault-Tolerant distributed applications has been made in recent years. These solutions are designed to make the applications resistant to successful attacks against the system, thereby making services intrusion tolerant. We know that one of the greatest challenge of the distributed computing is to build models and protocols that achieve robustness, reliability and safety. An approach that has been exploited to achieve that is to employ replication abstractions that in addiction with security protocols provide byzantine fault tolerant guarantees to the services. Despite the replication techniques had been studied twenty five years old, nowadays it's great relevance because many proposes that presents some optimizations on replication protocols. In this work we present an architecture for byzantine fault tolerant state machine replication that uses a shared memory abstraction called tuple space to solve distributed agreement for total order broadcast and to aggregate persistence to the replication scheme. Our architecture is based on "separating agreement from execution" model, but we use a dependable tuple space to coordinate replicas and to build persistent logging service and a stable and reliable storage system to store global checkpoints for efficient recovery process (when replicas needs recovery). The tuple space coordination service allows us to separate the agreement entities from the execution entities with an advantage to have simple algorithms and a powerful abstraction rather than a simple agreement layer (emulated by a replication queue). The integration of the generative coordination model and replication state machine scheme proposed by this work allows that the systems provides reliability, integrity and confidentiality (provided by the tuple space) even that a subset of replicas are controlled by adversaries. We show that a system implemented by our architecture incur to practical result when we compare it with previous works that do the same, but without use of a tuple space.

Keywords: State Machine Replication, Tuple Space, Byzantine Fault Tolerance.

Capítulo 1

Introdução

1.1 Motivação

Soluções para o desenvolvimento de aplicações distribuídas tolerantes a faltas têm sido investigadas por mais de vinte e cinco anos [1, 2]. No entanto, esta área de investigação foi maior evidenciada na última década, por meio de trabalhos que produziram soluções com viabilidade prática para Replicação Máquina de Estado (RME) tendo foco não somente em modelos de **faltas benignas**, mas principalmente em **faltas bizantinas** [3, 4, 5, 6, 7]. Estas soluções tem como premissa tornar as aplicações resistentes a ataques maliciosos que acabam por ser bem sucedidos contra o sistema (também chamados intrusões), permitindo especificar e construir serviços **tolerantes a intrusões** [8, 9]. Além do mais, algumas destas pesquisas têm demonstrado a viabilidade de se utilizar estas soluções para aumentar a confiabilidade de serviços reais [3, 4, 5].

Os algoritmos para BFT (*Byzantine Fault Tolerance*) visam viabilizar a implementação de serviços capazes de atender aos requisitos de confiabilidade, integridade e disponibilidade através da adaptação de técnicas de redundância em nível de aplicação e de ambiente de execução. Estes requisitos são fundamentais quando se deseja obter a **segurança de funcionamento** (*dependability*) [10] para a preservar a fidedignidade do sistema de computação. Nos últimos tempos, vários trabalhos com propostas para BFT têm apresentado soluções elegantes para, por exemplo, reduzir a complexidade de mensagens, obterem o acordo em poucos passos de comunicação [6], e também para prover otimizações em termos de recursos computacionais [5]. Além do mais, algumas soluções também procuram circunscrever

restrições teóricas (ex. impossibilidade FLP [11] e acordo com pelo menos $3f + 1$ [12]), por meio da admissão de componentes seguros no modelo de sistema [7, 13] e do relaxamento das premissas de sincronismo.

Contudo, as características dinâmicas dos ambientes de comunicação e computação, principalmente em sistemas distribuídos de larga escala, introduzem alguns problemas que prejudicam a segurança e a integridade das aplicações. Tornando assim, o desenvolvimento de mecanismos para RME para esses sistemas muito mais complexos. Por outro lado, a comunidade vem investigando modelos de programação alternativos [14], que permitam e/ou facilitem o desenvolvimento de aplicações complexas, capazes de se adaptar às condições adversas destes tipos de ambientes. O paradigma de **coordenação** [15] tem como premissa a separação das atividades dos sistemas em coordenação e computação. Neste contexto, o **modelo de coordenação generativa** (ou coordenação por espaço de tuplas) [16] tem se mostrado como uma alternativa ao modelo de comunicação por passagem de mensagens devido ao seu poder e simplicidade: poucas e simples operações fornecem um modelo de comunicação desacoplado tanto no tempo quanto no espaço [14], i.e., os processos comunicantes não precisam estar ativos ao mesmo tempo nem tampouco conhecer os endereços uns dos outros para se comunicarem.

É sabido que uma das maiores dificuldades em se implementar e/ou adaptar mecanismos de RME tolerante a faltas bizantinas está no suporte que implementa os protocolos de acordo responsáveis por definir, para os servidores replicados, uma ordem atômica de execução das requisições clientes. Usualmente, as abordagens existentes envolvem um alto custo de *hardware* e *software*, e impõe restrição de que é necessário pelo menos $3f + 1$ réplicas para faltas bizantinas [12]. Porém, uma otimização recém introduzida por [5] propõe um modelo em camadas visando reduzir esse custo para $2f + 1$ réplicas de aplicação. A abordagem introduzida por [5] consiste basicamente na **separação** das réplicas em dois conjuntos, onde um destes contendo $3f + 1$ réplicas constitui uma camada de acordo (que implementa o protocolo de difusão com ordem total tolerante a faltas bizantinas), e o outro com $2f + 1$ réplicas constitui uma camada de execução de requisições (camada de aplicação), e assim, utiliza-se um conjunto de servidores para estabelecer a ordem das mensagens (executando os protocolos de acordo e, portanto requerendo $3f + 1$ servidores) e outro conjunto para executar as requisições clientes (requerendo apenas $2f + 1$ servidores).

1.2 Objetivos da Dissertação

O objetivo deste trabalho é investigar um modelo que facilite a implementação de técnicas de replicação em sistemas distribuídos. Diferentes dos trabalhos já propostos na literatura, nossa proposta não é baseada no paradigma de passagem de mensagens, mas sim em uma memória compartilhada distribuída. Diante deste fato, pretende-se usar de forma extensiva uma abstração de memória compartilhada, como suporte de comunicação e coordenação para a implementação de técnicas de replicação com faltas bizantinas. A abstração de memória compartilha a ser explorada é um espaço de tuplas, que será usado para a implementação de diversos componentes de uma arquitetura para replicação de serviços tolerantes a faltas bizantinas, nos mesmos moldes do trabalho proposto em [5].

Também faz parte de nossos objetivos, verificar a viabilidade de uso de uma abstração de memória compartilhada confiável e segura, na implementação de serviços replicados, visando a provisão de disponibilidade, integridade e confiabilidade dos respectivos serviços. Considerando o trabalho de [5], nossa proposta irá explorar o uso do espaço de tuplas como camada/serviço de acordo do esquema modularizado de replicação. Além de que também serão avaliados os custos computacionais envolvidos no trabalho proposto, tomando como base outros protocolos e modelos e arquiteturas de replicação com faltas bizantinas já consolidadas na literatura.

Tomando em conta a definição da arquitetura, o objetivo é a formalização e concretização de um suporte de replicação que tendo como base o modelo de coordenação generativa [16], mais especificamente a extensão PEATS (*Policy-Enforced Augmented Tuple Space*) [17, 18].

Com base nos objetivos gerais, alguns objetivos mais específicos são:

1. Levantamento bibliográfico no intuito de formar um arcabouço teórico com conceitos, premissas e requisitos necessários para permitir o emprego da abstração de espaço de tuplas na implementação de serviços replicados;

2. Propor uma arquitetura para RME baseada em espaço de tuplas, que ofereça modularidade na implementação de sistemas tolerante a faltas bizantinas. Em complemento, o propósito é alcançar a redução do custo de replicação para $2f + 1$. A concretização desta arquitetura será denominada RePEATS (*Replication over Policy-Enforced Augmented Tuple Space*);

3. Formalização e construção de um conjunto de protocolos para a arquitetura RePEATS, tais como protocolo replicação, protocolo de difusão e ordenação de mensagens, protocolo de sincronização e manutenção de réplicas (*checkpointing*) e mecanismos adicionais de *logging* estável e coleta de lixo, todos a serem implementados sobre o espaço de tuplas, admitindo-o como elemento de comunicação e coordenação;

4. Avaliação da eficiência do RePEATS a partir da implementação de um protótipo, e utilização deste para replicar serviços em um ambiente real. Também se pretende avaliar a eficiência do RePEATS a partir de comparações com outros protocolos de RME.

1.3 Estrutura da Dissertação

Esta dissertação está organizada de acordo com as etapas realizadas para a concretização deste trabalho. O trabalho é composto por 6 capítulos, incluindo esta introdução. O restante da dissertação está organizado conforme se descreve a seguir.

Capítulo 2: Conceitos Básicos em Sistemas Distribuídos

Neste capítulo apresentam-se os conceitos preliminares (e fundamentais) de sistemas distribuídos, que constitui parte do arcabouço teórico desta dissertação. Além disso, também é apresentado o modelo de sistema (e sub-modelos) admitido no contexto do projeto e desenvolvimento do trabalho. Os conceitos básicos apresentados concernem aos problemas relacionados ao tema tolerância a faltas bizantinas, que é a ênfase deste trabalho.

Capítulo 3: Fundamentos de Replicação em Sistemas Distribuídos

Neste capítulo são descritos todos os conceitos e o **estado da arte** no que concernem à abstração de **Replicação Máquina de Estados** e sua aplicação no modelo de faltas bizantinas. Também são apresentados os principais protocolos de replicação apresentados na literatura, que podem ser vistos como trabalhos correlatos à nossa proposta.

Capítulo 4: Fundamentos em Espaço de Tuplas

Este capítulo descreve os conceitos sobre coordenação em sistemas distribuídos e sua aplicação no contexto do desenvolvimento de sistemas tolerantes a faltas. Em primeira instância é apresentado o conceito de coordenação, com ênfase para o modelo de coordenação generativa e sua extensão ao modelo PEATS, onde se evidencia os pontos relevantes correlacionados ao PEATS e ao modelo de faltas bizantinas. Também apresentada uma implementação do modelo PEATS, denominada DepSpace.

Capítulo 5: Uma Arquitetura para Replicação Tolerante a Faltas Bizantinas em Espaço de Tuplas

Neste capítulo é apresentado o RePEATS, que consiste na arquitetura resultante da integração da abstração de replicação máquina de estados com o modelo de coordenação generativa, fruto desta dissertação. Inicialmente discute-se sobre as vantagens do emprego de um espaço de tuplas em uma arquitetura de replicação de serviços. Dando seqüencia são apresentados os algoritmos formais que dão suporte a arquitetura, bem como as respectivas provas de correção destes algoritmos.

Capítulo 6: Aspectos de Implementação e Resultados

Este capítulo aborda os aspectos técnicos relacionados ao desenvolvimento do protótipo, sua implementação e a avaliação dos resultados obtidos, além de comparações com os principais protocolos de replicação da literatura. Por fim é apresentado um estudo de caso que serve como prova de conceito e atesta a viabilidade de uso da proposta para a concepção de sistemas reais.

Capítulo 7: Conclusões e Perspectivas Futuras

Finalmente neste capítulo são apresentadas as conclusões finais sobre o projeto, bem como alguns pontos que possibilitam a continuação deste trabalho em projetos futuros.

Capítulo 2

Conceitos Básicos em Sistemas Distribuídos

Informalmente, um sistema distribuído por ser visto como uma coleção de entidades independentes e espacialmente dispersas, que cooperam entre si, no que tange a resolução de problemas [19]. Em outros termos, um sistema distribuído é definido como um sistema informático que contém diversos elementos de computação (processos, processadores, máquinas) interconectados por meio de um sistema subjacente de comunicação (ex. uma rede de comunicação) [20, 21]. Neste sistema os elementos se comunicam uns com os outros por meio de mensagens que são enviadas e recebidas através da rede de comunicação. Estes elementos que formam o sistema distribuído realizam ações locais e cooperam entre si (através de trocas de mensagens), para a realização das tarefas globais [22]. No entanto, um ponto de crucial importância para a concepção de sistemas distribuídos é a sincronização, pois, uma vez que os elementos são independentes e realizam computações locais, a sincronização se faz necessária para que as ações/atividades do sistema distribuído sejam precisamente coordenadas (ex. alocação de recursos, confirmação ou retrocesso de uma ação).

A construção de aplicações distribuídas é motivada por diversos fatores, dentre os quais se podem destacar principalmente o compartilhamento de recursos, o aumento da capacidade de processamento, a cooperação de serviços e a melhoria no aspecto de confiabilidade (se o sistema for especificado com mecanismos adequados). No entanto, a construção de sistemas distribuídos abre precedentes para uma série de desafios, quando verificamos que na atualidade há uma grande heterogeneidade dos elementos/componentes que podem pertencer a este tipo

de sistema (sistemas operacionais, tecnologias de redes, linguagens de programação etc). Em face destes fatos, é cada vez mais desejável que o sistema distribuído seja expansível, seguro e tolerante a faltas, mesmo na presença das adversidades oriundas da heterogeneidade dos ambientes de computação.

O objetivo deste capítulo é apresentar os conceitos fundamentais de sistemas distribuídos que serão abordados ao longo do trabalho. Dentro deste contexto serão apresentados o modelo de sistema no qual está amparado o trabalho, os conceitos preliminares em tolerância a faltas e os principais problemas encontrados na concepção de sistemas distribuídos ao longo dos anos. Este capítulo também apresenta uma breve descrição das premissas fundamentais para a implementação de sistemas tolerantes a faltas bizantinas.

2.1 Modelo de Sistema

Neste trabalho referenciamos "sistema" como sendo o ambiente de execução, e "modelo" como sendo as propriedades que se pode inferir no sistema. Assim, um modelo de sistema pode ser visto como a descrição formal do comportamento e das propriedades que concernem a um determinado projeto de sistema. Em suma, o modelo de sistema define as premissas que dão sustentação aos algoritmos, fundamentam suas características e permite (baseado em hipóteses) alcançar as garantias que o sistema (e seus algoritmos) propõe. Em se tratando de sistemas distribuídos, um conjunto de sub-modelos podem ser encapsulados dentro do modelo de sistema, onde cada sub-modelo define as peculiaridades de cada componente do sistema (ex. processos, tipos de falhas etc.).

Nesta seção são apresentados os modelos que fundamentam as propriedades do sistema proposto, sendo evidenciadas as principais suposições/hipóteses admitidas no contexto do trabalho, bem como as garantias que o sistema pode fornecer.

2.1.1 Processos

No contexto de sistemas distribuídos, um processo consiste em um elemento lógico que é apto a realizar as computações do sistema [23], e este elemento por sua vez, reflete a execução de um algoritmo em um determinado processador (também conhecido como um nó

do sistema) [24, 25]. Formalmente, cada processo é modelado como um autômato (uma para cada processo) que é composto por um conjunto de estados, que vão evoluindo na medida em que o processo computa/executa os eventos (ou passos), que são disparados por meio de estímulos (recepção/envio de mensagem). O estado global do sistema distribuído é composto pelo estado local de cada um dos processos e pelo estado dos canais de comunicação [26]. Estes canais de comunicação (também conhecidos como *links*[23]) são responsáveis por estabelecer o vínculo entre os processos, para que seja possível realizar as comunicações. A inicialização de um sistema distribuído ocorre quando os processos se encontram em seus estados iniciais (arbitrários) e os canais estão vazios [25].

Neste trabalho consideramos um conjunto contendo um número arbitrário de processos "deterministas", divididos em dois subconjuntos: $\Pi = \{p_1, p_2, ..., p_n\}$ com um número arbitrário de clientes e $U = \{s_1, s_2, ..., s_n\}$ com n servidores que implementam as réplicas de um serviço. Tendo em vista a possibilidade de se ter múltiplos serviços replicados partilhando a mesma infra-estrutura de coordenação, o conjunto de processos U pode conter diversos subconjuntos, e neste caso pode ser representado por $\{U_1, U_2, ..., U_n\} \subset U$.

2.1.2 Relógios

Um aspecto bastante interessante (e importante) em sistemas distribuídos, é que nestes é praticamente impossível alcançar uma sincronização de relógios, pois, como os elementos se comunicam por meio de passagens de mensagens, atrasos inconstantes podem vir a ocorrer durante as comunicações. Por este motivo, torna-se inviável o uso de relógios físicos e/ou algum mecanismo referente ao tempo para fins de sincronização de atividades entre os elementos. A primeira solução para este problema foi proposta por Lamport [1], que introduziu a noção de causalidade [1, 27] (ou relação de precedência causal), elucidando que, apesar dos problemas e fatos contrários à sincronização, é possível haver a existência de uma relação causal entre os eventos emitidos pelos elementos de um sistema distribuído. Assim, no intuito de prover uma ordenação parcial da seqüencia de eventos do sistema, Lamport definiu o conceito de "relógios lógicos" (ou relógios de Lamport) [1], modelo este onde cada processo é equipado com um relógio lógico que pode ser incrementado por meio aplicação de um conjunto de regras. Este relógio definido por Lamport não marca horas, mas sim uma seqüencia de valores, onde, a cada evento (envio ou recepção de mensagens, processamentos internos) é associado uma

estampilha de tempo (com o valor correspondente ao evento), por meio da qual o processo pode inferir a seqüencia (relação causal) dos eventos. Mais precisamente, a unidade de tempo considerada é o conjunto dos números inteiros positivos. O avanço/incremento do relógio lógico, precede a execução de um evento, acrescentando-se à base de tempo anterior o número que representa algum intervalo (geralmente uma unidade). No caso de um evento de recepção de nova mensagem, o relógio é incrementado, no entanto tomando como base de tempo o valor da estampilha da mensagem recebida, caso esse valor for maior do que o atual. A partir do uso deste método é possível obter uma ordenação total dos eventos em cada processo.

No contexto deste trabalho admitimos que todos os processos têm relógios locais, porém, estes relógios não são sincronizados, mas têm como única premissa o progresso.

2.1.3 Modelos de Falhas

Assim como qualquer outro sistema de computação, um sistema distribuído também é passível de falhas. Neste caso, se considerarmos todos os elementos necessários para a execução de um sistema distribuído, as falhas podem ser oriundas de defeitos de *software* ou *hardware*, de erros de projeto, ou ainda por motivos alheios (ex. queima de componente eletrônico, incêndio, invasões etc.). É importante evidenciar que se o sistema não estiver preparado para lidar com a ocorrência destes fenômenos, a execução correta do sistema pode ser comprometida. Um sistema é considerado **falho** (ou **faltoso**) quando o seu comportamento é desviado do requerido pela sua especificação [21], de outra forma o sistema é considerado **correto**.

A definição de um "modelo de falhas" (e as respectivas semânticas associadas) requer o conhecimento das anomalias a que se sujeita o funcionamento de um sistema de computação. Estas anomalias conhecidas na literatura como falta, erro e falha, são de fundamental compreensão no contexto de modelos de falhas. Em [28] a seguinte terminologia é apresentada:

- **falta**: pode ser definida como a causa fenomenológica de um erro, que por sua vez pode ser algorítmica ou física (*hardware*);
- **erro**: é um estado errôneo do sistema que pode conduzi-lo a uma falha quando do seu processamento;

- **falha**: é definido como o desvio do comportamento do sistema da especificação requerida, é o meio pelo qual é possível observar que o sistema se desviou de seu propósito.

Os modelos de falhas (e suas respectivas semânticas) definem as formas pela qual um sistema pode desviar de sua especificação. O modelo de falhas de um sistema fornece subsídios para quantificar e qualificar os requisitos e mecanismos necessários para que o sistema possa operar de forma correta e contínua mesmo na ocorrência de faltas. As semânticas associadas aos modelos de falhas são definidas nos domínios de **tempo** e **valor** conforme [29], e são agrupadas em cinco categorias conforme ilustrado na figura 2.1.

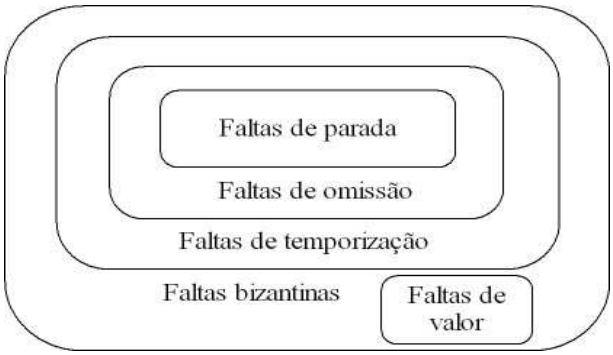


Figura 2.1: Modelos de Falhas/Faltas

Falta de Parada (ou *crash*) é o tipo de falta mais restritivo. As faltas desta natureza são caracterizadas pelo travamento ou bloqueio completo do sistema, fazendo com que ele nunca seja submetido a um novo estado (evolução).

Falta de Omissão é caracterizada pelo comportamento efêmero do sistema, no que concerne ao envio das respostas às requisições enviadas, isto é, o sistema deixa de responder a algumas requisições durante seu ciclo de vida.

Falta de Temporização este tipo de falta só é valido quando o sistema impõe restrições temporais. Assim, se o sistema responde corretamente a uma requisição, porém, fora do intervalo de tempo considerado pelo sistema, então é caracterizada uma falta desta natureza (ex. atraso na entrega das mensagens).

Falta de Valor é o oposto do modelo anterior, e ocorre quando o sistema atende as requisições dentro do intervalo de tempo especificado, porém emitindo respostas fora do valor especificado (ex. corrupção de uma cadeia de *bytes* contida na mensagem).

Falta Arbitrária (ou bizantina) é o tipo de faltas mais severo e o menos restritivo, pois encapsula todos os modelos de faltas conhecidos. A falta desta natureza é caracterizada pelo comportamento arbitrário (desvios algoritmicos maliciosos) que o sistema pode apresentar durante sua execução.

Neste trabalho admitimos que todos os processo estão sujeitos a faltas bizantinas [12]. Desta forma, os processos do sistema podem apresentar comportamento arbitrário em sua execução, podendo parar, omitir o envio e recepção de mensagens, alterar o conteúdo das mensagens, enviar mensagens espúrias, além de fazer conluio com outros processos **maliciosos** visando a corrupção do sistema. Por fim, um processo que apresenta comportamento de falha é referenciado como **faltoso**, e do contrário é referenciado como **correto**.

Assumimos que as faltas/falhas bizantinas podem ser oriundas de incidentes (ex. erros de programação, falhas de hardware) ou por meio de intrusões [9] (ataques de segurança bem sucedidos, que conduzem o sistema ao comportamento de falha).

2.1.4 Modelos de Sincronismo

A admissão de um modelo de sincronismo (ou modelo de interação [22]) adequado é fundamental para a especificação de um sistema distribuído, pois é através das premissas concernentes ao sincronismo que são conhecidos os limites de tempo necessários para a computação dos eventos no ambiente distribuído. O atributo sincronismo está intimamente relacionado tanto ao comportamento dos processos como da rede de comunicações. Na literatura são encontrados diversos modelos de sincronismo para sistemas distribuídos, e como extremos destes modelos temos os que são completamente dependentes de tempo (também conhecido como síncronos), e os que não impõem nenhuma premissa em relação ao tempo (ou assíncronos) [25, 24]. Além dos extremos, na literatura encontramos uma diversidade de modelos de sincronismo intermediários, que por sua vez relaxam alguma(s) propriedade(s) e característica(s) dos modelos extremos (são mais práticas e realistas).

Dentre os modelos intermediários, dois destes merecem destaque: **assíncrono temporizado** [30] e **parcialmente síncrono** (ou síncrono terminal) [31]. O modelo assíncrono temporizado [30] define um conjunto de propriedades no intuito de identificar comportamentos síncronos em execuções assíncronas. Para tanto, algumas premissas são admitidas, para dar sustentação ao modelo. São elas: (*i*) todos os serviços no sistema consideram em sua especificação um limite de tempo, no qual devem ser produzidas as saídas do sistema; (*ii*) a comunicação ocorre por meio de um serviço de datagrama não confiável, o qual está sujeito a faltas de desempenho; (*iii*) os processos podem falhar por parada ou desempenho; (*iv*) os processos têm acesso aos relógios físicos, que podem apresentar desvios em relação ao relógio de tempo real; e (*v*) as falhas nas instâncias permitidas podem ocorrer sem limites. Em suma, este modelo descreve de forma bastante realista os sistemas distribuídos da atualidade, pois se analisarmos as premissas admitidas com a realidade vemos que é factível a existência de *timeouts*; grande parte dos sistemas/computadores atuais são equipados com relógios; e os serviços de datagramas especificado estão disponíveis em protocolos como o UDP [32]. De outro lado, o modelo parcialmente síncrono [31] (também conhecido como eventualmente síncrono ou de sincronismo terminal) estabelece que, para todas as execuções do sistema existe um limite Δ e um instante de tempo conhecido como GST (*Global Stabilization Time*), de modo que toda mensagem enviada por um processo correto após um instante $u >$ GST é recebida antes de $u + \Delta$. Contudo, apesar da existência destes limites, eles são desconhecidos pelos processos, e nem tampouco são os mesmos em diferentes execuções do sistema. A grande questão por trás deste modelo, é que ele trabalha de forma assíncrona em grande parte do tempo, mas durante períodos de estabilidade, o tempo de transmissão de mensagens é limitado.

Em se tratando dos modelos extremos, é considerado síncrono um sistema em que todas as computações admitem restrições temporais, onde a partir das quais permite assegurar as seguintes propriedades [33]:

- Os limites mínimos e máximos no que concerne ao tempo de execução de um passo de processamento;
- O tempo máximo para o recebimento de cada mensagem transmitida sobre o canal de comunicação;

- O limite máximo de desvio em relação ao tempo real, para os relógios locais dos processos.

A grande vantagem de admitir um sistema síncrono advém do fato que nele é possível estabelecer premissas temporais, o que provê facilidades na implementação de serviços tolerantes a faltas, como é o caso dos serviços baseados em consenso distribuído, que por sua vez só são completamente solúveis em ambientes síncronos [11].

Do outro lado está o modelo assíncrono, onde não são admitidos quaisquer limites de ordem temporal, isto é, não existem limites de tempo para a ocorrência dos eventos no sistema. Este tipo sistema é mais desfavorável para a implementação de sistemas tolerantes a faltas dada a existência da impossibilidade FLP[1] [11], cujo arcabouço teórico de mais de duas décadas prova que o consenso só admite "solução determinista" no modelo síncrono, mas não no modelo assíncrono [11]. Esta insolubilidade do consenso no modelo assíncrono advém da dificuldade de distinguir quando um processo é falho ou um processo é lento. Assim, para que a os problemas distribuídos baseados em consenso sejam solúveis no modelo assíncrono é necessário que os algoritmos sejam especificados por meio da abordagem probabilista (algoritmos aleatórios) [34, 35], ou ainda, que os algoritmos sejam equipados com oráculos, também conhecidos como "detectores de falhas" [36]. Portanto, pode-se dizer que o modelo assíncrono não admite solução "determinista" para resolução de consenso, porém, é provado que a introdução de aleatoriedade permite resolver os problemas distribuídos, que de forma determinista não tem solução.

Em suma, as computações de um sistema assíncrono não admitem nenhuma hipótese relacionada a tempo, e deste modo não existe qualquer restrição no que concerne à [22]:

- Velocidade de execução dos passos de processamento pelos processos, isto é, um passo de processamento pode ser executado em um intervalo de tempo arbitrário;
- Atraso na transmissão de mensagens, isto é, uma mensagem pode ser recebida após um longo intervalo de tempo;
- Sincronização e desvio de relógios, isto é, os relógios não são sincronizados e podem apresentar desvios arbitrários.

[1] O termo FLP advém das iniciais dos autores, que são Fischer, Lynch e Paterson, respectivamente

Para a realização deste trabalho admitimos a existência de um modelo de sincronismo híbrido, havendo um ambiente de computação assíncrono (e desacoplado) fornecido pelo PEATS[2] [17] onde os processos se comunicam através do compartilhamento de informações em tuplas armazenadas em um espaço confiável e não suscetível a falhas[3]. Contudo, dadas as condições insolubilidade do consenso em sistemas puramente assíncronos [11], algumas premissas temporais se fazem necessárias tendo em vista que o substrato de espaço de tuplas (PEATS) usado é replicado através do algoritmo PAXOS Bizantino [37]. Deste modo, para assegurar a terminação do algoritmo de acordo empregado na construção do substrato de espaço de tuplas, é admitido o emprego do modelo parcialmente síncrono (síncrono terminal) [31] na camada que implementa o substrato de comunicação de baixo nível.

2.1.5 Canais de Comunicação

Uma computação distribuída ocorre através da interação entre processos e, de modo geral uma interação se dá a partir de trocas de mensagens entre os processos. Deste modo, para uma interação é requerida comunicação e coordenação (ordenação de eventos [1]) entre os processos [22]. Formalmente uma interação pode ser representada por um grafo, onde os vértices representam os processos e as arestas representam os canais de comunicação que ligam os processos uns aos outros, conforme ilustrado na figura 2.2.

Figura 2.2: Interação em Sistemas Distribuídos [22]

Na literatura, as abstrações que representam os canais de comunicação que ligam os processos são conhecidos como *links* [23]. Em sistemas distribuídos, os *links* representam

[2]Modelo de Espaço de Tuplas adotado neste trabalho

[3]Na realidade o Espaço de Tuplas pode sofrer até f falhas nas $3f + 1$ réplicas que o implementa

não somente o sistema subjacente de comunicação, mas também fornecem uma topologia de conectividade completa entre os elementos de computação do sistema. Entretanto, em sistemas onde é admitida a possibilidade de falhas (tal como as redes de larga escala) é comum ocorrência de perdas, duplicação e corrupção de mensagens. Assim, para que o sistema não tenha sua execução comprometida em decorrência destes problemas é requerido o uso de abstrações de comunicação mais fortes que permitam estabelecer limites nas perdas, ou ainda, que permitam não haver perdas. Em se tratando da perda de mensagens, duas propriedades para os canais de comunicação podem ser inferidas [38]:

- **Sem Perdas**: Se um processo p envia uma mensagem m ao processo q, e q é correto, então q terminará por receber m;
- **Perda Justa**: Se um processo p envia um número infinito de mensagens ao processo q, e q é correto, então q receberá um número infinito de mensagens enviadas por p;

É valido ressaltar que, para fins de fortalecimento do modelo, grande parte dos sistemas distribuídos consideram além destas propriedades (uma ou outra, dependendo do modelo admitido) que os canais não criam, não alteram, nem duplicam as mensagens que trafegam sobre o sistema subjacente de comunicação. O primeiro caso, isto é, canais que não admitem perdas são referenciados na literatura como canais confiáveis (*reliable channels*) [39, 40], e estes canais por sua vez, só podem ser implementados em sistemas com capacidade de armazenamento infinito para o *buffer* de mensagens, uma vez que sistemas com *buffer* finito são suscetíveis a inundação (*overflow*), o que resulta em perdas. No segundo caso, os canais que admitem perdas são definidos como *fair-lossy links* [40]. Os *fair-lossy links* são abstrações mais realistas, pois admitem perdas de mensagens de forma eqüitativa, onde geralmente são usados para modelar canais que falham temporariamente, e que podem perder mensagens devido a capacidade finita de armazenamento (entre outros fatores). Cabe salientar primitivas de difusão e recepção (*send*/*receive*) mais fortes como os canais confiáveis podem ser implementadas a partir dos *fair-lossy links*, se empregado mecanismos de reconhecimento e restransmissão [38].

Para a realização deste trabalho, admitimos que os processos que implementam os algoritmos se comunicam através de um PEATS (i.e. espaço de tuplas confiável). E, uma vez que o PEATS é confiável, assumimos que os canais de comunicação que ligam os processos ao PEATS, bem como os canais que ligam diretamente os processos (quando necessário), são

confiáveis e autenticados. Estes canais são implementados a partir de *sockets* TCP/IP juntamente com MACs [41, 42] e chaves de sessão, onde também são empregados mecanismos de retransmissão e reconexão quando apropriado. Uma outra aproximação pragmática para este tipo canal é o uso de TLS/SSL [43] em sua implementação.

2.2 Acordo em Sistemas Distribuídos

Um dos problemas que constitui a base fundamental dos sistemas distribuídos é o acordo [44]. O problema por trás do acordo distribuído está em assegurar que todos os participantes do sistema cheguem a uma decisão comum para um determinado valor, que tenha sido previamente proposto por alguma das entidades do sistema. A maioria dos sistemas distribuídos requer algum tipo de acordo para a realização das computações. Este acordo pode variar desde a simples troca de mensagens entre dois processos, onde ambos devem concordar com a seqüencia de mensagens que foram emitidas, ou até mesmo em sistemas com um grande número de processos, onde todos eles devem chegar a um comum acordo quanto à realização de uma tarefa do sistema, como por exemplo, a confirmação ou o retrocesso de uma transação.

O problema do acordo vem sendo estudado tanto em termos teóricos quanto práticos há pelo menos trinta anos [45]. Ao considerar os modelos de sistema existentes, com o passar dos anos chegou-se a conclusão de que o acordo só é solúvel em um sistema distribuído, caso sejam admitas algumas premissas no modelo, do contrário o algoritmo de acordo não tem terminação garantida [11]. No contexto de sistemas distribuídos existem diversas variações do problema de acordo, onde cada uma delas depende da natureza do problema (ou do cenário). O fato é que a essência do problema está na unanimidade e na irretratabilidade sobre a decisão dos processos envolvidos no acordo, isto é, os processos corretos devem chegar a uma decisão comum quanto à computação a ser realizada, independente da ocorrência ou da ausência de falhas.

2.2.1 Consenso

O **consenso** [45] é a generalização mais conhecida do problema de acordo em sistemas distribuídos. O problema do consenso consiste em assegurar que todos os processos corretos envolvidos em uma computação distribuída terminem por obter uma saída comum baseados nas mesmas entradas, em outros termos, todos os processos devem decidir por um mesmo valor,

desde que tal valor tenha sido previamente proposto por algum destes processos. O consenso pode ser visto como um dos problemas mais estudados em sistemas distribuídos, dada sua crucial importância teórica (fatores limitantes para a implementação do acordo em sistemas computacionais) e prática (a base algorítmica para a grande maioria dos problemas de acordo).

Formalmente, a diferença entre os problemas de acordo e consenso reside no fato de que, o acordo é uma espécie de abordagem ditatorial, isto é, um único processo do sistema tem a posse do valor inicial (que será tomado como valor de decisão) e este tem de utilizar meios que assegurar que os demais participantes tenham ciência deste valor para realizar as computações, enquanto que, no problema do consenso todos os processos têm um valor inicial e a abordagem é democrática, isto é, os processos decidem um valor, dentre os valores iniciais. É importante clarificar esta questão, uma vez que ambos os termos (acordo e consenso) são usados na literatura para referenciar um ao outro.

Mais precisamente, o problema do consenso envolve um conjunto de processos $\Pi = \{p_1, p_2, ..., p_n\}$, onde cada processo correto p_i propõe um valor $v_i \in V$, sendo que ao final do processamento todos os processos corretos devem decidir de forma irrevogável e unânime sobre o valor contido no conjunto dos valores propostos V. Como complemento é admitida a existência de duas primitivas conforme [33]:

- $propose(P, v)$: denota que o valor v é proposto a um conjunto de processos P;
- $decide(v)$: indica que o valor v é decidido.

Formalmente o problema do consenso pode ser definido a partir das seguintes propriedades [46, 44]:

- **Acordo**: se um processo correto decide por um valor v, então todos os processos corretos decidem pelo mesmo valor v;
- **Validade**: se um processo decide por um valor $v \in V$, então v foi previamente proposto por algum processo;
- **Integridade**: um processo decide no máximo uma vez;
- **Terminação**: todos os processos corretos chegam ao valor de decisão.

Ao analisar estas propriedades do problema do acordo se verifica que a primeira propriedade (acordo) captura a essência do problema, isto é, a decisão de todos os processos corretos por um mesmo valor. A segunda propriedade (validade) concerne a corretude (*safety*) do problema, onde assegura a consistência do valor decidido com algum dos valores propostos. Por fim, a terminação especifica que o algoritmo de consenso deve ter progresso (*liveness*).

Uma generalização do consenso denominada **consenso uniforme** [47] define uma propriedade ainda mais forte para o problema do consenso, pois na literatura é discutido que as propriedades definidas para o problema do acordo abrem precedentes para que os processos decidam valores diferentes caso um deles venha a falhar. Assim, além das propriedades já citadas, uma nova propriedade é definida para evitar a possibilidade de desacordo entre os processos, no caso da ocorrência de falhas [46, 47].

- **Acordo Uniforme**: Dois processos, sejam corretos ou faltosos, não decidem por valores diferentes.

Embora as propriedades vistas até então sejam consideradas suficientes para implementar a maiorias dos algoritmos de acordo (e consenso), alguns autores afirmam que em ambientes onde se admite a possibilidade de faltas bizantinas elas não têm muita utilidade prática [25], já que a partir delas é possível que o valor decidido seja um valor proposto por processos faltosos. Assim, para admitir o modelo de faltas bizantinas as propriedades do acordo devem ser modificadas da seguinte forma [25]:

- **Acordo**: todos os processos corretos decidem pelo mesmo valor;
- **Validade**: se todos os processos corretos iniciam com o mesmo valor $v \in V$, então v é o único valor de decisão possível para um processo correto.

Tomando em conta a farta literatura sobre acordo e consenso encontramos inúmeras variações para o problema do consenso [33, 48, 11], tais como consenso unário (decisão sobre um único valor proposto), consenso binário (decisão sobre dois valores propostos) e consenso multivalorado (decisão sobre um domínio de valores propostos), porém, a semântica destas variações sempre é equivalente. Uma generalização particularmente interessante do problema do consenso é o k − **set consensus** (ou k − **agreement**) [25, 49]. Este problema modifica a propriedade de **acordo** de tal forma que ao invés dos processos decidirem por um único

valor, eles podem decidir por conjunto que contém k possíveis valores de decisão. No entanto, para que isso seja possível, as propriedades de acordo e validade originais sofrem as seguintes modificações [25]:

- **Acordo**: existe um subconjunto $W \subseteq V$, onde $|W| = k$, tal que todos os possíveis valores de decisão são valores presentes em W;
- **Validade**: qualquer decisão de qualquer processo é um valor proposto por algum processo.

A generalização original do problema do consenso e o problema do consenso multivalorado são equivalentes. Isto se confirma pelo fato de que o consenso multivalorado ($k-$**set consensus**) é redutível ao consenso padrão se considerado $k = 1$ (para consenso unário) e $k = 2$ (para consenso binário).

2.2.2 Acordo Bizantino

Uma generalização do consenso de grande interesse prático e teórico é admissão da possibilidade de faltas bizantinas nos processos participantes do consenso. De forma intuitiva, para se chegar a um valor de decisão em um ambiente onde é admitida a possibilidade de faltas, poderíamos simplesmente considerar que se a maioria dos participantes é correta (não faltosa) é possível chegar ao consenso. No entanto, esta intuição não é válida se admitimos a possibilidade de **faltas bizantinas** [12] entre os participantes. Quando consideramos a possibilidade de faltas desta natureza são necessários que mais de $2/3$ dos participantes sejam corretos de um total de $3f + 1$ participantes, onde f denota o número máximo permitido de participantes faltosos.

Esta questão é devidamente provada em [12], onde os autores ilustram a realização do consenso na presença de faltas, relacionando o acordo ao problema clássico dos "generais bizantinos", e então dando origem ao nome **faltas bizantinas**. Neste trabalho é provado que se entre três participantes houver pelo menos um traidor, o acordo não é alcançado conforme ilustrado na figura 2.3 (modelo sem assinaturas).

O problema é definido da seguinte forma: em um exército existe um general e seus tenentes. O general emite uma ordem aos tenentes, ordem esta que pode ser para atacar ou

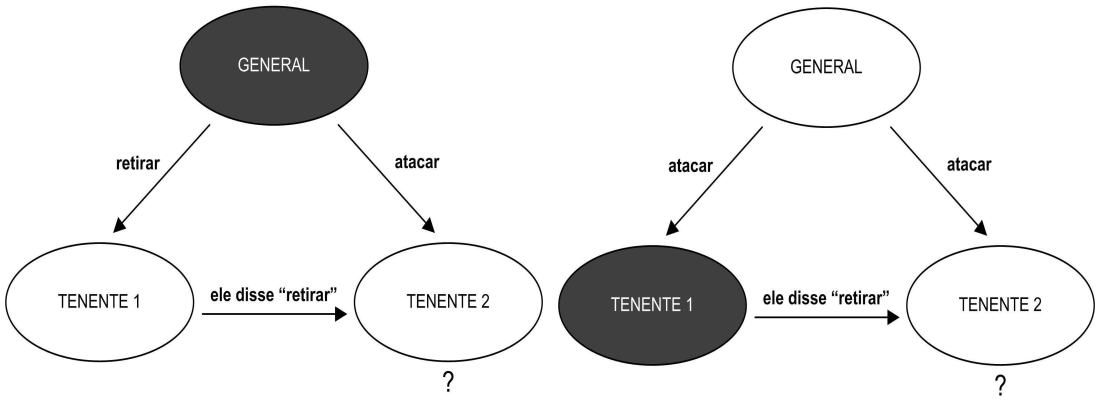


Figura 2.3: Impossibilidade de Acordo Bizantino com $n = 3$ e $f = 1$

retirar. Os tenentes trocam informações entre si para validar a ordem emitida pelo general. No primeiro caso (figura 2.3 (a)) o general é o traidor e emite ordens diferentes aos tenentes, então o tenente 1 envia uma confirmação ao tenente 2 atestando que o general deu ordem para retirar, enquanto o tenente 2 envia a confirmação ao tenente 1 atestando que o general lhe deu ordem para atacar, e deste modo é causado um colapso no ambiente pois não se sabe qual ação deve ser tomada. No segundo caso (figura 2.3 (b)), o traidor é o tenente 1 que tenta forjar a ordem emitida pelo general, e assim fazendo com que o tenente 2 fique indeciso sobre a ação a ser executada. Em ambos os casos não há como saber quem é o traidor, e assim não é possível ter progresso na execução da ordem requisitada.

O acordo bizantino é possível em um ambiente onde existem pelo menos $3f + 1$ participantes, e até f deles é malicioso conforme demonstra a figura 2.4. Esta afirmação pode ser observada a partir da segunda fase do algoritmo dos generais bizantinos, onde os tenentes trocam mensagens entre si para validar a ordem do emitida pelo general. Ao final da execução da segunda fase o tenente 1 terá o valores “a,r,a”, o tenente 2 “r,a,a” e o tenente 3 “a,r,a”, e deste modo todos irão decidir pelo valor (ou ordem) que aparece em predominância na lista de valores, neste caso “a”. Este cenário prova que o acordo bizantino é solúvel onde existe pelo menos $3f + 1$ participantes.

A impossibilidade de acordo bizantino com $n < 3f + 1$ é valida tanto em sistemas síncronos quanto em sistemas assíncronos. Em face deste fato, na literatura também é apre-

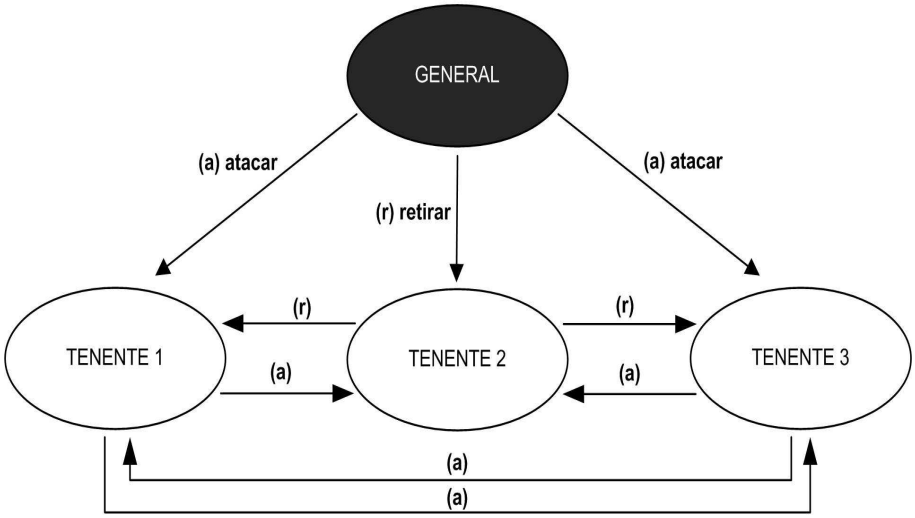


Figura 2.4: Acordo Bizantino com $n \geq 3f + 1$

sentada a possibilidade de empregar mecanismos criptográficos para assinatura/autenticação das mensagens [50]. Desta forma, em sistemas síncronos equipados com mecanismos criptográficos para a verificação da autenticidade o traidor pode ser facilmente identificado, e assim é possível construir algoritmos que toleram até f faltas, tendo pelo menos $f + 2$ participantes [12] (e, portanto sendo $< 3f + 1$).

Contudo, mesmo admitindo o emprego de ferramentas criptográficas, a impossibilidade do acordo bizantino ainda é valida em sistemas assíncronos, caso o número de participantes corretos não seja $> 2/3$ do número total de participantes [51]. Este resultado é baseado no fato de que mesmo tendo provas quanto à autenticidade da mensagem, um processo malicioso pode se prevalecer do adiamento na entrega das mensagens enviadas (já que o sistema é assíncrono) e com isso fazer com que os processos corretos decidam valores diferentes caso o número de processos corretos seja $\leq 2/3$ do total de processos.

2.2.3 Detecção de Falhas em Sistemas Distribuídos

Ao longo dos anos a crença de que o problema do consenso não pode ser solúvel no modelo assíncrono motivou os pesquisadores a investigar por alternativas que permitissem a construção de algoritmos de consenso para este modelo. Conforme citado na seção 2.2.3,

a insolubilidade do consenso no modelo assíncrono advém do simples fato de que neste modelo é praticamente impossível distinguir um processo falho de um processo lento [11]. A introdução do conceito de **detectores de falhas não-confiáveis** [36] (ou simplesmente detectores de falhas) foi uma das respostas dos pesquisadores em contestação a essa crença, tendo caracterizado os detectores de falhas como uma alternativa para resolver o problema do consenso em ambientes assíncronos [11]. Em face deste fato, um detector de falhas torna-se um componente (ou serviço) fundamental para o desenvolvimento de sistemas tolerantes a faltas, pois (em teoria) ele provê subsídios para que o poder de um sistema assíncrono equipado com um detector de falhas seja equivalente ao poder de um sistema síncrono.

Um detector de falhas $\mathcal{D}$ [36] pode ser visto como um oráculo distribuído, que periodicamente fornece uma lista contendo informações a respeito dos processos suspeitos de terem falhado. Em suma, um detector de falhas consiste em um conjunto de módulos, onde cada módulo de detecção é associado a um processo do sistema. Cada um dos módulos monitora os demais processos e mantém uma lista que contém informações a respeito dos processos que atualmente são suspeitos de terem falhado. Em outros termos, cada processo p_i mantém uma lista denominada $suspected_i$, onde o detector pode continuamente realizar manutenções nesta lista, podendo adicionar ou remover informações quando for conveniente. Contudo, é valido salientar que em um sistema equipado com detectores de falhas é possível haver divergências entre as listas dos processos, pois não existe nenhuma premissa quanto a um acordo entre os detectores.

Formalmente, um detector de falhas é definido em termos das propriedades de **completude** e **precisão** [36]. Mais precisamente, Chandra e Toueg [36] definem duas propriedades de completude e quatro de precisão:

- **Completude Forte (*Strong Completness*):** cada processo faltoso é permanentemente detectado por cada processo correto;
- **Completude Fraca (*Weak Completness*):** cada processo faltoso é permanentemente detectado por algum processo correto;
- **Precisão Forte (*Strong Acurracy*):** nenhum processo é suspeito pelo detector, caso ele não seja realmente faltoso;

- **Precisão Fraca (*Weak Acurracy*):** algum processo correto nunca é suspeito pelo detector;
- **Precisão Eventualmente Forte (*Eventual Strong Acurracy*):** após um tempo os processos corretos não são mais suspeitos por qualquer processo correto;
- **Precisão Eventualmente Fraca (*Eventual Weak Acurracy*):** após um tempo, algum processo correto nunca mais é suspeito por qualquer processo correto.

Ao verificar as propriedades nota-se que a *completude* assegura que os processos faltosos serão de alguma forma suspeitos, já a *precisão* diz respeito aos equívocos que podem ser cometidos pelo detector $\mathcal{D}$. Na literatura, Chandra e Toueg [36] apresentam uma taxonomia que leva a definição de 8 classes de detectores de falhas de forma hierárquica, tomando como base a combinação das propriedades de *completude* e *precisão* descritas. Dentre as classes de detectores, as de maior interesse são os detectores $\mathcal{P}$ (detectores perfeitos) e os detectores $\diamond\mathcal{W}$ (eventualmente fracos) por serem os extremos das classes propostas em [36]. Os detectores da classe $\mathcal{P}$ são os mais fortes, e sendo assim, eles não cometem erros de nenhuma natureza. Já os detectores da classe $\diamond\mathcal{W}$ podem cometer erros tanto de completude como de precisão, e isto quer dizer que eles podem suspeitar de processos corretos e não suspeitar de processos falhos.

Desde sua introdução, os detectores de falhas têm sido largamente estudados tanto em termos teóricos (definição de novas classes de detectores e verificação das restrições necessárias para sua implementação), quanto práticos (sua exploração para a construção de sistemas distribuídos tolerantes a faltas) e com ênfase no contexto de faltas de parada, modelo de falhas para o qual ele foi inicialmente proposto. Contudo, há autores que afirmam que a aplicabilidade dos detectores de falhas no contexto de faltas bizantinas ainda é limitada em virtude do comportamento arbitrário que um processo faltoso pode apresentar [52], o que caracteriza um padrão totalmente diferente do que é observado nas faltas de parada. Assim, em [52] são apresentados os fatores que impedem a concretização de detectores de falhas bizantinas, considerando a abordagem já existente nos detectores de falhas de parada, e que por sua vez são:

- **Ortogonalidade de especificação:** a especificação dos detectores de falhas de parada é independente do(s) algoritmo(s) que o(s) usa, tendo em vista que o comportamento de um processo faltoso considerando a semântica de faltas de parada é sempre o mesmo. Assim, se for considerada a possibilidade de faltas bizantinas nos processos, essa independência

deixará de existir, uma vez que os possíveis comportamentos que caracterizam uma falta bizantina de um dado sistema é completamente dependente do algoritmo em execução;

- **Generalidade de implementação:** a implementação de detectores de falhas de parada usa a noção de *timeouts* e algumas premissas de sincronismo. Deste modo, ele pode ser usado para implementar diversos algoritmos considerando as características do ambiente. No entanto, se considerarmos a semântica de faltas bizantinas (arbitrárias) não é possível implementar detectores de falhas que possam ser usados em diversos algoritmos;

- **Encapsulamento de falhas:** os detectores de falhas de parada encapsulam todos os aspectos relacionados as faltas/falhas, e deste modo ele pode ser facilmente usado por qualquer processo que conheça sua interface. No contexto de faltas bizantinas esse encapsulamento não é possível, uma vez que o comportamento bizantino é diretamente relacionado ao algoritmo a ser implementado;

- **Transparência de falhas:** dado que os detectores de falhas de parada encapsulam os aspectos relacionados a faltas/falhas do sistema, os algoritmos (de acordo e consenso) que usam estes detectores podem ser usados para implementar outros algoritmos (como o de *atomic broadcast*). No entanto, essa hipótese não é valida se considerarmos a possibilidade de faltas bizantinas.

Tomando em conta estes fatores, os mesmos autores introduziram o conceito de **detector de mudez** [53, 52], que consiste em uma abordagem intermediária entre detecção de falhas bizantinas e detecção de falhas de parada, conforme ilustrado na figura 2.5. Para viabilizar a detecção é introduzido um novo padrão de comportamento dos algoritmos considerando os aspectos sintáticos (das mensagens) requeridos pelos mesmos. Assim, se um processo não envia mensagens regularmente ou então se a sintaxe das mensagens recebidas não está de acordo com os requisitos dos algoritmos é dito que o processo sofreu uma **falta/falha de mudez**. O **detector de mudez** tenta capturar paradas algorítmicas (não físicas) que indicam o processo parou de enviar mensagens de acordo com o algoritmo que ele deveria executar. Desta forma, se um algoritmo $\mathcal{A}$ que gera um conjunto de mensagens que seguem uma sintaxe regular, chamadas de mensagens $m_{\mathcal{A}}$, define-se que um processo p é **mudo** se ao executar um algoritmo $\mathcal{A}$ em conjunto com um processo q, ele pára de enviar mensagens $m_{\mathcal{A}}$ prematuramente para q. A classe de detectores de mudez $\diamond\mathcal{M}_{\mathcal{A}}$ é definida a partir das seguintes propriedades [52]:

- **Completude $\mathcal{A}$ muda:** existe um tempo após o qual todos os processos mudos para um processo correto p, em relação ao algoritmo $\mathcal{A}$, são suspeitos por p continuamente;
- **Precisão $\mathcal{A}$ eventualmente fraca:** existe um tempo após o qual um processo correto p não poderá ser suspeito de ser mudo, em relação ao algoritmo $\mathcal{A}$, por nenhum outro processo correto.

Da mesma forma que os detectores de falhas de parada, a utilização dos detectores de mudez requer alguma premissa concernente ao sincronismo para que se possa assegurar a precisão. Contudo, é valido salientar que os detectores de mudez não podem ser implementados para qualquer algoritmo, para tanto o algoritmo $\mathcal{A}$ deve ser caracterizado por três atributos, conforme [52]: (a) o algoritmo deve ser baseado em *rounds*; (b) em cada *round* o algoritmo deve ter um coordenador (possivelmente rotativo); (c) os processos corretos devem ter um padrão de comunicação homogêneo. Ainda assim, como os detectores da classe $\diamond\mathcal{M}_{\mathcal{A}}$ não encapsulam as faltas bizantinas em plenitude, mas somente as de mudez, alguns tipos de faltas podem não ser detectadas como é o caso de mensagens conflitantes, mensagens inválidas (considerando a precedência de eventos) e mensagens omitidas (deixar de enviar uma mensagem de uma seqüencia).

Figura 2.5: Relação entre os modelos de falhas [52]

2.3 Conclusões do Capítulo

Neste capítulo foi apresentado o estudo realizado a fim de formar o embasamento teórico necessário para a consolidação da proposta desta dissertação. A partir deste embasa-

mento pudemos verificar que grande parte do arcabouço existente em algoritmos tolerantes a faltas é voltado para sistemas síncronos, isto é, nenhum deles garante progresso em sistemas puramente assíncronos. Assim, ao longo dos anos foram propostas diversas abordagens intermediárias ao modelo assíncrono, no intuito de estabelecer premissas e requisitos mínimos para o funcionamento dos algoritmos distribuídos tolerantes a faltas neste modelo.

Além do mais, uma explanação do conceito de modelo de sistema também foi apresentada. A compreensão do conceito de modelo de sistema é fundamental quando se trata da especificação de sistemas tolerantes a faltas, pois é a partir deste modelo que é possível estabelecer que garantias o sistema pode fornecer, mediante aos tipos de faltas para o qual o sistema foi especificado.

Capítulo 3

Fundamentos de Replicação em Sistemas Distribuídos

Na atualidade se verifica que há uma crescente participação dos computadores, e dos serviços fornecidos pela Internet e outras redes de computadores na vida da sociedade. Como conseqüência, as expectativas dos usuários em relação à qualidade dos sistemas/serviços têm aumentado na mesma proporção. Entretanto, como os sistemas de computação são cada vez mais complexos, dinâmicos e conseqüentemente mais suscetíveis a falhas, a manutenção destes atributos de qualidade é uma tarefa cada vez mais difícil de concretizar. A noção de **segurança de funcionamento** [10] (*dependability*) foi a introduzida no intuito de estabelecer um conjunto de atributos e meios com base nas ameaças existentes, e também para definir mecanismos visando preservar a confiabilidade, disponibilidade, integridade, reparabilidade e desempenho dos sistemas. Idéia por trás do conceito de **segurança de funcionamento** é prover subsídios para concretizar as expectativas que concernem a qualidade de serviço exigida pelos usuários, e, deste modo pode-se dizer que **segurança de funcionamento** concerne à qualidade do serviço fornecido, de tal forma que os usuários possam depositar no mesmo uma confiança justificada.

Além do mais, ao longo dos anos têm se verificado que a fidedignidade (*dependability*) de um sistema de computação pode ser alcançada por meio do emprego de técnicas de **tolerância a faltas** [21]. O uso destas técnicas permite que o sistema permaneça em funcionamento mesmo que uma parte de seus componentes venha a falhar, além de prover ao sistema a capacidade de recuperação de falhas parciais. A tolerância a faltas [21] pode ser vista

como capacidade do sistema se manter de acordo com sua especificação mesmo na presença de circunstâncias adversas no ambiente (faltas/erros/falhas). Usualmente a tolerância a faltas é obtida na prática por meio de redundâncias em nível de *software* e de *hardware*. Em se tratando de *softwares*, a redundância é implementada a partir de abstrações de replicação [54], que em conjunto com outros mecanismos permitem implementar serviços tolerantes a faltas (vide capítulo 2) de qualquer natureza.

Este capítulo visa apresentar as principais técnicas de replicação em sistemas distribuídos, bem como o seu enquadramento no modelo de faltas/falhas bizantinas (alvo deste trabalho). Também é apresentado um estudo sobre o estado da arte em replicação de sistemas/serviços, considerando o modelo de falhas bizantinas.

3.1 Técnicas de Replicação em Sistemas Distribuídos

A redundância de *software* por meio de replicação [54, 21] é uma das técnicas mais usuais no âmbito de tolerância a faltas. A unidade de replicação é conhecida na literatura como réplica [54], que por sua vez consiste em um componente de software, no qual são encapsulados alguns dados que são caracterizados como "estado" da réplica. Por meio de técnicas de replicação adequadas é possível fornecer aos sistemas distribuídos: confiabilidade, disponibilidade e integridade. Conforme já citado, o emprego de técnicas de replicação tem permitido a concretização de sistemas distribuídos mais robustos e capazes de suportar faltas de diversas naturezas, não permitindo que a falha de uma réplica do conjunto comprometa o funcionamento do todo. Cabe ressaltar que esse tratamento de falhas é realizado de forma abstrata/transparente ao usuário final da aplicação, causando a impressão de que o mesmo está acessando uma aplicação implementada por um único componente.

Na literatura são encontradas quatro técnicas de replicação, onde cada uma delas provê características distintas quanto ao método que emprega para manter a consistência de estado das réplicas. A corretude, propriedade de segurança (*safety*) de um sistema replicado, é estritamente depende de uma modelo de consistência conhecido como linearilidade [55], que mantém a propriedade de consistência de forma rigorosa, fazendo com que operações executadas concorrentemente sobre um dado objeto replicado aparentem ter sido executadas de forma seqüencial, uma após a outra. No entanto, para a implementação de semânticas mais fortes

como a linearilidade é requerido o consenso/acordo tolerante a faltas, e para tanto, as abordagens de replicação linearizáveis fazem o uso de protocolos de coordenação de replicas (acordo, difusão atômica) [56, 2] para assegurar a consistência de estados e a transparência do conjunto, bem como para prover mecanismos para o gerenciamento e administração de réplicas, no sentido de permitir a recuperação (parcial ou total) das réplicas que por ventura venham a falhar.

As quatro técnicas de replicação encontradas na literatura são: (a) replicação passiva (ou primária-backup); (b) replicação ativa (ou máquina de estados); (c) replicação semi-ativa; e (d) replicação semi-passiva. O tratamento de faltas/falhas nas abordagens passiva, semi-ativa e semi-passiva consiste no isolamento da réplica faltosa do serviço, enquanto que a abordagem ativa realiza o mascaramento das faltas/falhas parciais do serviço. Convém salientar que a escolha da técnica de replicação mais adequada para a implementação de um dado sistema deve ser realizada tomando como base o tipo aplicação, a(s) classe(s) de faltas a serem toleradas e também as características do sistema distribuído [28]. Além do mais, o grau de replicação (ou número de réplicas do conjunto) também está estritamente ligado a(s) classe(s) de falta(s) que se deseja tolerar.

3.1.1 A Abordagem de Replicação Primária-Backup

A abordagem de replicação primária-backup [56] (ou replicação passiva) implementa um serviço a partir de um conjunto de réplicas sendo que uma delas exerce o papel de primária, a qual por sua vez tem a responsabilidade de receber as requisições advindas dos clientes, processá-las e enviar as respostas aos respectivos clientes. As demais réplicas (passivas) são utilizadas para fins de contingência e têm a função de substituir o primário caso este venha a falhar. As réplicas passivas não interagem diretamente com os clientes, sendo que a única comunicação efetuada por elas ocorre entre a primária e elas.

A replicação passiva é a abordagem mais simples dentre as existentes. Esta técnica provê a linearilidade a partir do algoritmo de **coordenação de réplicas** empregado, pois tendo em vista que a réplica primária é a única a receber as requisições dos clientes, é ela quem define (e assegura) a ordem total das requisições para todas as réplicas. Esta abordagem permite realizar a manutenção da consistência de estados por meio da geração de *checkpoints*[1] entre

[1] Pontos de Sincronização do Protocolo e Serviço

intervalos de requisições executadas, ou a cada período de tempo. Isto significa que, a cada *N* requisições processadas ou a cada *X* unidades de tempo (de acordo com o método escolhido) o estado é propagado para as réplicas de *backup* para fins de estabilização do serviço, conforme apresentado na figura 3.1.

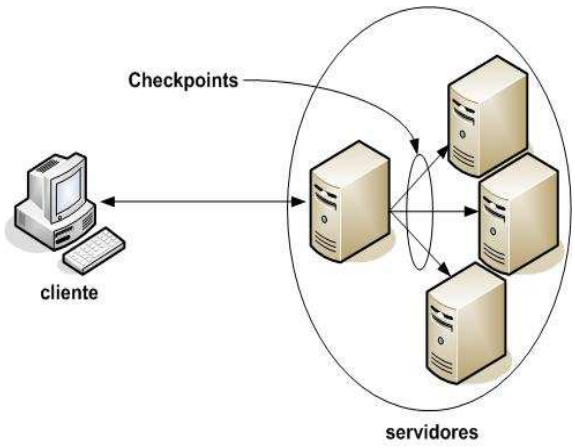


Figura 3.1: Replicação Passiva (primária-backup)

Esta abordagem de replicação prega que na ocorrência de faltas(s) no primário uma das réplicas de *backup* assume o papel de primário para assegurar o progresso (*liveness*) [57] do protocolo (e do serviço replicado). A escolha do novo primário é feita por meio de um algoritmo de eleição [58, 59], onde após a escolha do novo primário o serviço passa por um processo de estabilização (possivelmente bloqueante) para realizar a manutenção do estado do novo primário, e então permitir o recebimento de novas requisições.

Uma análise na abordagem permite verificar que, quando da ocorrência de falta(s) o processo de estabilização pode implicar na perda das requisições que foram enviadas até o momento da falha, isto é, todas as requisições posteriores ao último *checkpoint* são perdidas. Deste modo, para reduzir o impacto e minimizar as perdas admite-se o uso de um mecanismo de *logging* (em armazenamento estável) de requisições recebidas/executadas entre cada *checkpoint*. No entanto, como é sabido que a capacidade de armazenamento de *logging* é finita, esta abordagem de replicação deve definir uma política para coleta de lixo para evitar a saturação do *log*.

É valido salientar que em virtude da simplicidade da abordagem primária-backup esta

não é adequada para serviços críticos que não admitem interrupção. Isso se dá pelo fato de que o processo de recuperação do primário despende sobrecarga considerável no sistema, além de que o serviço permanece indisponível durante todo o processo de estabilização. Entretanto, a grande vantagem desta abordagem em relação às demais, é o fato dela não ser restrita à implentação de serviços deterministas, já que o primário impõe o seu estado sobre os *backups* [28]. Por fim, dada a sua simplicidade, esta abordagem de replicação é adequada somente para sistema onde se admite faltas/falhas de parada e de omissão.

3.1.2 A Abordagem de Replicação Máquina de Estados

A abordagem de replicação máquina de estados [2] (ou replicação ativa) é a técnica onde, ao contrário da abordagem anterior, não há réplicas de *backup* e todas as réplicas não faltosas são ativas. O gerenciamento de réplicas é feito através da abstração de máquinas de estados, e desta forma cada réplica admite uma composição de **variáveis de estado** (que armazenam os dados que concernem ao estado da máquina) para possibilitar a consistência entre o conjunto de réplicas, e um conjunto de **comandos** que representam as funções de transição de estados. Para tanto, as funções de transição de estado devem ser deterministas, de modo que permitam que uma seqüencia de operações (requisições) passadas para a função de transição produza o mesmo resultado de saída (respostas) em cada uma das réplicas. Como conseqüência, cada máquina de estados tem sua caracterização semântica garantida em virtude das saídas serem determinadas exclusivamente pelo estado inicial da réplica e pela seqüencia de operações por ela processadas, sem interferência de outros fatores como tempo ou particularidades do ambiente de execução [2].

Visto que nesta abordagem não existe uma réplica primária (todas são ativas), todas as réplicas corretas devem receber as mesmas requisições, processar e responder aos respectivos clientes, conforme ilustrado na figura 3.2. Deste modo, para manter a consistência de estados entre as réplicas, esta abordagem de replicação tem como premissa um requisito conhecido como **determinismo de réplica** [2]. Este requisito define que réplicas corretas, partindo de um mesmo estado inicial e sujeitas a mesma seqüencia de operações, isto é, na mesma ordem relativa [1] devem produzir as mesmas saídas (ou o mesmo estado final). Em suma, a abordagem de replicação máquina de estados pode ser entendida como um conjunto de réplicas que iniciam suas execuções a partir de um mesmo estado inicial e processam a mesmas seqüencia

de operações. Esta característica é conhecida na literatura como **coordenação de réplicas** e admite para tanto, dois requisitos [2]:

- **Acordo:** todas as réplicas corretas recebem as mesmas requisições;
- **Ordem:** todas as réplicas corretas processam as requisições recebidas em uma mesma ordem.

Usualmente a coordenação de réplicas é realizada com o auxilio de protocolos de difusão com ordem total [60] (ou difusão atômica), para se obter estes requisitos quando do envio das requisições ao conjunto de réplicas. Assim, o funcionamento desta abordagem se dá da seguinte forma: (*i*) o cliente envia uma requisição ao serviço para todas as réplicas por intermédio do protocolo de difusão atômica; (*ii*) ao receber a requisição, cada réplica processa a operação, atualiza seu estado (quando necessário) e envia ao cliente o resultado da operação. A consolidação da resposta por parte do cliente está intimamente relacionada ao modelo de falhas que se deseja tolerar, e pode ser realizado por intermédio de algumas alternativas como: o primeiro resultado recebido é retornado; os resultados passam por um votador (componente do cliente) que seleciona o valor mais freqüente; ou ainda para faltas bizantinas, uma resposta é aceita se o cliente acusa o recebimento de $f + 1$ (f é o número máximo de faltas toleradas) respostas iguais de diferentes réplicas (que indica um quorum onde há pelo menos uma réplica correta).

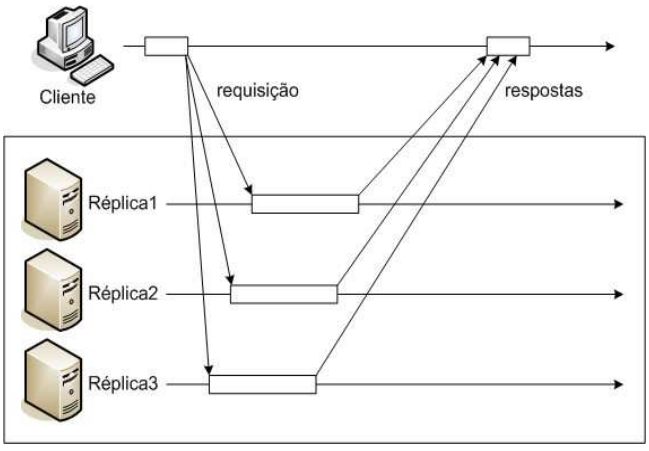


Figura 3.2: Replicação Máquina de Estados

Esta abordagem de replicação admite algumas otimizações para permitir o relaxamento dos requisitos de acordo e ordenação durante a execução de algumas operações. Por exemplo,

operações que não produzem alteração do estado das réplicas (operações de leitura) podem ser enviadas diretamente ao conjunto de réplicas sem passar pelo protocolo de difusão atômica. Desta forma, ao receber a requisição cada réplica responde imediatamente ao cliente, que, consolida a operação quando acusar o recebimento de $n - f$ respostas iguais. Esta otimização permite que operações de leitura sejam executadas em dois passos de comunicação (envio e resposta) em casos onde há ausência de falhas.

As características desta abordagem tornam-na apta a tolerar todos os modelos de falhas. Contudo, convém salientar que o número de réplicas do conjunto é estritamente dependente do modelo de falhas que se deseja tolerar. Neste caso, para faltas de parada e de omissão são requeridas $f + 1$ réplicas; faltas bizantinas e faltas de valor são requeridas $2f + 1$ réplicas; e por fim para faltas de temporização são necessárias $f + 1$ réplicas ([61] *apud* [28]). Uma observação importante e passível de comentários é o número de réplicas necessárias para tolerar faltas bizantinas que é $2f + 1$, caso seja as mensagens cheguem ordenadas para as réplicas, e não seja necessária a realização de um acordo, nem tampouco comunicação entre elas. No caso das réplicas incorporarem o protocolo de acordo (e difusão atômica), o número de réplicas necessárias passa a ser $3f+1$ (mínimo necessário para o acordo bizantino).

Por fim, a abordagem de replicação máquina de estados é a mais apropriada para implementar aplicações que requerem serviços sem interrupção e com menor custo computacional no caso da ocorrência de falhas, já que as falhas parciais são mascaradas. Em particular, neste trabalho estamos interessados em fornecer uma arquitetura que implementa esta abordagem de replicação, dada a sua capacidade de tolerar faltas bizantinas.

3.1.3 As Abordagens de Replicação Semi-Ativa e Semi-Passiva

Outras abordagens de replicação encontradas na literatura são as abordagens semi-ativa [62] e semi-passiva [63]. Ambas as abordagens podem ser vistas como a combinação das abordagens de replicação passiva e replicação ativa em virtude de suas características, com algumas diferenças entre elas.

3.1.3.1 Replicação Semi-Ativa

Nesta abordagem de replicação todas as réplicas são ativas, mas uma única réplica do grupo é privilegiada (e exerce a função de líder). Conforme ilustrado na figura 3.3, todas as réplicas recebem e executam as requisições, no entanto, a réplica privilegiada é a responsável por determinar a ordem de execução das requisições, bem como pelo envio do resultado da operação ao cliente.

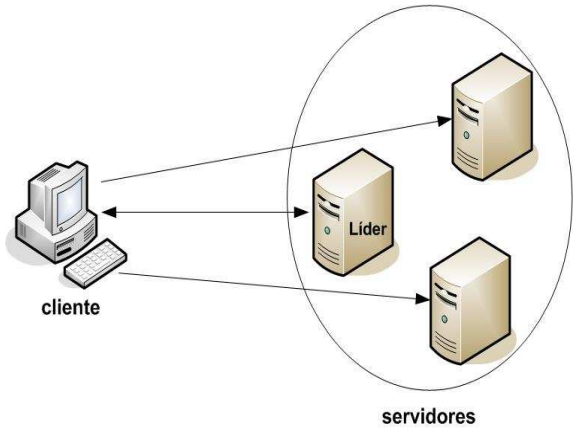


Figura 3.3: Replicação Semi-Ativa

Um requisito para a implementação desta abordagem de replicação reside no fato de que ela é restrita à implementação de serviços deterministas, uma vez que todas as réplicas são ativas e a consolidação do resultado depende da resposta de cada uma delas. No caso da réplica privilegiada falhar, uma das réplicas do grupo assume o papel de líder, que será escolhido por meio de um algoritmo de eleição.

3.1.3.2 Replicação Semi-Passiva

Na abordagem semi-passiva (figura 3.4) as requisições ao serviço são enviadas à todas as réplicas, no entanto, uma das réplicas é responsável por exercer o papel de líder que, é determinado a partir do paradigma do coordenador rotativo [36], isto quer dizer que, a cada operação executada a função do líder é exercida por uma réplica diferente, sendo geralmente determinado pela equação $l = (r \ \mathbf{mod} \ N) + 1$. Visto que o líder é o único a processar a requisição, o mesmo é responsável por executar a operação e impor o novo estado (a cada operação processada) para as réplicas de *backup* a partir do algoritmo *lazy consensus* [64].

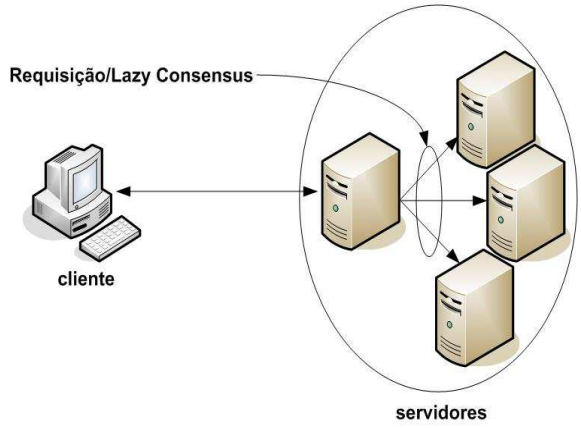


Figura 3.4: Replicação Semi-Passiva

Diferente da abordagem de replicação semi-ativa, as réplicas desta abordagem não necessitam ter comportamento determinista, uma vez que o estado das réplicas de *backup* é imposto pelo líder, periodicamente.

3.1.3.3 Consideração sobre Replicação Semi-Ativa e Replicação Semi-Passiva

Uma análise sobre as abordagens permite verificar que ambas tentam eliminar justamente os problemas das duas abordagens anteriores, sendo que um deles é evitar o uso de um protocolo de difusão atômica (que é relativamente caro e complexo), mas que é um requisito da abordagem de replicação ativa. Outra grande vantagem destas duas abordagens é que tanto as replicas semi-ativas como as semi-passivas não necessitam de recuperação de estado após a troca de líder. Para o primeiro caso, as réplicas são todas ativas, e, portanto, a evolução do estado do conjunto é sincronizada. No segundo caso, as réplicas têm seu estado atualizado (pelo líder) a cada requisição processada.

Considerando os modelos de falhas, a abordagem semi-ativa é adequada para os modelos de falhas por parada, omissão, temporização e valor, com uma ressalva para este último, onde o líder deve ser equipado com um votador para determinar a resposta correta. Já a semi-passiva é adequada apenas para tolerar faltas por parada. Neste caso, nenhuma das duas admite processos com comportamento arbitrário (bizantinos).

3.2 Técnicas de Replicação e o Modelo de Faltas Bizantinas

No contexto de aplicações tolerantes a faltas por meio de replicação de serviços, e dentre as abordagens de replicação apresentadas ao longo deste capítulo pode-se verificar que a abordagem de replicação máquina de estados [2] ganha destaque entre as demais, dada a sua capacidade de tolerar qualquer classe de faltas quando implementada com mecanismos apropriados. Além do mais, a abordagem de replicação máquina de estados define um arcabouço geral para permitir a implementação de serviços com confiabilidade e consistência. A confiabilidade pode ser vista como um dos requisitos mais importantes de um sistema distribuído, contudo, preservá-la durante o ciclo de vida do sistema é uma tarefa cada vez mais árdua, haja vista que com o passar dos anos a complexidade de software cresceu em grande proporção, e isso refletiu diretamente no processo de desenvolvimento de software. Como conseqüência, é cada vez mais difícil escrever aplicações livres de erros e *bugs*, e problemas como infecções por vírus e erros (humanos) de configurações também têm contribuido para a degradação da confiabilidade dos sistemas distribuídos.

Em face destes fatos, nos últimos tempos a comunidade de sistemas distribuídos tem concentrado esforços na investigação de modelos e protocolos mais robustos. Como conseqüência, muitos trabalhos têm produzido soluções práticas para replicação máquina de estados com foco à tolerância a faltas bizantinas [65, 5, 6, 7] visando tornar as aplicações resistentes inclusive a ataques de segurança bem sucedidos contra o sistema (intrusões), tornando assim os serviços tolerantes a intrusões [9]. Um ponto que merece destaque é que estas pesquisas têm demonstrado inclusive a viabilidade de se utilizar estas soluções para prover (e aumentar) a confiabilidade de sistemas/serviços reais e, portanto saindo somente do âmbito teórico.

3.2.1 Protocolos para Replicação Tolerante a Faltas Bizantinas

Nesta seção são apresentadas as principais propostas para tolerância a faltas bizantinas encontradas na literatura. Tendo em vista a existência de uma diversidade de propostas dentro deste âmbito, nos concentraremos em apresentar e discutir somente os trabalhos cujo foco é a viabilidade prática de implementação de sistemas, por estarem diretamente relacionados com a nossa proposta de trabalho.

3.2.1.1 PBFT - *Practical Byzantine Fault Tolerance*

O protocolo introduzido por Castro e Liskov [65] conhecido como PBFT (*Practical Byzantine Fault Tolerance*), foi um dos projetos pioneiros com foco a viabilidade prática de implementação de sistemas tolerantes a faltas bizantinas e é considerado por muitos o estado da arte em replicação com faltas bizantinas. O PBFT encapsula um conjunto de protocolos e mecanismos para a construção de sistemas tolerantes a faltas bizantinas, a partir da abstração de replicação máquina de estados [2]. O PBFT foi concebido com o objetivo de ser um protocolo robusto, e quando necessário, admite a degradação do desempenho no intuito de manter o serviço distribuído em funcionamento mesmo quando houver ataques e intrusões. Deste modo, os algoritmos e mecanismos do PBFT admitem uma série de otimizações para prover eficiência ao sistema na ausência de falhas. Do contrário, isto é, no caso da ocorrência de falhas, o desempenho do sistema é degradado, porém ainda assegurando as propriedades de correção do sistema.

O PBFT considera um *membership*[2] estático formado por $n \geq 3f + 1$ processos que implementam as réplicas do serviço, e um número arbitrário de clientes que não fazem parte do grupo de servidores. Cada réplica do PBFT implementa uma máquina de estados, bem como implementa uma cópia do serviço (que mantém um estado) e suas operações. As réplicas se comunicam umas com as outras no intuito de assegurar que todas elas executem a mesma seqüencia de operações, e para possibilitar o mascaramento das faltas individuais de réplica. Para o funcionamento pleno, o PBFT admite um sistema assíncrono com premissa de sincronismo terminal no sistema subjacente de comunicação. Um ponto interessante é o fato de que o protocolo tolera qualquer número de faltas durante a operação do serviço, desde que no máximo $f < \lfloor n/3 \rfloor$ ocorram simultaneamente. Visando o desempenho, os protocolos e mecanismos do PBFT usam MACs [42, 41] para autenticar as mensagens ponto a ponto (inclusive com os clientes) ao invés de usar criptografia assimétrica, pois os autores afirmam (e provam) que esse método de autenticação é mais eficaz se comparado com os métodos de autenticação convencionais [65].

O algoritmo de replicação do PBFT é baseado na abordagem de replicação ativa (máquina de estados) [2], onde a ordenação total (difusão atômica) [60] é realizada com base

[2]Indica o conjunto dos processos participantes do sistema

em um líder, função esta exercida por uma das réplicas do conjunto. O protocolo do PBFT é executado em sucessivas visões [66], onde o líder para uma visão é determinado pela expressão $idmodnum_v isao = 0$. O principio fundamental deste algoritmo é fazer com que as operações enviadas pelos clientes sejam executadas na mesma ordem por todas as réplicas. Neste caso, a réplica primária é quem determina a ordem de execução das operações recebidas, e se por ventura esta vier a falhar, a premissa de sincronismo admitida permite que o sistema inicie um protocolo de troca de visão para que uma outra réplica do grupo assuma o papel de primária.

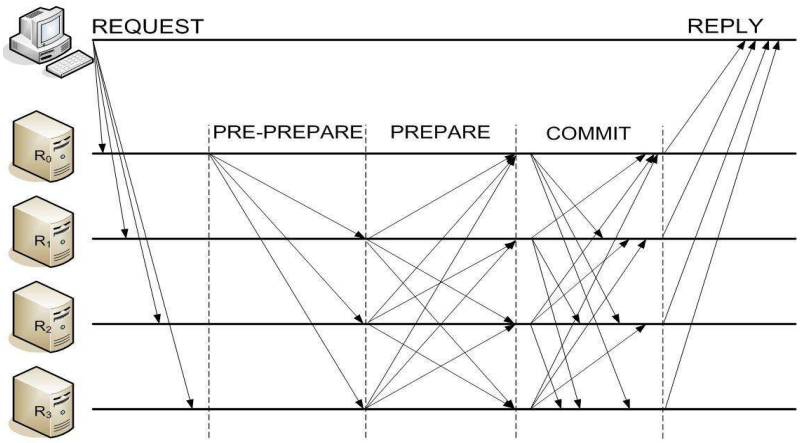


Figura 3.5: Execução do PBFT

O funcionamento do protocolo (ilustrado na figura 3.5) se dá da seguinte forma: inicialmente o cliente (já de posse das chaves simétricas para a autenticação) envia uma requisição através da difusão de uma mensagem *REQUEST* (contendo a operação ao serviço e argumentos) ao grupo de réplicas. As réplicas aceitam a requisição caso esta seja autêntica, e em seguida armazenam-na em um *log* para então dar inicio ao processo de ordenação. Neste ponto, a réplica primária define a ordem de execução da requisição recebida e então difunde ao grupo de réplicas uma mensagem *PRE-PREPARE* contendo o resumo criptográfico (doravante chamado apenas de resumo) da requisição e a ordem na qual deve ser executada. Tendo recebido a mensagem *PRE-PREPARE*, cada uma das réplicas difunde às demais réplicas uma mensagem *PREPARE* como forma de indicação aos demais de que está de acordo com a entrega da requisição em questão, e na ordem estabelecida. O protocolo avança para a próxima fase (o que indica que a mensagem está preparada) caso nenhuma outra requisição tenha sido preparada para a mesma ordem e se existir uma mensagem *PRE-PREPARE* e um certificado

de preparação[3] com $2f$ mensagens *PREPARE* concernentes a esta preparação. A geração deste certificado indica que pelo menos f processos corretos mais o primário receberam a mensagem e concordaram com a ordem determinada, o que torna improvável a existência de dois certificados de ordens diferentes para a mesma mensagem caso o número de participantes sejam $> 3f$.

Uma vez concluída a fase de preparação, resta apenas confirmar a execução da requisição. Deste modo, a última fase do algoritmo é a confirmação em que cada uma das réplicas difunde uma mensagem de *COMMIT* às demais réplicas, e consolida um certificado de confirmação a partir do recebimento $2f + 1$ mensagens de *COMMIT*. Esta última fase assegura que todas as réplicas concordam com a ordem da requisição, mesmo durante uma troca de visão. Por fim, as réplicas procedem com a execução da requisição do serviço tão logo que tenham concluído a execução de todas as requisições com o número de seqüencia anterior ao que acabou de ser confirmado. Finalizada a execução da requisição, cada réplica envia diretamente ao cliente uma mensagem *REPLY* contendo o resultado da operação, e o cliente por sua vez, aceita o resultado caso receba $f + 1$ mensagens *REPLY* iguais para a mesma requisição, o que corresponde a um certificado de resposta. Caso o cliente não receba a resposta dentro de um intervalo de tempo, a requisição é retransmitida por meio do algoritmo de replicação e este processo se repete até que o cliente tenha o certificado de resposta para a requisição.

Como se pode ver, o PBFT é um protocolo bastante robusto, porém, um pouco pesado. Para tanto, além da substituição da criptografia assimétrica por MACs, o PBFT implementa mais quatro otimizações:

- **Execução por antecipação:** esta otimização permite que as réplicas executem uma requisição logo após sua preparação, desde que as requisições anteriores tenham sido confirmadas. Com isso, é eliminada a fase de confirmação, onde as mensagens de *COMMIT* são enviadas juntamente com as próximas mensagens enviadas pelas réplicas (via *piggybacking*). No entanto, caso ocorra uma troca de visão é possível que as requisições não confirmadas sejam ignoradas e com isso é possível que o estado da réplica seja retornado para o do último *checkpoint* estável;

- **Requisições somente-leitura:** esta otimização permite que as requisições que não al-

[3] Na realidade o termo "certificado" denota um conjunto de mensagens iguais

teram o estado do serviço sejam executadas tão logo do seu recebimento pelas réplicas, sem a necessidade de passar pelo algoritmo de ordenação. O cliente consolida a operação se receber $n - f$ mensagens *REPLY* iguais para a mesma requisição;

- **Ordenação de requisições em lote:** esta otimização realiza o agrupamento de requisições, onde ao invés de executar o protocolo a cada requisição recebida, o algoritmo é executado para um conjunto de k requisições (um número de requisições agrupadas no lote). Nestes casos, o algoritmo só realiza a ordenação desde que alguns pontos sejam considerados, tais como, seja e o número de seqüência do último lote ordenado e p o número de seqüência da última requisição definida pelo primário. Quando o primário receber uma requisição, ele só inicia o protocolo se $p \geq e + k$, de outro modo a requisição é colocada em uma fila para posterior ordenação. Esta otimização é muito útil em situações onde a carga do sistema é alta.

- **Resumo de respostas:** a ultima otimização consiste em reduzir o consumo de banda a partir do envio de resumos criptográficos das respostas ao cliente. Assim, ao enviar a requisição o cliente designa uma das réplicas (baseado em algum esquema de aleatoriedade ou balanceamento) para enviar a resposta da requisição, enquanto as demais enviarão somente o resumo (*hash* ou *digest*) da resposta para fins de comparação. O cliente verifica a validade da resposta por meio dos resumos, onde aceita a resposta se a mesma for referente a f resumos enviados. Caso o cliente não obtenha a resposta, a requisição é re-enviada pelo método normal.

Para assegurar a vivacidade/progresso do protocolo no caso da ocorrência de falhas, o PBFT usa um mecanismo de troca de visão, que além de permitir o progresso do sistema, possibilita a realização de um acordo em relação à ordem das mensagens que ficaram pendentes na visão anterior. Para tanto, o algoritmo de troca de visão admite dois conjuntos de mensagens $\mathcal{P}$ e $\mathcal{Q}$, que contém as mensagens *PREPARE* e *PRE-PREPARE* de visões anteriores, respectivamente.

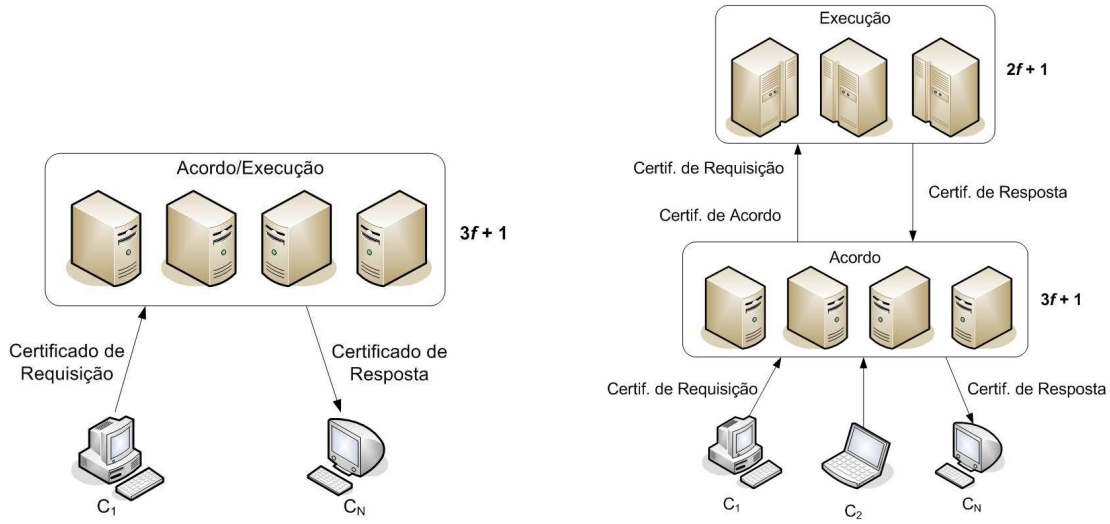
A troca de visão ocorre quando uma das réplicas de *backup* suspeita que na visão v a réplica primária é faltosa, então transpõe sua visão para $v + 1$ e envia uma mensagem *VIEW-CHANGE* para todas as réplicas do conjunto informando sobre sua troca de visão juntamente com seus conjuntos $\mathcal{P}$ e $\mathcal{Q}$. As demais réplicas aceitam a mensagem *VIEW-CHANGE* se e

somente se as mensagens contidas em $\mathcal{P}$ e $\mathcal{Q}$ pertencem a visões anteriores a $v + 1$, e então enviam ao emissor a notificação de aceitação por meio de uma mensagem *VIEW-CHANGE-ACK*. O novo primário da visão $v + 1$ é o processo que tem o identificador $p_i = ((v + 1)$ **mod** $n)$. Deste modo, o novo primário monta um certificado de troca de visão (denominado $\mathcal{S}$) baseado nas mensagens *VIEW-CHANGE* (uma única) e *VIEW-CHANGE-ACK* $(2f - 1)$ recebidas durante o processo de troca de visão. O certificado de troca de visão atesta que a maioria das réplicas corretas concordou com a troca de visão, além de permitir que o novo primário possa construir um ponto de sincronização (*checkpointing*) para que seja possível definir os números de seqüencia para as mensagens pendentes da visão anterior contidas nos conjuntos $\mathcal{P}$ e $\mathcal{Q}$. Por fim, o novo primário difunde uma mensagem *NEW-VIEW* ao grupo de réplicas, anexando o certificado de troca de visão, o *checkpoint* válido e as mensagens pendentes ordenadas. As demais réplicas aceitam e instalam a nova visão desde que tenham aceitado a mensagem *VIEW-CHANGE* correspondente.

Apesar de o PBFT ser considerado um protocolo pesado, as otimizações nele introduzidas permitem que ele se torne mais eficaz. A avaliação preliminar do PBFT apresentou latência 4 vezes maior em operações de escrita e 2 vez mais em operações de leitura, quando comparado a um serviço não replicado [65]. Além do mais, o PBFT é um dos únicos sistemas de replicação tolerante a faltas bizantinas que tem seu código aberto, o qual está disponível no sítio do projeto `http://www.pmg.lcs.mit.edu/bft/`.

3.2.1.2 Arquitetura de Separação do Acordo e Execução

A arquitetura de separação de acordo e execução introduzida por [5] contribuiu com um modelo engenhoso para a construção de sistemas tolerantes a faltas bizantinas baseados em replicação máquina de estados. Tradicionalmente os sistemas de replicação tolerantes a faltas bizantinas tais como o PBFT [3] e sua extensão BASE [4], combinam os aspectos de **acordo** e **execução** das requisições do serviço no mesmo conjunto de réplicas. Obviamente que estas características permitiram (e permitem) até então a implementação de sistemas com alto grau de confiabilidade e integridade. No entanto, a motivação de que existem boas práticas a serem exploradas sobre o tradicional modelo de replicação máquina de estados [2] permitiu que o trabalho de [5] fosse consolidado com muito êxito.



(a) Modelo de Replicação Tradicional (b) Separação de Acordo e Execução

Figura 3.6: Arquiteturas de Replicação

O objetivo fundamental desta arquitetura é separar as funções exercidas pelo conjunto de réplicas em dois conjuntos, sendo um para a realização do acordo bizantino [12], necessário para a ordenação das mensagens de requisição que chegam ao serviço, e o outro conjunto para realizar a execução das requisições já ordenadas no passo anterior pelo outro conjunto. Segundo [5], a separação dos serviços de acordo e execução em entidades distintas tira proveito de pelo menos duas vantagens em relação do modelo de replicação tradicional. A primeira vantagem é redução do custo de replicação e otimização da resistência nas réplicas de execução, uma vez que as requisições já chegam ordenadas para execução, o modelo requer somente a maioria das réplicas corretas para proceder com a execução das mensagens ordenadas, sendo então o número de réplicas necessárias para faltas bizantinas $2f + 1$ [67]. Entretanto, ainda são necessárias melo menos $3f + 1$ réplicas na camada de acordo bizantino para a ordenação das mensagens de requisição. Neste ponto a redução do custo de replicação é bem clara, uma vez que a execução das requisições requer um poder de computação relativamente maior o poder requerido por um simples serviço de acordo (e ordenação). Assim, a redução do número de réplicas de execução conseqüentemente reduz o custo de computação do serviço replicado em termos de operações de E/S, número de cópias do estado da aplicação e por fim, reduz o número de vezes que uma requisição é processada [5] no sistema. Além de também reduzir o custo de licenciamento de softwares, tendo em vista que menos versões da aplicação são necessárias. A segunda vantagem do desacoplamento do acordo e execução permite o uso de réplicas mais

baratas tanto em termos de *hardware* como de *software*, uma vez que são máquinas dedicadas a somente esta tarefa, e, portanto não requerendo grande capacidade de computação.

Para facilitar o detalhamento do protocolo, doravante o conjunto de servidores de acordo será denotado por $\mathcal{A}$, enquanto o conjunto de servidores de execução será representado por $\mathcal{E}$. Informalmente o protocolo de replicação proposto nesta arquitetura tem seu funcionamento da seguinte forma. Em primeira instância o cliente requisita uma operação ao serviço a partir do envio de um certificado de requisição contendo a operação (e argumentos), a estampilha de tempo e a identificação do cliente para os servidores $\mathcal{A}$, responsáveis por ordenar as requisições e entregá-las ao conjunto de servidores $\mathcal{E}$. Cada servidor da camada de acordo, ao receber um certificado de requisição executa um algoritmo de três passos. Primeiramente o servidor de acordo verifica se em seu *cache* existe um certificado de resposta para o cliente em questão, e com a mesma estampilha de tempo informada na requisição, caso a condição seja verdadeira o servidor envia o certificado contido no *cache* e não envia a requisição para processamento pelas réplicas de execução. Neste caso o servidor observa que se trata de uma retransmissão da requisição (mais detalhes adiante). É valido salientar que este *cache* contém a resposta da requisição mais recente enviada pelo cliente. Assim, caso o servidor não encontre o certificado de resposta em seu *cache*, ele inicia o processo de geração do certificado de acordo a partir da execução do protocolo PBFT/BASE [65, 4] (explicado em detalhes na seção 3.3.1.1) onde ao final da execução (fase de *COMMIT*) cada servidor $\mathcal{A}$ envia o certificado de requisição (recebido do cliente) e a ordem de execução da mesma aos servidores $\mathcal{E}$, constituindo assim, um certificado de acordo atestado/autenticado por pelo menos $2f + 1$ servidores $\mathcal{A}$ durante a realização do acordo.

Uma vez que o protocolo PBFT [65] foi usado para o acordo bizantino, e considerando que este protocolo implementa acordo e execução no mesmo conjunto de réplicas, foi necessária uma adaptação na arquitetura de separação de acordo e execução [5] para prover o funcionamento de acordo com o especificado. Mais precisamente foi construída uma espécie de **fila replicada** para emular o comportamento de uma máquina de estados. Assim, cada servidor do conjunto $\mathcal{A}$ é equipado com uma instância da **fila de mensagens** (sua máquina de estados), uma variável *maxN* que contém o número de seqüencia mais alto dentre as requisições recebidas e uma lista de pendências que contém os certificados de requisição e de acordo e as informações sobre o temporizador das requisições que foram enviadas mas

que ainda não receberam resposta(s). Quando é realizada uma operação de inserção, a fila de mensagens armazena os certificados na lista de pendências, atualiza a variável *maxN* e difunde os certificados para o conjunto de servidores $\mathcal{E}$, além de definir um valor arbitrário para o temporizador de cada requisição enviada. Para prevenir que a lista de pendências venha a saturar (com muitas requisições já ordenadas, mas não executadas), é definida uma variável P que indica o número máximo de pendências permitido, e deste modo uma operação de inserção para um número n só será completada caso a resposta da requisição $n - P$ já tenha sido recebida, do contrário a operação de inserção permanece bloqueada até que esta condição seja satisfeita. Quando uma instância da fila de mensagens receber um quorum de $f + 1$ respostas do conjunto de servidores $\mathcal{E}$ é realizada a exclusão dos certificados para a respectiva requisição da lista de pendências, todos os temporizadores (para o caso de retransmissão) são cancelados e a resposta é encaminhada ao cliente. No caso do temporizador de uma requisição contida na lista de pendências expirar, a instância da fila re-envia os certificados de requisição e de acordo e atualiza o intervalo do temporizador para duas vezes valor definido anteriormente.

Os servidores $\mathcal{E}$, que por sua vez implementam as réplicas (máquinas de estados) são responsáveis por executar as requisições já ordenadas pelos servidores $\mathcal{A}$. Deste modo, um servidor $\mathcal{E}$ ao receber um certificado de requisição (autenticado pelo cliente), bem como o certificado de acordo (indicando que a requisição tem número de seqüencia n) válidos para esta requisição, armazena os certificados em uma lista de requisições pendentes. Assim, uma vez que todas as requisições com números de seqüencia $< n$ foram recebidas e executadas cada servidor $\mathcal{E}$ inicia a execução da operação recebida (e retira da lista de pendentes). Ao proceder com a execução cada servidor $\mathcal{E}$ primeiramente verifica se a estampilha de tempo da requisição é maior que o da última requisição executada para este cliente e então efetua a execução da operação, atualiza uma tabela denominada $Reply_c$ com o resultado da operação e a estampilha de tempo (usada como mecanismo de retenção de resultados), e envia a resposta da operação ao conjunto de servidores $\mathcal{A}$. Caso a estampilha de tempo da requisição enviada seja igual ao que está contido em $Reply_c$, isso é caracterizado como uma retransmissão do cliente, e desta forma os servidores $\mathcal{E}$ enviam a resposta armazenada em $Reply_c$ juntamente com o identificador da visão e número de seqüencia nas quais a requisição foi executada, ao conjunto de servidores $\mathcal{A}$. Caso um servidor $\mathcal{E}$ receba a requisição n sem ter recebido a requisição $n - 1$, ele difunde uma mensagem de solicitação de re-envio da requisição $n - 1$ para o conjunto de servidores $\mathcal{A}$. Por fim, cada servidor $\mathcal{A}$ ao receber $f + 1$ respostas iguais para a mesma requisição as

armazena em seu *cache* (para o caso de retransmissão) e as retransmite ao cliente indicando que a operação foi executada/completada com êxito. Uma vez que a resposta foi autenticada por $f + 1$ servidores $\mathcal{E}$, isto constitui um certificado de resposta que atesta a validade da mesma. Uma observação relevante é o fato que ao enviar uma requisição, o cliente inicia um temporizador e caso ele seja expirado e todas as respostas ainda não tenham sido recebidas, o cliente retransmite requisição para os servidores $\mathcal{A}$.

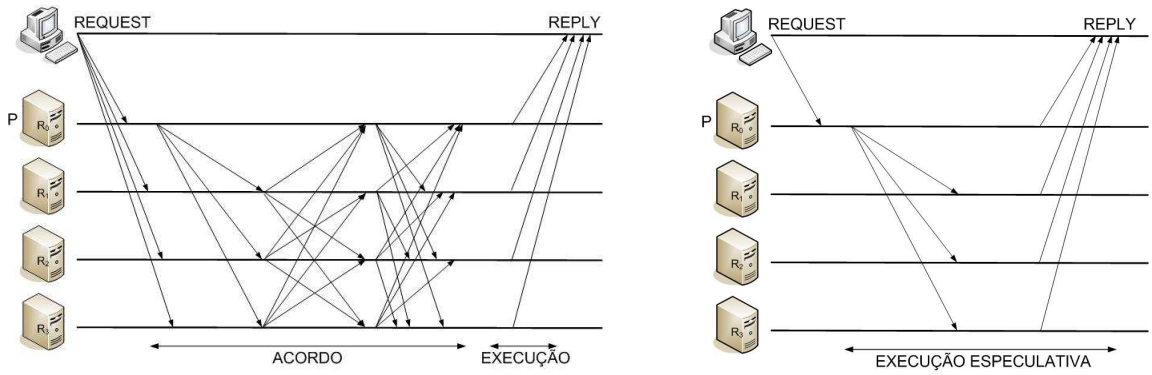
A fim de estabilizar o estado da aplicação, bem como realizar a coleta de lixo da lista de requisições pendentes, os servidores $\mathcal{E}$ periodicamente constroem *checkpoints* incluindo o estado da aplicação e a lista de respostas enviadas aos clientes ($Reply_c$). Após a geração do *checkpoint* os servidores $\mathcal{E}$ constroem uma prova de estabilidade para o mesmo, a partir da realização de um acordo entre eles constituindo então um certificado de *checkpoint* que é atestado por pelo menos $f + 1$ servidores $\mathcal{E}$.

Como se pode verificar, a adoção do modelo arquitetural de replicação em camadas permite uma série de vantagens de cunho prático em relação ao modelo tradicional, o que se pôde observar a partir dos resultados reportados para esta arquitetura em [5], os quais mostram um custo adicional (em termos de latência) moderado quando comparado aos outros trabalhos do mesmo âmbito. No entanto, além das vantagens do uso desta arquitetura já descritas no início desta seção, uma terceira vantagem é a capacidade de se entrepor um *firewall* de privacidade para proteger a confidencialidade do sistema replicado, item que geralmente é afetado pela replicação de serviços. Além do mais, a introdução de uma camada de confidencialidade permite eliminar a necessidade de um votador (no cliente) para classificar as respostas enviadas pelas réplicas [68].

3.2.1.3 Zyzzyva - *Speculative Byzantine Fault Tolerance*

O Zyzzyva [6] é a primeira proposta de um protocolo com foco à eficiência para a implementação de sistemas replicados tolerantes a faltas bizantinas, cujos objetivos são a confiabilidade e o alto-desempenho dos serviços replicados, e tendo sido projetado para ser um protocolo simples, considerando a complexidade dos protocolos propostos até então (e apresentados nas seções anteriores). Nesta abordagem, igualmente como nos demais protocolos de replicação com faltas bizantinas uma réplica exerce o papel de primária e é a

responsável por determinar a ordem de processamento das requisições clientes para as demais réplicas. No entanto, diferente dos demais protocolos BFT, as réplicas do Zyzzyva executam e respondem a requisição do cliente de forma especulativa, isto é, sem executar um protocolo de acordo/ordenação de três fases (como de costume) para estabelecer o acordo sobre a ordem na qual serão processadas as requisições. Ao invés disso, as réplicas adotam de forma otimista a ordem de processamento da requisição proposta pela réplica primária e respondem imediatamente ao cliente. Porém, como o protocolo é otimista, é possível que temporariamente as réplicas estejam inconsistentes entre si. O cliente por sua vez, observa as inconsistências e auxilia as réplicas a convergir para uma ordem atômica de processamento das requisições e considerando válidas somente as respostas que forem consistentes, caso contrário, o cliente até o sistema convergir para um estado estável (e consistente). Isto só é possível porque junto às respostas das réplicas são anexadas informações da história [55] executada, e assim o cliente tem subsídios para determinar se as respostas e a história são estáveis e podem ser eventualmente confirmadas.



(a) Protocolo BFT Tradicional (ordenação pessimista) (b) Zyzzyva (ordenação otimista)

Figura 3.7: Comparação de Protocolos de Replicação

Como se pode observar na figura 3.7(a), o protocolo de replicação tradicional incorre em uma sobrecarga no serviço replicado, pois antes da execução a ordem de processamento é assegurada de forma pessimista. No caso do Zyzzyva [6] (apresentado na figura 3.7(b)) não existe um acordo explícito entre as réplicas para determinar a ordem de processamento da requisição. Deste modo, a grande questão por trás deste protocolo é a garantia de que as respostas enviadas aos clientes **corretos** se tornam estáveis. Assim, mesmo tendo a participação do cliente na confirmação do protocolo, fica a cargo das réplicas a responsabilidade por garantir que todas as requisições oriundas dos clientes **corretos** eventualmente sejam completadas.

O Zyzzyva mantém suas propriedades de correção mesmo em ambientes assíncronos, no entanto, para assegurar a vivacidade admitem-se premissas de sincronismo fracas (períodos de sincronia). O protocolo de replicação do Zyzzyva é composto por três sub-protocolos que são: (*i*) acordo; (*ii*) troca de visão; e (*iii*) *checkpoint*. O sub-protocolo de acordo é responsável por ordenar as requisições para posterior processamento pelas réplicas tem seu inicio quando a réplica primária recebe uma requisição do cliente c. Ao receber a requisição (cujo formato é $m = \langle REQUEST, o, t, c \rangle$) o primário verifica se a estampilha de tempo é maior que o da última requisição recebida para este cliente ($t > t'$), e caso a condição for falsa, a requisição é descartada, do contrário ele atribui um número de seqüencia indicando a ordem na qual esta deve ser processada (baseada em seu estado), e difunde a requisição e a ordem para as demais réplicas através da mensagem $\langle \langle ORDER - REQ, v, n, h_n, d, ND \rangle, m \rangle$ onde v indica a visão na qual a ordem foi estabelecida, n indica o número de seqüencia proposto para a requisição m, $d = H(m)$ corresponde ao resumo criptográfico [41] da requisição m, $h_n = H(h_{n-1}, d)$ é o resumo criptográfico da história computada até então, e ND é um conjunto de valores para as variáveis não-determinísticas da aplicação (hora, data etc).

Quando uma réplica recebe a mensagem $\langle \langle ORDER - REQ, v, n, h_n, d, ND \rangle, m \rangle$ do primário, ela analisa se a requisição ordenada é "bem-formada" por meio da verificação dos resumos criptográficos da mensagem e da história, além de verificar se n é uma unidade maior que a última requisição executada. No entanto, em virtude de problemas como perda de mensagens, réplica primária faltosa etc, é possível ocorrer *gaps* entre as requisições recebidas, e deste modo, o protocolo toma ações reativas para evitar que isso afete a corretude do sistema. No caso de n ser menor que a seqüencia da última requisição processada, a requisição é descartada. Se n for maior que a seqüencia da última requisição processada mais uma unidade ($max + 1$), a réplica descarta a requisição e envia uma mensagem $\langle FILL - HOLE, v, max + 1, n, i \rangle$ para o primário, e inicia um temporizador. Quando o primário recebe uma mensagem $FILL - HOLE$ de uma réplica i, ele envia a mensagem $\langle \langle ORDER - REQ, v, n', h_n, d, ND \rangle, m' \rangle$ para cada requisição m' tal que $((max + 1) \leq n' \leq n)$ que foram ordenadas na visão atual, e deste modo ignorando mensagens $FILL - HOLE$ de outras visões. Assim, se a réplica i recebe as mensagens $ORDER - REQ$ necessárias para preencher as lacunas existentes ela cancela os temporizadores, de outra forma, caso o temporizador venha a expirar e ela não receba as mensagens $ORDER - REQ$ necessárias, difunde $\langle FILL - HOLE, v, max + 1, n, i \rangle$ para as

demais réplicas e inicia o protocolo de troca de visão. Uma vez que a requisição é aceita, a réplica anexa a requisição ordenada a sua história local, executa a requisição de forma especulativa, isto é, sem realizar qualquer tipo de comunicação com as demais réplicas, e pega o resultado r da operação e envia diretamente ao cliente a partir de uma mensagem $\langle\langle SPEC - RESPONSE, v, n, h_n, H(r), c, t\rangle, i, r, OR\rangle$, sendo i o identificador da réplica que enviou a mensagem e OR a requisição ordenada corresponde.

No caso do cliente, que recebe a mensagem de resposta $\langle\langle SPEC - RESPONSE, v, n, h_n, H(r), c, t\rangle, i, r, OR\rangle$ de cada uma das réplica, é possível tomar caminhos diferentes dependendo das respostas recebidas. No primeiro caso o cliente aceita a resposta se $3f + 1$ mensagens $SPEC - RESPONSE$ para uma requisição têm as informações $v, n, h_n, H(r), c, t, r$ idênticas. Um segundo caso ocorre quando o cliente recebe menos de $2f + 1$ mensagens $SPEC - RESPONSE$, neste caso ele re-envia a requisição para todo o grupo de réplicas. A réplica i, de posse da requisição re-enviada verifica se já havia a executado anteriormente e neste caso a resposta é re-enviada ao cliente. Do contrário, a réplica i envia ao primário uma mensagem $\langle CONFIRM - REQ, v, m, i\rangle$ para a requisição m recebida e inicia um temporizador. Se o primário enviar a mensagem $ORDER - REQ$ correspondente à solicitação $CONFIRM - REQ$, a réplica i cancela o temporizador e executa a requisição (fluxo normal). Porém, se o temporizador expirar a réplica i inicia o protocolo de troca de visão. Outro caso ocorre se o cliente receber entre $2f + 1$ e $3f$ mensagens $SPEC - RESPONSE$ iguais (figura 3.8, passo 3), o que significa que o acordo não foi plenamente estabelecido e pode haver divergência entre as réplicas. Para tanto, o cliente constrói um certificado de confirmação para auxiliar no processo de convergência de estado entre as réplicas (figura 3.8, passo 4) e envia este certificado para todo o grupo. Cada réplica ao receber o certificado de confirmação verifica as informações e caso necessário, retrocede seu estado para que possa adotar a ordem correta. O cliente por sua vez aguarda por $2f + 1$ mensagens de confirmação das réplicas que indicarão a estabilidade do sistema e assim permitirá que o mesmo faça progresso e aceite a resposta.

O sub-protocolo de troca de visão é responsável por coordenar a eleição de um novo primário, quando o primário atual é faltoso ou quando o sistema se encontra com degradação de desempenho. Assim, quando alguma das réplicas suspeita que o primário da visão atual é faltoso, ele propõe a troca de visão enviando inicialmente uma mensagem $\langle I - HATE - THE - PRIMARY, v\rangle$ para o grupo de réplicas. Se a replica que propôs a troca

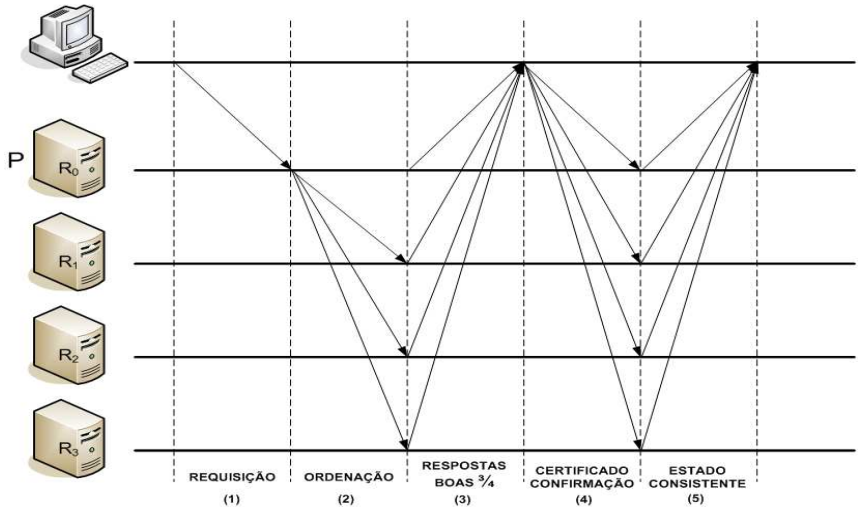


Figura 3.8: Zyzzyva com Faltas

receber $f + 1$ votos de suspeita do primário, ele usa os votos como prova e os envia junto a um certificado de troca de visão (mensagem VIEW-CHANGE) para as demais réplicas. Uma réplica correta, ao receber um certificado de troca de visão válido, se junta à nova visão e confirma a troca. Para possibilitar o restabelecimento do sistema a partir do ponto de parada (visto que o sistema é bloqueado na troca de visão), na mensagem VIEW-CHANGE são incluídos todos os certificados de confirmação de requisições, bem como as mensagens ORDER-REQ de conhecimento das réplicas desde o último *checkpoint* estável.

Por fim, o sub-protocolo de *checkpointing* permite que as réplicas construam pontos de sincronização do serviço dentro de um intervalo de requisições processadas. Para fins de sincronização, cada réplica mantém um *checkpoint* do estado da réplica, além do *snapshot* do estado da aplicação correspondente. A construção de um *checkpoint* estável é realizada por meio de um acordo entre as réplicas. Assim, quando uma réplica correta manifesta sua intenção em construir um ponto de sincronização, ela primeiramente gera os resumos criptográficos do estado da réplica (que inclui a história, as variáveis de controle etc) e do *snapshot* do estado da aplicação (ambos correspondentes a última requisição processada), os anexa a uma mensagem CHECKPOINT (devidamente assinada) e envia esta mensagem para as demais réplicas do grupo. Para o Zyzzyva, um *checkpoint* é considerado estável quando uma réplica correta recebe $2f + 1$ mensagens de *checkpoint* iguais e assinadas por diferentes réplicas. Note que o Zyzzyva requer $2f + 1$ mensagens iguais para consolidar um *checkpoint*, a despeito das $f + 1$ mensagens iguais requeridas pelos protocolos tradicionais [65, 5]. Isto se deve ao fato dos

clientes trabalharem com $3f + 1$ respostas especulativas.

O Zyzzyva foi o pioneiro a explorar a noção de execução "especulativa" [69] no contexto de faltas bizantinas. A idéia por trás da especulação é prover a capacidade dos clientes e- xecutarem as operações de um determinado serviço por tentativa, sem a necessidade de aguardar por uma confirmação. No entanto, o uso desta abordagem possibilita a perda de sincronismo por parte do serviço, e para resolver tal problema o protocolo também prevê a possibilidade de retroceder as operações que se tornaram inconsistentes (em virtude de tentativas de execução frustradas). A idéia de introduzir a abordagem de "especulação" no contexto de BFT trouxe ganhos extremamente significativos em termos de *throughput* e de latência, em relação às soluções usuais de replicação com faltas bizantinas.

3.2.1.4 A2M - *Attested Append-Only Memory*

É válido salientar que, além das diversas propostas de protocolos para replicação com faltas bizantinas, algumas soluções que buscam circunscrever algumas impossibilidades teóricas também têm sido propostas, principalmente no que tange a capacidade de implementar serviços tolerantes a faltas bizantinas com $n \leq 3f + 1$ réplicas. Um trabalho de grande interesse teórico e prático é o **A2M** [7], que por sua vez, permite reduzir o custo computacional de sistemas tolerantes a faltas bizantinas por meio de um componente seguro (uma abstração de *logging* confiável implementada sobre uma TCB).

Conforme já dito, o **A2M** (*Attested Append-Only Memory*) não é uma arquitetura para replicação propriamente dita, mas sim uma abstração *logging* confiável que permite realizar a atestação das mensagens do sistema no intuito de prover maior confiabilidade. O trabalho de [7] mostra que, por meio de um mecanismo de atestação é possível detectar a presença de um adversário e conseqüentemente realizar a prevenção/remoção das equivocações por ele emitidas ao sistema. A abstração do **A2M** é constituída por um conjunto de *logs* confiáveis (armazenados em uma TCB), nos quais são armazenadas as provas da linearilidade das mensagens emitidas no sistema. Cada elemento armazenado no *log* confiável é dotado de um rótulo, um número de seqüência da entrada, o valor da entrada e um resumo criptográfico acumulativo de todas as entradas inseridas no *log* até o momento ($H_n = H(H_{n-1} + V)$). Cabe ressaltar que no *log* só é permitida a inserção de novas entradas, e nunca a sua remoção e/ou

substituição. Em outros termos, a idéia por traz do **A2M** é armazenar as histórias [55] ocorridas no sistema em uma abstração de *logging* confiável e segura, para prover aos processos um mecanismo com capacidade de verificar e atestar a seqüência de mensagens circuladas no sistema até um dado momento (momento da realização da atestação).

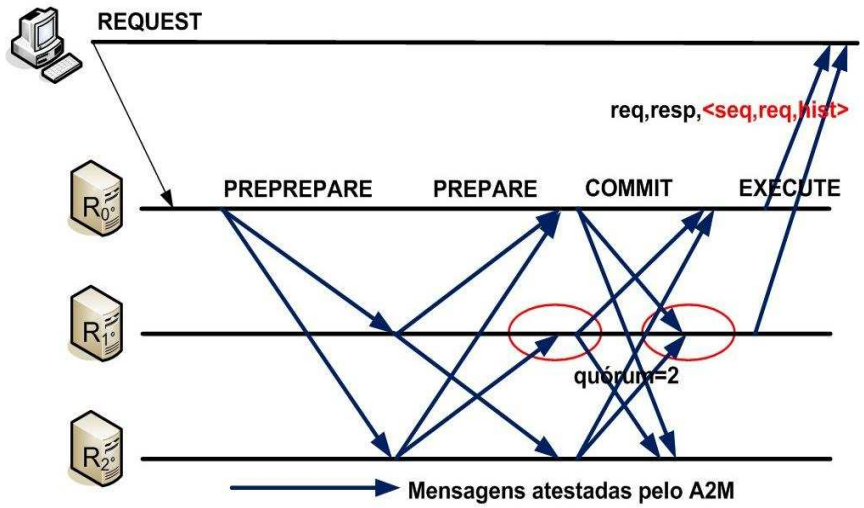


Figura 3.9: PBFT equipado com o A2M

Um caso particularmente interessante de uso do **A2M**, é a implementação do protocolo PBFT [65] em que as réplicas são equipadas com um *log* confiável para o armazenamento das mensagens trocadas entre elas durante todas as fases do protocolo (figura 3.9). O protocolo batizado de A2M-PBFT-EA é $2f + 1$ resistente, porém, com uma única limitação em que não é permitida a ordenação de requisições em lotes, isto é, só é possível ordenar uma requisição a cada *round* do protocolo. Esta limitação é necessária para assegurar que as mensagens são anexadas aos *logs* correspondentes do A2m, na ordem correspondente ao número de seqüencia atribuído pelo protocolo de ordenação. Para que a implementação do PBFT com $2f + 1$ réplicas seja possível, cada mensagem do protocolo é anexada ao respectivo *log* respeitando o número de seqüência atribuído pelo PBFT. Todas as mensagens geradas durante um *round* do protocolo de acordo/ordenação e de execução são anexadas ao *log* da respectiva réplica antes de ser enviado para as demais. Assim, em cada passo de comunicação de um *round*, antes de proceder com o envio de uma mensagem, a réplica realiza a atestação desta em seu respectivo *log*, e, uma vez que a mensagem é atestada isso faz com que a réplica seja forçada a não enviar, por exemplo, mensagens diferentes para um mesmo número de seqüência. As réplicas receptoras

por sua vez, verificam se a mensagem e a atestação recebidas combinam com as respectivas mensagens em seu poder (inclusive as geradas por ela), sem a necessidade de realizar qualquer tipo de comunicação com as demais réplicas. Esta capacidade de atestação provida pelo **A2M** permite que o protocolo A2M-PBFT-EA tolere até f faltas em um conjunto de $2f + 1$ réplicas [7], isto é, case $1/2 + 1$ réplicas forem corretas, o protocolo tem seu *liveness* e seu *safety* assegurados. Este feito se dá pelo fato de que, cada mensagem do protocolo de acordo (e ordenação total) é atestada pelo *log* restringindo a possibilidade de uma réplica faltosa enviar diferentes propostas para uma requisição enviada em um determinado *round* do protocolo. Isto faz com que uma réplica faltosa nunca consiga equivocar as demais réplicas do protocolo com mensagens não corretas, sacrificando o seu *safety*.

3.3 Conclusões do Capítulo

Este capítulo apresentou um estudo sobre o conceito de replicação em sistemas distribuídos no que concerne aos aspectos relacionados aos componentes de software, bem como as principais abordagens de replicação existentes na literatura. Além do mais, foram apresentados os principais trabalhos propostos até então, e que podem ser vistos como estado da arte no contexto de protocolos resistentes a faltas bizantinas. Estes trabalhos deram subsídios para a definição da arquitetura de replicação proposta neste trabalho e que será discutida no capítulo 5.

A partir deste estudo foi possível verificar que a grande vantagem em se replicar componentes de software em relação à replicação em nível de hardware é custo de adaptação e implementação. Em se tratando das técnicas de replicação apresentadas, vemos que a abordagem de replicação ativa incorre em menor sobrecarga no sistema em situações de falhas, visto que todas as réplicas são ativas e como conseqüência, o mascaramento da falha de uma réplica é quase que instantâneo. De outro lado, a abordagem de replicação passiva despende maior sobrecarga no sistema em situações de falhas, já que é necessário re-estabilizar o protocolo de replicação (e o estado do serviço) a partir da eleição de um novo líder, e em alguns casos também é necessária a recuperação do estado do líder, para que o serviço tenha continuidade a partir do ponto de falha. A única vantagem da abordagem passiva é que ela não tem como requisito o determinismo de réplicas, pois cabe ao líder impor seu estado

sobre os *backups* durante os períodos de sincronização das réplicas (*checkpoint*). Por fim, a abordagem ativa ganha no aspecto relacionado às classes de faltas toleradas, já que a partir desta abordagem é possível tolerar qualquer tipo de falta.

Em se tratando do estudo realizado sobre os protocolos de replicação com faltas bizantinas vemos que, de fato, todos compartilham as mesmas características quando se trata de execuções livres de faltas, isto é, eles propõem otimizações que são direcionadas aos casos onde há a ausência de faltas. Neste caso, o Zyzzyva [6] obtém o melhor desempenho dentre todos, quando todas as réplicas são corretas. Um fato também curioso é que a grande maioria dos algoritmos tolerantes a faltas bizantinas são descendentes (quase que diretos) do PBFT [65], já que alguns deles tentam otimizar os custos deste algoritmo por meio de delegação de tarefas, como é o caso da proposta de separação do acordo e execução [5], onde a tarefa que concerne à execução da aplicação é extraída das réplicas do PBFT, permitindo modularidade no sistema de replicação, mas aumentando o número de réplicas se visto por outra ótica. A proposta que realmente permite reduzir o custo de replicação é o A2M [7] que, por outro lado não permite a implementação de otimizações como ordenação de mensagens em lote (otimização que trás ganhos imensos de desempenho). O fato é que até o presente momento não se chegou a um consenso quanto ao "estado da arte" no que concerne replicação com faltas bizantinas, visto que cada protocolo tem seus prós e seus contras.

Capítulo 4

Fundamentos em Espaço de Tuplas

Nos últimos anos os sistemas distribuídos vêm se caracterizando como mais dinâmicos e heterogêneos, em face das evoluções ocorridas nas arquiteturas de hardware e de software. Estas características que se fazem presentes, principalmente em sistemas de larga escala (tal como a Internet), abrem precedentes para uma série de problemas que afetam a segurança e integridade das aplicações. Há autores que afirmam que os mecanismos adotados até então para as interações em sistemas distribuídos não serão mais adequados para os sistemas distribuídos do futuro, e que para suportá-los serão necessários novos modelos e paradigmas de programação [70]. Por outro lado, há autores que têm investigado e proposto novas taxonomias para o desenvolvimento de aplicações distribuídas, baseadas em modelos de coordenação já consolidados na literatura [14].

O paradigma de **coordenação** [15] evidencia que as atividades realizadas por um processo integrante de um sistema distribuído podem ser classificadas como coordenação e computação [71], dando ênfase para os aspectos relacionados à coordenação entre os processos. Para [71], a computação concerne às manipulações de dados e seu processamento pelos processos de aplicação, enquanto a coordenação diz respeito às interações entre os processos, e atua como elemento de confluência entre as entidades do sistema. Informalmente, a coordenação pode ser vista como a troca de informações entre as entidades cooperantes de uma aplicação distribuída [72].

No contexto de coordenação, o principal componente é o meio de coordenação, que por sua vez representa o sistema de comunicação subjacente usado pelos processos para rea-

lizar as interações. Na literatura são identificados quatro modelos de coordenação [14], que são definidos de acordo com o grau de acoplamento necessário para que as interações sejam bem sucedidas. Estes modelos de coordenação que são coordenação direta, coordenação orientada a encontro, coordenação quadro negro e coordenação generativa, são descritos junto aos respectivos graus de acoplamento na tabela 4.1.

Tabela 4.1: Classificação dos Modelos de Coordenação [14]

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

O modelo de coordenação **Direta** é o modelo de coordenação mais comum em aplicações distribuídas. Neste modelo a comunicação é realizada de forma direta por meio de interações fortemente acopladas e endereçadas, o que conseqüentemente implica em alguma forma de sincronismo entre as entidades. Exemplos de coordenação direta são comunicações via *Socket* TCP/IP e RPC. Este tipo de coordenação é típico de sistemas em ambientes de redes par-a-par e cliente-servidor. Por outro lado, o modelo **Orientado a Encontro** é aquele onde a comunicação entre os processos ocorre por meio de encontros espacialmente desacoplados, implementados por meio de canais públicos ou listas de distribuição de mensagens e/ou eventos. Este modelo de coordenação é acoplado temporalmente no sentido de que é necessário os processos se registrarem no canal para que possa ocorrer a comunicação. Cabe ressaltar que toda informação produzida no encontro é recebida por todas as entidades interessadas e presentes naquele momento. Exemplos de mecanismos que implementam este tipo de coordenação são as filas de mensagens e os serviços de eventos, além do mesmo ser amplamente difundido na comunicação de agentes móveis [73]. Já no modelo **Quadro Negro** as entidades realizam as interações por meio de espaços de dados compartilhados, que são usados como repositórios de mensagens. Estes espaços ficam localizados em determinados pontos/nós da rede. Este modelo é classificado como desacoplado temporalmente em virtude da possibilidade das entidades que desejam se comunicar escreverem e recuperarem informações no quadro a qualquer momento (i.e. duas entidades que se comunicam não precisam estar ativas no momento da comunicação). Contudo, este modelo é acoplado espacialmente pelo fato de que as mensagens colocadas no quadro devem ser identificadas no momento da manipulação (leitura/escrita), e isto implica que os processos que desejam ler uma determinada mensagem devem tomar conhecimento do identificador desta dentro do ambiente do quadro. O modelo **quadro negro** é largamente

empregado na coordenação de agentes de software. Por fim, o modelo **Generativo**, igualmente ao modelo **quadro negro** se apropria de um espaço de memória compartilhado para prover as comunicações. Neste modelo as informações são estruturadas na forma de tuplas, e o acesso a estas ocorre de forma associativa, através de combinações. O modelo **generativo** teve sua primeira implementação na linguagem LINDA [16], e, em virtude de suas características e flexibilidade, diversas implementações surgiram ao longo do tempo. Informações mais detalhadas sobre este modelo serão apresentadas nas próximas seções deste capítulo.

Dentre os modelos encontrados na literatura, o **modelo generativo** [16] ganha destaque entre os demais, por se mostrar como a alternativa mais viável ao modelo de comunicação tradicional de passagem de mensagens, em virtude de seu poder e sua simplicidade: é dotado de poucas (e simples) operações, e fornece um modelo de comunicação desacoplado tanto no tempo quanto no espaço (os processos não precisam estar ativos ao mesmo tempo nem tampouco conhecer os endereços uns dos outros para realizar uma interação [14]). Tendo em vista que o trabalho proposto nesta dissertação está intimamente fundamentado no modelo generativo (espaço de tuplas), o objetivo deste capítulo é descrever os conceitos fundamentais do modelo de coordenação generativa (modelo do qual o espaço de tuplas é oriundo), seu funcionamento e sua inserção no contexto de sistemas distribuídos. Também são apresentadas as principais implementações do referido modelo. Por fim é apresentado o modelo de comunicação (e sua implementação) adotado neste trabalho conhecido como PEATS, que por sua vez corresponde à adaptação do modelo de espaço de tuplas para o contexto de faltas bizantinas [18].

4.1 Modelo de Coordenação Generativa

O **modelo de coordenação generativa** [16], inicialmente definido como **modelo de comunicação por buffer global** [72] foi introduzido no contexto da linguagem Linda no intuito de fornecer um suporte para comunicações desacopladas no tempo e no espaço entre os processos paralelos/distribuídos. A concretização deste suporte de comunicação se deu a partir da definição de um espaço de memória compartilhada para prover estas interações desacopladas, ao qual foi dado o nome de **espaço de tuplas**. Neste espaço, os processos podem armazenar e recuperar estruturas de dados genéricas denominadas **tuplas** durante suas interações. Apesar do espaço de tuplas concretizar um sistema de memória compartilhada, isto não significa que este tipo de sistema é implementável somente em ambientes de memória

compartilhada. Pelo contrário, algumas das implementações o fazem a partir de uma rede de comunicação, constituindo um sistema de emulação de memória compartilhada através de troca de mensagens. No entanto, é valido ressaltar que as primeiras implementações do espaço de tuplas foram realizadas em ambientes de memória compartilhada distribuída [72, 16].

O elemento básico do modelo generativo (e do espaço de tuplas) é a **tupla**, que por sua vez consiste em uma seqüencia de campos, isto é, $t = \langle f_1, f_2, ..., f_n \rangle$. Cada campo f_i de t pode conter um valor definido (de um tipo de dados qualquer); poder ser um formal, que não contém o valor, mas somente a definição do tipo de dados do campo e é representado por uma variável precedida de um "?"; ou ainda pode ser um símbolo especial "*", que por sua vez é usado para representar qualquer valor de qualquer tipo de dados. Assim, uma tupla cujos campos têm valores e tipos definidos é denominada **entrada** (*entry*) e é representada por t. Por outro lado, uma tupla que dentre os seus campos possui algum formal"?" ou especial "*" é denominada **molde** (*template*), e é representada por $\bar{t}$. Esta distinção entre tuplas e moldes é necessária para a compreensão do funcionamento do espaço de tuplas, mais especificamente no que concerne à manipulação no espaço de tuplas, sendo que só é permitida a inserção de entradas, no entanto, o processo de recuperação de tuplas é feito com base nos moldes.

As manipulações no espaço de tuplas são realizadas por meio de invocações de três operações [16], as quais são ilustradas na figura 4.1.

- *out*(t): operação que realiza a inserção de uma tupla t no espaço de tuplas;
- *in*($\bar{t}$): operação que realiza a retirada da tupla t, que combina com o molde $\bar{t}$, do espaço de tuplas;
- *rd*($\bar{t}$): operação que realiza a leitura da tupla t (que combina com $\bar{t}$) sem retirá-la do espaço de tuplas.

Convém salientar que as operações de leitura e remoção (*rd* e *in*) são não-deterministas e bloqueantes. Desta forma, caso exista um conjunto de tuplas que combine com o molde especificado, qualquer uma delas pode ser escolhida como resposta da operação. Considerando o aspecto concernente ao bloqueio, caso no momento da execução de uma das operações *rd* ou *in* não exista nenhuma tupla t no espaço, que corresponda ao molde $\bar{t}$ fornecido na operação o processo permanece bloqueado até que uma tupla que combine com o molde seja

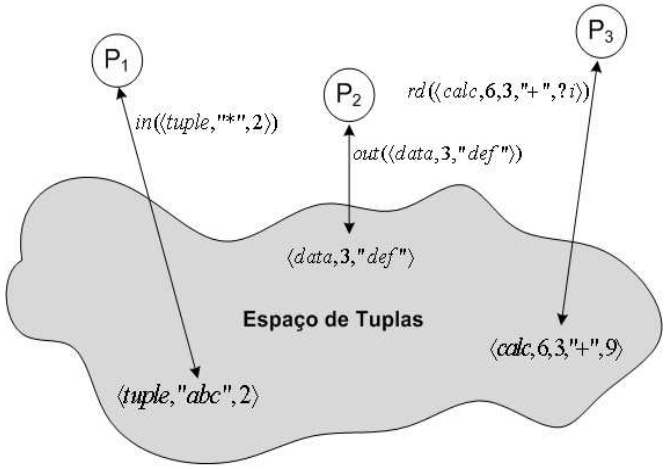


Figura 4.1: Interação de Processos no Espaço de Tuplas

inserida para que a operação seja completada. Além do mais, caso mais de um processo esteja bloqueado a espera de tuplas com as mesmas características (com moldes equivalentes), apenas um dos processos terá a operação completada (determinado de forma aleatória), e os demais continuarão bloqueados a espera da(s) tupla(s). Além destas operações básicas, variantes não bloqueantes das operações de leitura são também suportadas. Estas operações são denominadas *inp* e *rdp*, e possuem o mesmo comportamento de suas versões bloqueantes, exceto pelo fato delas retornarem mesmo quando não existe nenhuma tupla que combine com o molde fornecido (elas retornam um valor booleano que sinaliza se uma tupla foi encontrada ou não).

Para que seja possível a recuperação de tuplas por meio das operações de leitura e remoção (incluindo suas variantes não bloqueantes), o modelo de coordenação generativa define um mecanismo de combinação de tuplas, que por sua vez é formalizado a partir das seguintes premissas. Um campo f_i (formal ou definido) de uma tupla t tem um tipo associado, sendo o tipo denotado pela função $\tau : \mathcal{F} \rightarrow \mathcal{T}$, onde $\mathcal{F}$ é o conjunto de todos os possíveis campos (definidos ou formais), e $\mathcal{T}$ é o conjunto dos possíveis tipos de dados admitidos no modelo de computação [74]. Portanto, uma tupla $t = \langle f_1, f_2, ..., f_3 \rangle$ e um molde $\bar{t} = \langle f_1, f_2, ..., f_3 \rangle$ combinam, se e somente se ambas têm o mesmo número de campos, e se cada campo f_i da tupla e do molde são compatíveis, tal que $\forall i = 1..n : \bar{f_i} = * \vee \bar{f_i} = f_i \vee (formal(\bar{f_i}) \wedge \tau(f_i) = \tau(\bar{f_i}))$. Por exemplo, uma tupla $\langle$"$PPGIA''$, 2008$\rangle$ combina com os moldes $\langle$"$PPGIA''$, $*\rangle$, $\langle *, 2008\rangle$, $\langle *, *\rangle$, $\langle ?s, ?i\rangle$[1], $\langle ?s, 2008\rangle$ e $\langle$"$PPGIA''$, $?i\rangle$, mas não com $\langle *, 2007\rangle$, $\langle ?i, *\rangle$, $\langle *, ?s\rangle$ e $\langle *, *, *\rangle$.

[1]Neste caso *s* e *i* representam campos formais dos tipos *string* e *inteiro*, respectivamente

Um aspecto importante quanto ao funcionamento do espaço de tuplas, e que o difere dos demais modelos de coordenação é o acesso associativo, isto é, os dados/tuplas são recuperados por meio de seus conteúdos, e não pelos seus endereços, similar a linguagem SQL. Uma vez que a tupla é depositada no espaço, a mesma não pode mais ser alterada, a não ser por meio de sua remoção, seguida da inserção desta tupla, no entanto com parte de seu conteúdo modificado. Além do mais, a existência de uma tupla é totalmente independente do processo que a criou, em outros termos, se um processo p cria uma tupla t, e este processo é finalizado, a tupla t perdura no espaço até que algum outro processo efetue a sua retirada do mesmo (dai a origem do nome generativo).

4.2 Espaço de Tuplas Aumentado e Protegido por Políticas

O modelo clássico de coordenação generativa não define e/ou provê mecanismos para permitir que o espaço de tuplas seja capaz de coordenar processos sujeitos à faltas bizantinas. Visando atacar este problema, alguns trabalhos propuseram alternativas para construir espaços de tuplas tolerantes a faltas usando mecanismos de replicação e transações [75, 76], enquanto outros se concentraram nos aspectos relacionados a segurança [77]. No entanto, o problema da coordenação de processos bizantinos a partir de um espaço de tuplas foi estudado com mais ênfase em [18], onde os autores estenderam o modelo de coordenação generativa incluindo na especificação do espaço de tuplas uma operação de inserção condicional denominada *Conditional Atomic Swap*. Esta operação, que é denotada por $cas(\bar{t}, t)$ [78] recebe como argumentos um molde $\bar{t}$ e uma entrada t, e baseado nestes argumentos executa de forma indivisível o código:

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

Isto significa que, se a leitura do molde $\bar{t}$ falhar, então a inserção da tupla t é realizada no espaço, caso contrário, a tupla que combina com o molde $\bar{t}$ é retornada ao processo que invocou a operação. O uso da operação *cas* permite que o espaço de tuplas seja capaz de resolver o problema do consenso para um grupo de processos assíncronos, se comunicando por meio de memória compartilhada [78], além de possibilitar a implementação de protocolos de consenso para qualquer número de processos interagindo sobre o espaço, já que trabalhos prévios provaram que um espaço de tuplas só era capaz de resolver o consenso com no máximo dois processos [79, 78] interagindo sobre o espaço. A admissão da operação *cas* na

especificação do espaço de tuplas consolida o que é conhecido na literatura como “espaço de tuplas aumentado” [78]. Deste modo, o “espaço de tuplas aumentado” pode ser visto como uma extensão do espaço de tuplas introduzido no modelo generativo [16] que provê operações de inclusão, remoção, leitura e inclusão condicional de estruturas de dados em forma de tuplas.

Recentemente foi introduzido a noção de controle de acesso sobre objetos de memória compartilhada através do uso de políticas de acesso [17]. Dentro deste contexto está inserido o modelo de computação denominado PEATS (*Policy-Enforced Augmented Tuple Space*) [18], que por sua vez consiste em um modelo que provê uma abstração de memória compartilhada de espaço de tuplas, onde as interações entre os processos (ou acessos ao espaço) são reguladas por **políticas de acesso de granularidade fina**. Estas políticas, que são usadas como mecanismo de controle de acesso ao espaço de tuplas, são compostas por um conjunto de regras que são definidas por meio da especificação de padrões de invocação para operações no espaço de tuplas e condições (expressões lógicas) que devem ser satisfeitas para que estas invocações possam ser executadas, ou de outra forma negadas. Para permitir a execução de uma operação sobre o espaço, o PEATS considera os dados oriundos da invocação (o processo invocador e os parâmetros da invocação) e o estado do atual do espaço. No modelo PEATS, a cada operação permitida para execução no espaço de tuplas é associada uma lista dos processos que têm acesso a operação, e desta forma, somente os processos que têm acesso a determinada operação podem invocá-la. A proteção quanto aos acessos não autorizados é concretizado no PEATS por meio de um monitor de referência, que é responsável por verificar se a invocação de uma operação satisfaz a política de acesso definida para o PEATS. Uma observação importante é o fato de que as operações não especificadas na política têm sua execução negada por padrão.

O uso de um PEATS possibilita a construção de algoritmos tolerantes a faltas bizantinas muito mais simples e modulares, já que a restrição ao comportamento malicioso dos processos fica a cargo das políticas definidas para o sistema subjacente de comunicação e computação (neste caso, para o espaço de tuplas), e deste modo, retirando esta tarefa da especificação dos algoritmos. Para ilustrar a simplicidade dos algoritmos usando um PEATS, no algoritmo 1 é apresentada a resolução do consenso binário por meio da coordenação através do espaço de tuplas [18]. A política de acesso do PEATS é apresentada na figura 4.2.

Algoritmo 1 Consenso Binário entre N processos (processo p_i).

Variáveis compartilhadas:

1: $ts = \emptyset$ {PEATS, inicialmente vazio}

procedure propose(v)

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

O funcionamento deste algoritmo é bem simples, note que inicialmente o processo p_i insere sua proposta no espaço de tuplas ts a partir da inserção da tupla PROPOSE (linha 2). Em seguida p_i tenta continuamente ler as propostas emitidas pelos demais processos (linha 7), até encontrar um valor comum proposto por pelo menos $f + 1$ processos (linhas 5-11). Note que a idéia por trás das $f + 1$ proposta é a mesma empregada em sistemas tolerantes a faltas, o que indica que pelo menos um processo correto propôs um valor dentre os $f + 1$, já que até f processos podem falhar. Assim, o algoritmo continua e o primeiro valor a se enquadrar nesta quantidade é tomado como valor de decisão pelo processo p_i e então é inserido no espaço de tuplas através da operação *cas* (linha 10). O uso da operação *cas* é essencial no algoritmo, uma vez que outros processos podem coletar $f + 1$ propostas para diferentes valores, tentando-os tomar como valor de decisão. Neste caso, a inserção da tupla DECISION a partir da operação *cas* assegura que uma única decisão será tomada, já que as invocações posteriores retornarão o valor de decisão inserido pelo primeiro processo a chegar ao consenso (linhas 12-14).

Se tratando das regras, elas são descritas na notação da linguagem Prolog, que significa que as regras ao lado esquerdo são avaliadas como verdadeiras se e somente se as condições descritas à direita são verdadeiras (as regras e condições estão separadas por " $: -$ "). Note que o predicado *execute* indica se a operação deve ser executada ou não, e o predicado *invoke* avalia a operação invocada tem boa formação, isto é, se esta de acordo com o formato exigido pela sua regra. Mais especificamente para a política descrita na figura 4.2, são permitidas somente

as operações *rdp*, *out* e *cas*, controladas pelas regras R_{rdp}, R_{out} e R_{cas}, respectivamente. A regra R_{rdp} especifica que a operação *rdp* pode ser usada para qualquer tipo de tupla, isto é, não leva em consideração a formação, os campos, etc. Já a regra R_{out} especifica que cada processo pode incluir somente uma proposta (tupla PROPOSE) no espaço. Por fim, na regra R_{cas} é definido que a operação só pode ser executada se o segundo campo do molde fornecido for um formal (não contendo valor), e que a inserção só é permitida desde que o valor v passado como segundo argumento da tupla apareça nas propostas de pelo menos $f+1$ processos (previamente inseridos nas tuplas PROPOSE). A partir deste exemplo pode-se verificar que os processos bizantinos ficam limitados as operações definidas na política, e com isso algoritmos para problemas mais complexos se tornam mais simples.

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

Figura 4.2: Política de acesso do PEATS usado no algoritmo 1

4.3 DepSpace: Uma Implementação do PEATS

O DepSpace [80] é a primeira concretização de um espaço de tuplas tolerante a faltas bizantinas, admitindo também as propriedades do modelo PEATS. Mais especificamente, o DepSpace é uma implementação de um espaço de tuplas confiável e seguro (*dependable*) [10], e provê o suporte a todos os atributos de segurança de funcionamento aplicáveis ao contexto de espaço de tuplas desde que algumas premissas sejam satisfeitas no modelo de sistema. Os atributos de segurança de funcionamento pertinentes a espaços de tuplas são [81]:

- **Confiabilidade**: as operações efetuadas sobre o espaço de tuplas realizam a transição de seu estado conforme sua especificação;
- **Disponibilidade**: o espaço de tuplas sempre está apto a executar as operações requisitadas por entidades autorizadas;
- **Integridade**: o estado do espaço de tuplas só pode ser realizado por meio da execução correta de suas operações;

- **Confidencialidade**: o conteúdo (total ou parcial) de uma tupla só pode ser revelado à entidades autorizadas.

Para prover todos estes atributos, o DepSpace admite um conjunto de requisitos e mecanismos no ambiente de execução. O primeiro mecanismo empregado é a replicação máquina de estados [2], isto é, o espaço de tuplas mantém uma réplica de seu estado em pelo menos $3f + 1$ servidores, e assim permite que o sistema mantenha sua estabilidade desde que até f servidores apresentem comportamento de falha bizantina [12]. O algoritmo de acordo e coordenação de réplicas empregado no núcleo do DepSpace é o Paxos bizantino [37], e como este algoritmo é natureza determinista, se faz necessária a existência de um sistema de comunicação subjacente, bem como de um ambiente de computação/execução parcialmente síncrono [31] (síncrono terminal, evidenciando a existência de limites para as comunicações e computações). O uso de replicação está relacionado às propriedades de confiabilidade e disponibilidade, já que esta abordagem assegura a consistência linearizável [55] na qual todas as operações são executadas na mesma seqüencia em todas as replicas do sistema, e assim permite que o espaço de tuplas mantenha sua especificação mesmo que parte de suas réplicas sejam falhas, além de mascarar as falhas parciais no sistema. O funcionamento do algoritmo de replicação é bastante simples, o cliente envia a requisição da operação a partir do protocolo de difusão atômica (baseado no Paxos bizantino) a todas as réplicas, e aguarda por $f + 1$ respostas iguais de diferentes servidores para consolidar a operação. Uma vez que as operações são enviadas por meio de um protocolo de difusão atômica, e considerando que o espaço de tuplas é determinístico (em termos de estado e não das operações), sempre haverá um conjunto de $n - f \geq 2f + 1$ servidores corretos para executar a operação requisitada.

O atributo de confidencialidade é provisionado por meio de um esquema de **confiança distribuída**, mais especificamente pelo PVSS (*public verifiable secret sharing*) [82], tendo em vista que o espaço de tuplas é replicado e não se pode delegar esta atividade a uma única réplica, mas a todas elas (uma tupla não deve ser armazenada inteiramente em um único servidor). O processo de distribuição do segredo é delegado ao cliente, que o faz no momento da inserção da tupla, da seguite forma: o cliente cifra a tupla e em seguida gera um conjunto de n fragmentos e os distribui para cada um dos n servidores. Este tipo de esquema de confidencialidade é construído com base em dois números n e t, onde n corresponde ao número de fragmentos gerados para o segredo, e t é o número mínimo de fragmentos requeridos para

remontar o segredo. Neste caso, $t = (f + 1)$, o que indica que pelo menos um processo correto participará da remontagem do segredo, além de evitar que uma coalisão de servidores faltosos consiga obter o segredo, já que até f podem falhar simultaneamente (sendo $f < f + 1$). No entanto, um esquema de confidencialidade, caso não seja bem definido pode afetar o processo de combinação de tuplas e moldes, já que a comparação entre eles é necessária nas operações de leitura e exclusão. Deste modo, para permitir a combinação de tuplas sem revelar o conteúdo das mesmas, o DepSpace implementa um esquema baseado em *fingerprint* para cada campo da tupla, onde utiliza resumos criptográficos para possibilitar as comparações.

Por fim, a manutenção da integridade e parte da confidencialidade do DepSpace é realizada por meio de um mecanismo de controle de acesso ao espaço de tuplas. Este controle de acesso é usado para prevenir que clientes não autorizados obtenham acesso às tuplas, bem como para impedir que clientes com comportamento de falha saturem o espaço com tuplas espúrias. Para tanto, o DepSpace implementa o controle de acesso de em nível de espaço e em nível de tuplas: (*i*) **baseado em credenciais**, que consiste no controle em nível de tuplas onde para cada tupla inserida no espaço (opcionalmente) são definidas as credenciais necessárias para acessá-las, tanto em termos de leitura como de remoção. A implementação deste tipo de controle de acesso é feita a partir da associação de ACL's (listas de controle de acesso) a cada tupla, na qual são definidos os clientes que podem ler ou remover determinadas tuplas; (*ii*) **políticas de granularidade fina**, nas quais é baseado o controle de acesso em nível de espaço, isto é, o acesso ao espaço é controlado pelas políticas. O espaço de tuplas tem uma única política de acesso que é especificada no momento de sua criação, e, uma vez definida a política é ela que permite/nega o acesso ao espaço a partir da verificação de três parâmetros: o identificador do processo; a operação a ser executada e seus argumentos; e as tuplas atualmente armazenadas no espaço. Um exemplo de regra que pode ser associada a um espaço de tuplas seria "uma operação $out(\langle REQ, p, op, seq\rangle)$ só pode ser executada se o processo invocador é p **e** não existe nenhuma tupla $\langle REQ, q, x, seq\rangle$ no espaço **e** existe uma tupla $\langle REQ, r, y, seq-1\rangle$".

Para toda operação recebida por um servidor correto do DepSpace é realizada a verificação se esta satisfaz a política de acesso, sendo que a verificação consiste na avaliação de uma expressão lógica previamente especificada na regra de acesso da operação invocada. No caso de negação da operação, os servidores enviam um código de erro ao cliente, que por sua vez aceita-a caso receba $f + 1$ respostas com o mesmo código de erro. Como o

DepSpace é implementado em Java, as políticas de acesso são definidas em *scripts* descritos em Groovy [83] por esta ser amplamente suportada pela JVM. No momento da criação do espaço o *script* (que descreve uma operação de autorização para cada operação suportada pelo espaço) das políticas é enviado aos servidores como uma *string*, e quando do seu recebimento cada servidor transforma esta *string* em *bytecode* Java a partir da instanciação de um monitor de referência que verifica todas as operações que são requisitadas ao espaço.

A abstração de espaço de tuplas implementada pelo DepSpace é descrita por meio de uma interface denominada `DepSpace`, a qual fornece todas as operações suportadas pelo DepSpace. Esta interface é ilustrada na figura 4.3.

```
public interface DepSpace {

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

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

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

Figura 4.3: Interface do DepSpace

Como se pode ver na interface, em termos de implementação tanto as tuplas como os moldes são representados pela abstração `DepTuple`. O parâmetro `Context` representa o contexto da invocação da operação atual, e é passado do cliente para o servidor e vice-versa. Este contexto por sua vez encapsula, por exemplo, as credencias do cliente que são passadas entre as invocações das operações.

4.4 Conclusões do Capítulo

Este capítulo apresentou um estudo sobre o conceito de coordenação em sistemas distribuídos, envolvendo os tipos e modelos de coordenação existentes na literatura. Dentre os modelos existentes, neste capítulo foi dada maior ênfase ao modelo de coordenação generativa,

por este constituir parte fundamental de nossa proposta de trabalho. As características do modelo generativo são muito atraentes quando se considera a interação de processos em sistemas distribuídos.

Uma extensão particularmente interessante do modelo generativo, o PEATS também foi apresentada. O PEATS é a introdução do modelo de coordenação por espaço de tuplas em sistemas onde os processos estão sujeitos a faltas bizantinas. O estudo realizado sobre o modelo de coordenação generativa foi uma das grandes motivações para a proposta deste trabalho. A concretização do PEATS é o DepSpace, que consiste em um sistema que encapsula todos os requisitos necessários para a criação de espaços de tuplas tolerante a intrusões. O PEATS, bem como DepSpace são considerados fundamentais para a realização de nossa proposta, haja vista que os mesmos (modelo e concretização) são empregados como suporte/base para a proposta, fruto desta dissertação.

Capítulo 5

RePEATS - Uma Arquitetura para Replicação de Serviços Tolerante a Faltas Bizantinas em Espaço de Tuplas

Este capítulo trata de nossa proposta de trabalho, onde se apresenta a especificação de uma arquitetura denominada RePEATS [84], que provê uma solução alternativa (e inovadora) para replicação de serviços tolerantes a faltas bizantinas. A especificação desta arquitetura contempla desde a descrição dos protocolos empregados em sua construção, até a formalização de seus algoritmos e das respectivas provas. Para a construção desta arquitetura exploramos uma abstração de memória compartilhada distribuída, a qual é usada como alternativa ao paradigma de passagem de mensagens, isto é, toda a gestão de comunicação e coordenação dos protocolos é realizada por meio de uma memória compartilhada, e não por passagem de mensagens como geralmente acontece nos sistemas distribuídos tradicionais. Mais especificamente, nossa proposta realiza a tarefa de coordenação de réplicas por meio de um repositório de dados compartilhado conhecido na literatura como espaço de tuplas [15], onde a coordenação orientada à dados provida pelo espaço de tuplas permite que seja efetuado o armazenamento das informações comuns compartilhadas pelas replicas, tal como *checkpoints*, *logs*, estados consistentes, herdando todas as propriedades do modelo de espaço de tuplas adotado na proposta, bem como suas características de desacoplamento espacial e temporal. O modelo de espaço de tuplas adotado é o PEATS [18], que é dotado de segurança de funcionamento (*dependability*) e provê um sistema subjacente de comunicação e coordenação tolerante a faltas e intrusões.

5.1 Contextualização e Motivação

O mecanismo fundamental para a construção de sistemas tolerantes a faltas é a replicação, sendo que para o tipo de faltas mais severa tal como a bizantina [12], a técnica comumente empregada é a replicação máquina de estados [1, 2]. No entanto, uma das grandes dificuldades da implementação de uma arquitetura para replicação tolerante a faltas bizantinas reside no suporte que implementa os protocolos de acordo para difusão atômica de mensagens, que é um requisito fundamental para assegurar a semântica de linearilidade [55] (ou consistência seqüencial) do sistema replicado. Nos últimos tempos, diversos trabalhos têm produzido protocolos e arquiteturas de replicação com foco à viabilidade prática de implementação de sistemas tolerantes a faltas [65, 5], e demonstrando inclusive a possibilidade de aplicar tais soluções em serviços reais. A grande questão por trás destas soluções, é que elas envolvem alto custo de software e hardware, pois os algoritmos têm funcionamento restrito para até f faltas de um total de $3f + 1$ réplicas, já que as cabe as réplicas implementar o protocolo de acordo e a aplicação/serviço.

Recentemente foi introduzido o conceito de objetos de memória compartilhada protegidos por políticas [17], que controlam as operações que podem ser realizadas sobre tais objetos por meio de políticas de granularidade fina. Estudos mostram que o uso destes objetos possibilita a construção de algoritmos tolerantes a faltas bizantinas muito mais simples, já que o comportamento das entidades maliciosas é coibido pelas políticas existentes nestes objetos, o que favorece em muito o uso desta abordagem para a implementação de aplicações que requerem um elevado grau de confiabilidade.

A proposta deste trabalho é empregar o uso destes objetos protegidos por políticas na construção de sistemas/serviços replicados. A arquitetura fruto de nossa proposta tem como alicerce o espaço de tuplas protegido por políticas descrito no capítulo 4, e define componentes de suporte à replicação, tais como *checkpointing*, *logging* estável e difusão atômica, todos coordenados por meio do espaço de tuplas. Estes componentes em conjunto com as respectivas políticas de acesso, permitem que uma aplicação seja capaz de tolerar faltas bizantinas mesmo que f partes de um total de $2f + 1$ sejam afetadas[1].

[1] Número de réplicas requerido para replicação com faltas bizantinas [67]

5.2 RePEATS - *Replication Over Policy-Enforced Augmented Tuple Space*

5.2.1 Visão Geral do RePEATS

A proposta do RePEATS [84] é fundamentada na otimização dos recursos computacionais empregados em um ambiente replicado, cujo objetivo é prover o maior aproveitamento dos recursos por parte das aplicações, sem ferir as propriedades que asseguram a consistência e a resistência a faltas bizantinas do sistema. Para tanto, propomos a concretização de um protocolo de replicação **Máquina de Estados** [2] que emprega como elemento de comunicação e coordenação entre os processos (réplicas e clientes) um PEATS, e assim dando origem a uma arquitetura de suporte a replicação tolerante a faltas bizantinas. O protocolo de replicação do RePEATS é constituído por dois sub-protocolos, sendo estes os de ordenação e *checkpointing*, e alguns mecanismos de suporte a replicação tais como *logging* persistente e estável, controle de acesso, entre outros.

O RePEATS suporta falhas parciais, no sentido que todos seus componentes podem falhar de forma arbitrária e ainda assim o sistema continua a se portar como esperado. Este feito é garantido através da combinação de técnicas de tolerância a faltas e de segurança. A disponibilidade, integridade e confiabilidade do espaço de tuplas são garantidas pela replicação do mesmo. A diferença fundamental do RePEATS em relação aos demais trabalhos BFT reside no fato de que procuramos prover um suporte de replicação tolerante a faltas bizantinas através da combinação de um conjunto de técnicas e mecanismos de tolerância a faltas, tendo como núcleo (e alicerce) um espaço de tuplas, que por sua vez provê todo o suporte à comunicação e aos mecanismos de tolerância a faltas. Os trabalhos desenvolvido neste âmbito até então [65, 5] propuseram soluções baseadas no paradigma de trocas de mensagens e com outras abordagens que visam a otimização, mas ainda incidindo em um custo razoável para sua adoção.

Em nossa abordagem, um PEATS [18] fornece o suporte de comunicação tolerante a intrusões, para as interações entre os clientes e as réplicas de um serviço, isto é, os processos (clientes e servidores) se coordenam através de um PEATS. Além do mais, a adoção do PEATS proporciona ao sistema um ambiente de armazenamento estável e seguro para todas as informações compartilhadas entre as réplicas. A figura 5.1 ilustra de forma resumida o

funcionamento do RePEATS, onde os clientes inserem suas requisições na forma de tuplas no PEATS (passo 1) e as réplicas do serviço que está sendo acessado lêem estas tuplas do PEATS para obter as requisições a serem executadas (passo 2). Em seguida, os serviços replicados processam as requisições e enviam os resultados também na forma de tuplas no PEATS (passo 3), para que os clientes possam obter as respostas (passo 4).

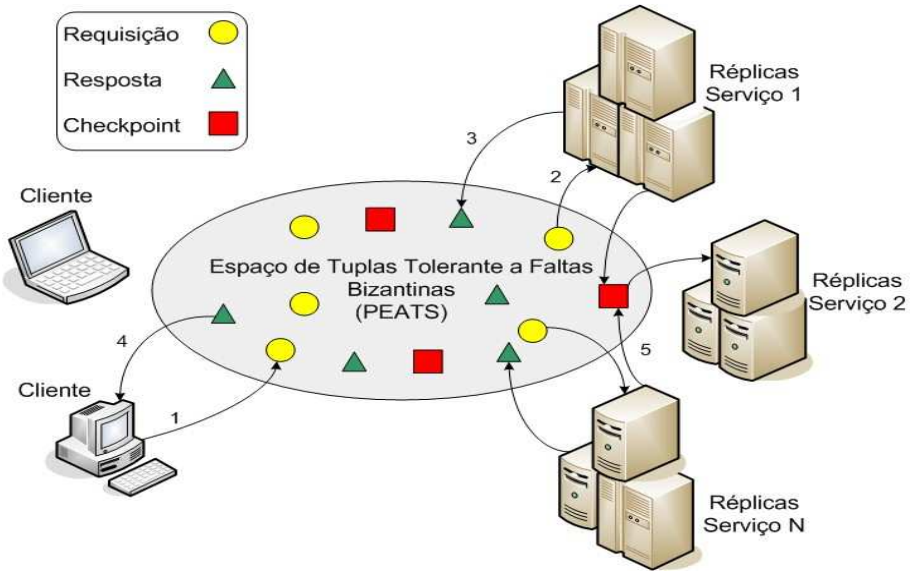


Figura 5.1: Modelo de execução do RePEATS

O RePEATS pode ser visto como uma proposta para a separação do acordo e execução [5] em replicação tolerante a faltas bizantinas, com foco à otimização dos recursos por meio de uma camada de acordo implementada sobre um espaço de tuplas confiável e seguro. O protocolo de ordenação de requisições é implementado sobre o espaço de tuplas, e suas propriedades asseguram que as requisições cheguem ordenadas aos servidores de aplicação (réplicas do serviço), requerendo somente $2f + 1$ réplicas de máquina de estados para um serviço bizantino [67]. O modelo adotado no RePEATS é genérico o suficiente para comportar diversos conjuntos de serviços (diferentes aplicações) partilhando o mesmo suporte de comunicação e coordenação. Em face deste fato a arquitetura RePEATS pode ser considerada como de resistência ótima, já que para n serviços são requeridas $n(2f + 1)$ réplicas de aplicação, a despeito dos protocolos PBFT e Zyzzyva que para os mesmos n serviços requerem $n(3f + 1)$ réplicas.

Uma premissa fundamental para o funcionamento do RePEATS é o **determinismo de réplica** [2]. Este requisito define que réplicas partindo de um mesmo estado inicial e sujeitas a execução de uma mesma seqüência de operações, devem chegar ao mesmo estado final. Assim, em um sistema onde as réplicas implementam um serviço determinista, esta propriedade é implementada por meio do uso de protocolos de **difusão com ordem total** [60], que garantem que todas as operações enviadas ao sistema são processadas por todas as réplicas (acordo) em uma mesma ordem (ordem total). A partir daí, cada réplica executa a operação, atualiza seu estado (quando há necessidade) e envia ao cliente o resultado obtido. O cliente aceita o resultado da operação caso receba $f + 1$ respostas iguais de diferentes réplicas, considerando f o número máximo de servidores que podem sofrer faltas bizantinas. No RePEATS, o algoritmo de difusão com ordem total é implementado através do PEATS.

5.2.2 Propriedades

Em geral, os sistemas distribuídos são especificados considerando as noções de segurança/correção (*safety*) e vivacidade/progresso/terminação (*liveness*). No RePEATS, algumas propriedades devem ser satisfeitas para que o sistema seja mantenha seu *liveness* e seu *safety*, são elas:

- **Ordem total:** as requisições são executadas com ordem total por todas as réplicas corretas do sistema;
- **Liberdade de bloqueio (*lock-freedom*):** em qualquer execução do sistema, se um processo correto envia uma requisição para execução onde existam requisições pendentes de execução, alguma requisição será executada.

Além das propriedades descritas, um serviço replicado usando o RePEATS apresenta um comportamento equivalente a sua implementação em um sistema não replicado, isto é, satisfaz o modelo de consistência conhecido por **linearilidade** [55] (escalonamento seqüencial das operações).

5.2.3 Controle de Acesso

Um mecanismo fundamental para que os algoritmos do RePEATS tolerem o comportamentos maliciosos dos processos é o controle de acesso provido pelas políticas de granula-

ridade fina suportadas pelo PEATS [17]. Este controle acontece através da implementação de políticas de segurança associadas ao espaço de tuplas, de modo que, quando uma operação é invocada no PEATS, as regras especificadas nestas políticas são verificadas tomando como base o identificador do processo que invoca a operação, a operação que está sendo invocada (e seus argumentos) e o estado atual do espaço de tuplas para negar ou permitir a execução da operação. Um exemplo de regra que pode ser associada a um espaço de tuplas é "uma operação $out(\langle REQ, p, op, seq\rangle)$ só poderá ser executada se o processo invocador for p **e** não existir nenhuma tupla $\langle REQ, q, x, seq\rangle$ no espaço **e** existir uma tupla $\langle REQ, r, y, seq-1\rangle$". Uma característica fundamental do modelo de controle de acesso do PEATS é que as operações não especificadas na política têm sua execução negada por padrão. Neste trabalho apresentamos as políticas de acesso seguindo o mesmo formato usado em [17].

5.2.4 Fila de Mensagens e Ordenação de Requisições

Na arquitetura proposta, o algoritmo de difusão com ordem total é implementado através do PEATS, onde ao contrário das abordagens já existentes na literatura [65, 5], o protocolo de ordenação total do RePEATS é concretizado por meio de uma fila de mensagens que é construída sobre o espaço de tuplas. Esta fila de mensagens em conjunto com políticas de acesso adequadas permite assegurar:

- Mensagens entregues na mesma ordem a todos os processos (réplicas);
- Mensagens entregues a todos os processos mesmo na ocorrência de faltas e recuperações posteriores;
- Mensagens persistentes.

Estas características das filas de mensagens mantidas pelo RePEATS são em sua essência muito similares as características dos protocolos de comunicação em grupo [85], no entanto, na arquitetura RePEATS a ordem de entrega das mensagens é determinada e controlada exclusivamente pelo estado da aplicação/réplica, e não por meio de trocas de visões como ocorre usualmente nos protocolos de comunicação em grupo. A capacidade de persistência de mensagens provida pelas filas de mensagens do RePEATS, também permite concretizar um mecanismo de *logging* de recuperação mais eficiente (explicado em maiores detalhes na seção 5.3.6), onde no caso da falha e posterior recuperação de uma réplica não é necessário

realizar um acordo explícito entre as réplicas (para recuperar e determinar a seqüencia de execução das mensagens que ainda não são estáveis), já que todas as mensagens que circundam o sistema são persistentes e estão disponíveis na fila. Esta facilidade só é possível em face do fato de empregarmos a memória compartilhada, e das propriedades da abstração de **replicação ativa** [1] (ou máquina de estados). Como a aplicação/réplica é modelada como uma máquina de estados determinista, as ações/interações com a aplicação são modeladas como transições da máquina de estados, desta forma, as mensagens são recuperadas na ordem já estabelecida na fila, fazendo com que a réplica em recuperação seja restabelecida ao mesmo ponto/estado em se encontram as demais réplicas do sistema.

É sabido que a difusão com ordem total é um requisito fundamental para replicação Máquina de Estados para que sejam asseguradas as propriedades que concernem ao determinismo de réplicas. Uma vez que todas as comunicações ocorrem através do PEATS, as mensagens depositadas no espaço (na memória compartilhada) ficam disponíveis no mesmo instante para todos os processos que têm acesso ao espaço. Deste modo, ao invés de executar um protocolo de acordo de diversas fases para ordenar as mensagens, a ordenação total das requisições é feita a partir de um algoritmo de seqüenciamento de mensagens implementado sobre o espaço de tuplas, o qual constrói uma fila virtual de mensagens persistentes sobre este espaço, seguindo a mesma abordagem usada para a implementação de um sistema de comunicação em grupo construído sobre o sistema SINFONIA [86]. Neste algoritmo, as requisições são inseridas no espaço juntamente com um número de seqüencia único, em tuplas. Estes números de seqüencia estabelecem e indicam a ordem na qual as requisições devem ser executadas por todas as réplicas de um serviço. A formalização deste protocolo é apresentada no algoritmo 2. É valido ressaltar que este algoritmo, bem como os outros que suportam a arquitetura proposta, são executados sobre um PEATS, e têm como alicerce as políticas de controle de acesso que impedem processos maliciosos de afetar a corretude do esquema de replicação. O uso do número seqüencial da tupla/requisição é necessário porque não é possível determinar a ordem de chegada das mensagens pelo espaço de tuplas, assim este número de seqüencia é usado como identificador de envio que indica a posição na qual a mensagem foi inserida na fila de mensagens (similar a uma ordem de envio), e respectivamente indica a ordem na qual ela deve ser entregue pelas réplicas.

O uso da política se faz necessário em face do fato de admitirmos a possibilidade de faltas bizantinas no ambiente, já que não é possível confiar na boa conduta dos processos clientes (i.e. sem um meio de proteção e verificação não é possível determinar se as requisições que chegam ao serviço têm boa formação e/ou estão corretas). Cabe ressaltar que a política não garante a boa conduta dos processos, no entanto, ela evita que mensagens inválidas sejam inseridas no espaço por estes processos.

Para que o REPEATS possa assegurar a correção do protocolo de ordenação, algumas premissas são admitidas:

1. Cada requisição de um cliente tem um identificador único e crescente;
2. Um cliente só envia uma requisição após ter recebido a resposta da requisição anterior[2];
3. A cada requisição enviada o cliente associa um temporizador, e caso este venha a expirar e a resposta ainda não tenha sido obtida, o cliente reenvia a requisição;
4. Cada réplica mantém um mecanismo de retenção de resultados [87] que armazena a resposta da última requisição (ou últimas k requisições) de cada cliente, este *buffer* de respostas é usado para evitar o reprocessamento das requisições que por ventura são reenviadas, além que permitir o restabelecimento pontual do estado de uma réplica, que eventualmente ficou atrasada em relação às demais.

O funcionamento do algoritmo executado pelo cliente (linhas 2-8) é bastante simples. Quando o cliente tem de enviar um comando C, primeiramente ele incrementa o identificador de sua requisição (inicialmente 0 – linha 3) e então começa a tentar colocar uma tupla REQUEST no espaço com um número de seqüencia uma unidade maior que o número de seqüencia que indica a posição final da fila (linha 6), através de uma invocação a operação *cas*, que verifica se existe uma tupla REQUEST com o número de seqüencia em questão no espaço e, em não havendo, insere a tupla pretendida (linha 7). Quando não é bem sucedido nesta inserção, o cliente faz uma nova tentativa incrementando o número de seqüencia da fila (linhas 5-7). Ao ser bem sucedido na inserção da tupla, o cliente é bloqueado na função *wait_response* (linha 8), onde aguardará por $f + 1$ respostas iguais advindas de diferentes servidores. A função *find_tail* (linha 6) serve para evitar que um cliente que tenha iniciado sua execução após

[2]Note que esta limitação pode ser relaxada para k requisições caso os servidores tenham a capacidade de armazenar as últimas k respostas de cada cliente.

Algoritmo 2 Algoritmo de Ordenação de Requisições (cliente p_i e servidor s_i).

Variáveis compartilhadas:

1: $ts = \emptyset$ {espaço de tuplas}

{Parte Cliente}
Variáveis locais:

2: $seqno = 0$ {indica a posição final da fila}
3: $reqid = 0$ {número da ultima requisição enviada}

procedure *execute*(C)

4: $reqid \leftarrow reqid + 1$
5: **repeat**
6: $\quad seqno \leftarrow find\_tail() + 1$
7: **until** $ts.cas(\langle REQUEST, seqno, *, *, * \rangle, \langle REQUEST, seqno, p_i, reqid, C \rangle)$
8: **return** *wait_response*($reqid$)

{Parte Servidor}
Variáveis locais:

9: $seqno = 0$ {última posição da fila}
10: $client\_data = \emptyset$ {buffer de retenção de resultados}
11: $req\_ckpt = \perp$ {número da maior requisição contida no último checkpoint}

tarefa principal

12: **loop**
13: $\quad$ **if** $req\_ckpt > seqno$ **then**
14: $\quad\quad ts.rd(\langle CKPT, ?ckptno, ?tail, ?state, ?buffer \rangle)$
15: $\quad\quad local\_state \leftarrow state$
16: $\quad\quad client\_data \leftarrow buffer$
17: $\quad\quad seqno \leftarrow tail$
18: $\quad$ **end if**
19: $\quad seqno \leftarrow seqno + 1$
20: $\quad ts.rd(\langle REQUEST, seqno, ?client_i, ?reqid, ?command \rangle)$
21: $\quad$ **if** $\nexists client_i \in client\_data$ **then**
22: $\quad\quad response \leftarrow local\_execute(command)$
23: $\quad$ **else**
24: $\quad\quad \langle reply, last\_reqid \rangle \leftarrow get\_client\_data(client_i)$
25: $\quad\quad$ **switch** $reqid$ **do**
26: $\quad\quad\quad$ **case** $last\_reqid + 1$
27: $\quad\quad\quad\quad response \leftarrow local\_execute(command)$
28: $\quad\quad\quad$ **case** $last\_reqid$
29: $\quad\quad\quad\quad response \leftarrow reply$
30: $\quad\quad\quad$ **otherwise**
31: $\quad\quad\quad\quad continue$
32: $\quad\quad\quad$ **end case**
33: $\quad\quad$ **end switch**
34: $\quad$ **end if**
35: $\quad update\_client\_data(client_i, reqid, response)$
36: $\quad send\_response(client_i, response)$
37: **end loop**

já haver um número grande de requisições no espaço tenha de iterar muitas vezes pelo laço das linhas 5-7 (cada vez invocando uma operação no espaço de tuplas). Esta função implementa uma estratégia que inspeciona o espaço de tuplas e tenta encontrar um número de seqüencia

mais próximo daquele presente na última requisição inserida no espaço, ao final da fila de requisições. A existência desta função não interfere na correção do algoritmo, mas melhora em muito o desempenho de clientes "atrasados".

Uma possível estratégia para implementação dessa função seria verificar qual o *checkpoint* mais novo do sistema e retornar o número da última requisição contida neste *checkpoint* (ver seção 5.2.5). Outras possíveis estratégias para a implementação desta função podem ser utilizadas tirando-se proveito de funcionalidades especiais que uma determinada implementação do PEATS pode implementar (ex. suporte a operações que lêem um conjunto de tuplas ou a capacidade de se especificar se o resultado de uma leitura deve pegar a tupla mais nova ou mais velha do espaço que combina com o molde fornecido).

O algoritmo dos servidores (linhas 9-37) é inicializado com o número de seqüencia da fila igual 0 (linha 9). Primeiramente, a cada iteração da tarefa principal a réplica verifica se está com o estado desatualizado em relação às demais réplicas (linha 13), e neste caso ela atualiza seu estado a partir do último *checkpoint* válido, note que isso pode ocorrer em virtude da réplica ter demorado muito a processar a última requisição, e com isso tendo ficado atrasada em relação ao grupo de réplicas. As réplicas processam os comandos contidos nas tuplas REQUEST em ordem ascendente primeiro obtendo a tupla com o número de seqüencia uma unidade maior que o da última requisição executada (linhas 19-20) que indica a posição atual da fila. Uma vez obtida a requisição a réplica verifica se há alguma informação para o cliente que emitiu a requisição no *buffer* de respostas, e caso não exista nenhuma informação para o cliente (indicando que é a sua primeira requisição) a réplica realiza a execução do comando contido nesta tupla através da função *local_execute* (linha 22), atualiza a tabela de respostas com o resultado da operação e envia a resposta ao cliente, cuja identidade está indicada na tupla REQUEST, através da função *send_response* (linhas 35-36). Por razões de desempenho, esta função envia uma mensagem com a resposta **diretamente ao cliente**, sem utilizar o PEATS. Caso exista uma entrada para o cliente no *buffer* de respostas, a réplica inicia um processo de checagem da consistência da requisição, onde primeiramente é verificado se a requisição recebida é uma unidade maior que a última requisição processada para o cliente em questão (linha 26), e neste caso processa a requisição (linha 27) enviando em seguida a resposta diretamente ao cliente (linhas 35-36). No caso da réplica constatar que o número da requisição recebida é igual ao da última requisição processada para o respectivo cliente (linha 28), o algoritmo trata o evento com

uma re-transmissão, e neste caso, como a réplica obteve os dados da requisição a partir do *buffer* de respostas (linha 24), ela envia o resultado diretamente ao cliente (linha 35-36). Por fim, se nenhuma das condições forem avaliadas como verdadeiras (linhas 26 e 28), a réplica descarta a requisição e avança a iteração do laço para recuperar a próxima requisição da fila de mensagens.

A política de acesso do PEATS para o algoritmo 2 é descrita na figura 5.2. A principal finalidade das regras desta política é evitar que processos maliciosos quebrem a ordem total de execução, inserindo tuplas REQUEST fora do intervalo de seqüencia no espaço (na fila). Para isso, a inclusão de requisições (tupla REQUEST) só pode ser feita através da instrução *cas*, na condição de que a tupla REQUEST com número seqüencial anterior ao que está sendo incluído esteja presente no espaço, ou então, caso a tupla correspondente a anterior tenha sido removida (ver coleta de lixo na seção 5.2.7), é verificado se ela já existiu em algum momento a partir do último *checkpoint* válido do sistema. Além disso, a política especifica que a operação *rd* é permitida somente para tuplas REQUEST, desde que os campos 3, 4, e 5 do molde sejam formais.

Object State TS

R_{rd}: $execute(rd(\langle REQUEST, seqno, x, y, z\rangle)) \ :-$
$\quad invoke(p, rd(\langle REQUEST, seqno, x, y, z\rangle)) \wedge formal(x) \wedge formal(y) \wedge formal(z)$

R_{cas}: $execute(cas(\langle REQUEST, seqno, x, y, z\rangle, \langle REQUEST, seqno, p, reqid, inv\rangle)) \ :-$
$\quad invoke(p, cas(\langle REQUEST, seqno, x, y, z\rangle, \langle REQUEST, seqno, p, reqid, inv\rangle)) \wedge$
$\quad (seqno = 1 \wedge ((\nexists\, i, j, k, l : \langle REQUEST, i, j, k, l\rangle \in TS) \wedge$
$\qquad \nexists\, n, o, p, q : \langle CKPT, n, o, p, q\rangle \in TS)) \vee$
$\quad (\exists\, i, j, k, l : \langle REQUEST, seqno - 1, i, j, k, l\rangle \in TS \vee$
$\qquad \exists\, m, n : \langle CKPT, ckptnum, seqno - 1, m, n\rangle \in TS))$

R_{in}: $execute(in(\langle REPLY, p_i, reqid, x, y\rangle)) \ :-$
$\quad invoke(p, in(\langle REPLY, x, reqid, y, z\rangle)) \wedge (x = p) \wedge formal(y) \wedge formal(z)$

Figura 5.2: Política de acesso para o PEATS usado no algoritmo 2.

5.2.5 *Checkpointing* e Transferência de Estados

Em um sistema de replicação a transferência de estados pode ser necessária em determinadas situações. Em geral esta funcionalidade é usada para permitir que as réplicas sejam sincronizadas quando alguma delas perde uma ou mais requisições por questões de desempenho, ou devido a falhas em nível de comunicação etc (i.e. recebeu a requisição n, mas não a $n-1$). Um meio simples, no entanto, muito ineficiente para transferência de estados é a

réplica atrasada solicitar o estado para todas as demais réplicas, estado este que consiste nas requisições perdidas e não processadas pela réplica em atraso (já que cada réplica armazena todas as requisições recebidas em um *logging*). Neste caso a réplica atrasada aguarda por $f + 1$ respostas (estados) iguais das demais réplicas. Uma abordagem mais otimista para este problema seria evitar que todas as réplicas corretas enviassem o estado requisitado, mas sim, que uma única réplica o fizesse e as demais enviassem somente o resumo criptográfico do estado. Deste modo, a validação do estado seria feita a partir dos $f + 1$ resumos. Entretanto, outro problema que concerne à transferência de estados é observado, onde no caso da réplica ficar atrasada por muito tempo o estado a ser recuperado pode ser muito grande, tendendo a causar uma perda de desempenho no sistema.

O RePEATS emprega uma estratégia baseada em *checkpoints* coordenados [88] para permitir a recuperação de réplicas faltosas, para estabilizar o serviço no caso de atrasos, e também para fins de coleta de lixo (ver próxima seção). Como o RePEATS é concretizado sobre uma abstração de memória compartilhada, a garantia da consistência dos *checkpoints* é de fundamental importância para a realização da coleta de lixo, haja vista que a memória do espaço de tuplas é finita (note que o algoritmo da seção anterior mantém todas as requisições dos clientes no espaço de tuplas). O algoritmo descrito nesta seção pode ser usado na produção de *checkpoints* em diferentes níveis (completos, incrementais e acumulativos).

A idéia fundamental do algoritmo de *checkpointing* é regularmente guardar o estado completo[3] das réplicas corretas do serviço (que será o mesmo) no espaço de tuplas, isto é, a cada N requisições executadas, sendo N um número configurado igualmente em todas as réplicas corretas do serviço por meio do parâmetro *CP_INTERVAL*. Note que o uso de uma estratégia de *checkpoint* reduz substancialmente o custo da transferência de estados entre as réplicas, já que, ao invés de enviar os *logs* parciais das requisições executadas, é enviado somente o estado contido no *checkpoint* e os *logs* posteriores a ele.

O estado de cada réplica é constituído pelo estado da aplicação (i.e. arquivos, metadados), um identificador numérico da última requisição executada (e ordenada) e uma tabela que contém informações (incluindo a resposta) da última requisição processada de cada cliente,

[3]Note que, dependendo das características do serviço implementado é possível haver um estado transiente e um estado persistente.

sendo esta tabela usada também como mecanismo de retenção de resultados [87] para prevenir a execução de requisições duplicadas ou re-enviadas, bem como para assegurar a recuperação pontual de uma réplica atrasada ou faltosa. A admissão do *buffer* de resultados/respostas no estado da réplica também se faz necessário para assegurar a correção do protocolo de replicação do REPEATS em virtude da estratégia de coleta de lixo (ver próxima seção) empregada no sistema.

No entanto, é importante salientar que para concretizar uma estratégia de *checkpointing* alguns cuidados devem ser tomados para se evitar que réplicas maliciosas criem *checkpoints* com estados incorretos do serviço. O protocolo de *checkpointing* executado no REPEATS é descrito no algoritmo 3.

Algoritmo 3 Algoritmo de *Checkpoint* (processo servidor s_i)

{Servidor}
Variáveis compartilhadas:
1: $ts = \emptyset$ {espaço de tuplas}
Variáveis locais:
2: $ckptnum = 0$ {número de seqüência do *checkpoint*}
procedure checkpoint(*seqno*)
3: **if** $ckptnum > 0$ **then**
4: **wait until** : $ts.rdp(\langle CKPT, ckptnum, *, *, * \rangle)$ {aguarda até completar o *checkpoint* anterior}
5: **end if**
6: $ckptnum \leftarrow ckptnum + 1$
7: $state \leftarrow snapshot()$ {estado atual da réplica e da aplicação}
8: $hash \leftarrow D(state)$ {D é a função que computa o Digest (Hash)}
9: $ts.out(\langle CKPTPROP, p_i, ckptnum, seqno, hash \rangle)$
10: $RemainingServers \leftarrow S \setminus \{s_i\}$ {inicialmente $2f$ servidores}
11: **while** $|RemainingServers| > f$ **do**
12: **for all** $s_j \in RemainingServers$ **do**
13: **if** $ts.rdp(\langle CKPTPROP, ?s_j, ckptnum, seqno, hash \rangle)$ **then**
14: $RemainingServers \leftarrow RemainingServers \setminus \{s_j\}$
15: **end if**
16: **end for**
17: **end while**
18: $ts.cas(\langle CKPT, ckptnum, seqno, *, * \rangle, \langle CKPT, ckptnum, seqno, state, hash \rangle)$

O algoritmo de *checkpoint* é iniciado logo após a réplica processar uma requisição cujo número de seqüencia é múltiplo do parâmetro *CP_INTERVAL* ($reqnum$ **mod** $CP\_INTERVAL = 0$). Este algoritmo recebe como argumento o número da última requisição executada pelo servidor. Ao iniciar o algoritmo, primeiramente a réplica verifica se o *checkpoint* em questão é o primeiro a ser gerado por ela, e neste caso prossegue com a geração. Caso uma instância

do algoritmo de *checkpoint* já tenha sido executada pela réplica em questão (i.e $ckptnum > 0$ linha 3) ela verifica se ele já é estável a partir da verificação da existência da tupla CKPT para o número de checkpoint atual, do contrário, a réplica aguarda até que o *checkpoint* anterior se torne estável, bloqueando o processamento de novas requisições (note que esta situação pode ocorrer quando uma réplica é correta mas está bem atrasada em relação às demais). Uma vez que o *checkpoint* anterior se torna estável o algoritmo prossegue e a réplica incrementa o número do *checkpoint*, pega uma imagem de seu estado atual que é constituído pelo estado da aplicação, pelo estado da replica (variáveis de controle etc.) e pela tabela usada como mecanismo de retenção de resultados (vide seção 5.2.4), e em seguida é computado o resumo criptográfico do estado obtido. De posse destas informações no próximo passo a réplica insere sua proposta de *checkpoint* (uma tupla CKPTPROP), que reflete o seu estado atual (linha 9). A tupla da proposta contém o número do *checkpoint* em questão, o número da última requisição processada e o resumo criptográfico do estado da réplica (calculado na linha 8). A partir daí os servidores inspecionam o espaço de tuplas até coletarem f propostas deste *checkpoint* feitas por outros servidores (linhas 11-17). Finalmente, após a coleta da proposta emitida pela maioria, resta consolidar o *checkpoint* para torná-lo estável. Assim, o último passo do algoritmo é inserir a prova do *checkpoint* no espaço. Para evitar que todas as réplicas insiram uma prova de *checkpoint* (só é necessário uma única), a operação *cas* é utilizada nesta inserção (linha 18), i.e., um servidor só insere a tupla CKPT se esta não tiver sido inserida previamente. Cabe ressaltar que se o estado da aplicação é muito grande, o que inviabiliza sua inserção no espaço por meio da tupla de *checkpoint*, a função $snapshot$ realiza uma cópia (de *backup*) do estado atual da aplicação e inclui na tupla de *checkpoint* somente o estado da réplica (variáveis de controle, tabela de retenção de resultados), além do resumo criptográfico que inclui também o estado copiado. Esta cópia do estado é necessária para permitir a recuperação pontual de uma réplica, caso haja uma requisição para recuperação do estado a partir do *checkpoint*. Convém salientar que a cópia do estado sempre é efetuada de forma incremental, isto é, no caso do primeiro *checkpoint* é realizada uma cópia integral do estado, para os *checkpoints* posteriores são realizadas cópias incrementais do estado da aplicação.

A política de acesso para o algoritmo 3 é descrita na figura 3. Nesta política descrevem-se em que situações as operações *rdp*, *cas* e *out* têm sua execução permitida. A operação *out* pode ser usada para depositar no espaço tuplas CKPTPROP que representam as propostas de *checkpoint*, desde que um servidor s_i não tenha já depositado uma proposta de *checkpoint* com

esse número. Invocações à operação *rdp* são permitidas desde que usadas para coletar as tuplas CKPTPROP. Em se tratando de invocações à operação *inp* (para o processo de coleta de lixo descrito na seção 5.2.7), só é permitida a exclusão de tuplas que contém requisições (tuplas **REQUEST**), desde que elas já tenham sido processadas e/ou executadas pelos servidores e estejam contidas em algum *checkpoint* válido, do contrário estas tuplas permanecem no espaço até que a condição seja satisfeita.

Por fim, a consolidação do *checkpoint* (tupla CKPT) é feita somente através da operação *cas*, sendo que para tal deve haver no mínimo $f + 1$ propostas no espaço de tuplas com o mesmo número, número de requisição atual e resumo. Além disso, a tupla que corresponde ao *checkpoint* anterior deve estar contida no espaço, caso este *checkpoint* não seja o primeiro. Note que da forma como a regra para a operação *cas* funciona, é impossível que as réplicas maliciosas (no máximo f) insiram *checkpoints* forjados no espaço (elas não podem fazer $f + 1$ propostas com o resumo desse *checkpoint*).

Object State TS

R_{out}: $execute(out(\langle CKPTPROP, p, ckptnum, x, y\rangle)) \ :-$
$\quad invoke(p, out(\langle CKPTPROP, p, ckptnum, x, y\rangle)) \wedge (\nexists i, j : \langle CKPTPROP, p, ckptnum, i, j\rangle) \in TS)$
R_{rdp}: $execute(rdp(\langle CKPTPROP, p, ckptnum, x, y\rangle)) \ :-$
$\quad invoke(p, rdp(\langle CKPTPROP, p, ckptnum, x, y\rangle))$
R_{rdp}: $execute(rdp(\langle CKPT, ckptnum, x, y, z\rangle)) \ :-$
$\quad invoke(p, rdp(\langle CKPT, ckptnum, x, y, z\rangle))$
R_{cas}: $execute(cas(\langle CKPT, ckptnum, seqno, *, hash\rangle, \langle CKPT, ckptnum, seqno, state, hash\rangle)) \ :-$
$\quad invoke(p, cas(\langle CKPT, ckptnum, seqno, *, *\rangle, \langle CKPT, ckptnum, seqno, state, hash\rangle)) \wedge$
$\quad \exists X \subset S : (|X| \geq f + 1 \wedge \forall s \in X : \langle CKPTPROP, s, ckptnum, seqno, hash\rangle) \wedge$
$\quad (ckptnum = 1 \vee \exists x, y, z : \langle CKPT, ckptnum - 1, x, y, z\rangle) \in TS)$
R_{inp}: $execute(inp(\langle REQUEST, seqno, w, x, y, z\rangle)) \ :-$
$\quad invoke(p, inp(\langle REQUEST, seqno, w, x, y, z\rangle)) \wedge (\exists i, j, k, l : \langle CKPT, i, j, k, l\rangle \in TS) \wedge seqno \leq j)$

Figura 5.3: Política de acesso para o PEATS usado no algoritmo 3.

5.2.6 Mecanismo de *Logging* e Recuperação de Mensagens

Como todas as requisições são armazenadas no espaço de tuplas (PEATS), exploramos esta facilidade para fins de definição e especificação de um mecanismo de *logging* estável e persistente. Este mecanismo é necessário para a recuperação das réplicas que por ventura venham a falhar. As requisições enviadas perduram no espaço até o momento da gravação de um *checkpoint* posterior, que sinaliza que as requisições anteriores a ele não são mais necessárias durante o processo de recuperação de réplicas. Deste modo, para a recuperação pontual de

uma réplica o processo restaura os *checkpoints* necessários, e se houverem requisições após o último *checkpoint*, estas são recuperadas diretamente do espaço.

No entanto, conforme já descrito na seção anterior, caso o estado da aplicação seja muito grande inviabilizando seu armazenamento no PEATS, além da recuperação dos dados contidos no *checkpoint*, a recuperação do estado da aplicação é realizado por intermédio de uma rotina de recuperação auxiliar. Neste caso, a réplica em estado de recuperação requisita às demais réplicas o estado da aplicação a partir do envio de uma tupla $\langle RECOVER, p_i, p_j, D(state)\rangle$, onde p_i corresponde ao identificador da réplica que requisitou a recuperação, p_j é o identificador da réplica escolhida para enviar o estado, este processo é escolhido de forma aleatória tomando como base o número último *checkpoint* estável onde $j = ckptnum \bmod |R|$, e $D(state)$ corresponde ao resumo criptográfico do estado que se tornou estável durante o último *checkpoint*, estado o qual foi copiado por cada uma das réplicas. Caso o estado recebido pela réplica em recuperação seja confirmado como estável, a réplica passa a incorporá-lo na aplicação. Este processo de confirmação é realizado a partir da computação do resumo criptográfico do estado recém-recebido, sendo este resumo comparado com o resumo do estado contido no *checkpoint*. No entanto, caso o estado recuperado não seja confirmado (não é considerado estável), a réplica em recuperação escolhe outra réplica do grupo para lhe enviar o estado e o processo é reiniciado.

5.2.7 Mecanismo de Coleta de Lixo

Para evitar a saturação do espaço de tuplas (com muitas tuplas REQUEST, por exemplo) propomos um mecanismo para coleta de lixo. Este mecanismo é responsável por limpar as tuplas de requisições e de *checkpoints* não mais usadas pelo sistema, e com garantia de que as tuplas candidatas à exclusão não mais estão sendo utilizadas por servidores corretos.

A coleta de lixo no RePEATS é feita logo após a criação de um *checkpoint*. Todos os servidores tentam remover tuplas REQUEST, CKPTPROP e CKPT (criadas nos algoritmos 2 e 3) que tenham se tornadas obsoletas devido ao último *checkpoint* criado. Por exemplo, quando se cria um *checkpoint* de número k e com o número de última requisição executada r, potencialmente é possível apagar todas tuplas REQUEST com número de seqüencia menor que r e todas as tuplas CKPT e CKPTPROP com número menor que k. No entanto, é preciso ter cuidado para não apagar requisições que ainda não foram processadas por réplicas "atrasadas",

pois neste caso, em havendo pelo menos uma réplica maliciosa no sistema, é possível que os clientes que enviaram as requisições apagadas fiquem bloqueados indefinidamente a espera de $f + 1$ respostas (mesmo retransmitindo as requisições).

A resolução deste problema só é possível em virtude do fato de admitirmos que a tabela usada como mecanismo de retenção de resultados faz parte do estado contido no último *checkpoint* estável, desta forma todo servidor (inclusive os atrasados e recuperados, que estabeleceram seu estado a partir de um *checkpoint* no sistema) sabe o que responder a um cliente em caso de requisições repetidas. Em se tratando do cliente, conforme descrito na seção 5.2.4, cada requisição enviada possui um número de identificação que é incrementado a cada requisição enviada, desta forma, cada uma de suas requisições é única e, no caso de uma requisição colocada no espaço demorar muito a ser respondida, ela é re-enviada pelo cliente (através do algoritmo de replicação).

5.2.8 Correção do REPEATS

Nesta seção demonstramos que o REPEATS satisfaz as propriedades especificadas na seção 5.2.2. Para realizarmos esta prova admitimos as hipóteses e premissas já descritas na seção 5.2.4 para os algoritmos 2 (replicação) e 3 (*checkpointing*), nomeadamente:

1. Cada cliente inclui no comando contido em sua requisição um identificador numérico único e crescente para cada requisição colocada no espaço;

2. O estado das réplicas armazenado através do algoritmo de *checkpointing* contém o identificador, o resumo criptográfico e a resposta do último comando executado de cada cliente;

3. Clientes reenviam comandos que demoram muito a ter resposta (usa-se um temporizador no cliente);

4. Quando um servidor (muito lento) percebe que uma requisição ainda não processada por ele já foi removida do espaço (verifica-se que uma requisição i não existe no espaço, porém há um *checkpoint* com número de requisição $j > i$), este servidor atualiza seu estado usando os dados contidos no último *checkpoint* do espaço;

5. Ao lerem uma requisição de um cliente no espaço, os servidores verificam se o comando

contido nesta já foi executado em seu estado, se sim, respondem com a resposta armazenada, caso contrário a requisição é processada normalmente.

6. A cada N requisições processadas, as réplicas executam o algoritmo de *checkpointing* com vista a armazenar o estado resultante dos comandos executados. Após criar o *checkpoint*, todas as tuplas REQUEST, CKPTPROP e CKPT anteriores são removidas.

Com essas premissas é possível provar a corretude do algoritmo mesmo tendo em conta a execução de um algoritmo de coleta de lixo para uma gestão de memória mais racional no PEATS.

Antes de provar que o RePEATS satisfaz as propriedades de *safety* e *liveness* descritas na seção 5.2.2, apresentamos uma série de lemas intermediários úteis na prova destas propriedades.

Lema 1 *Em qualquer ponto da execução do RePEATS, as seguintes propriedades são invariantes do PEATS usado nos algoritmos 2 e 3:*

1. *Para qualquer* $seqno \geq 1$*, existe no máximo uma tupla* $\langle REQUEST, seqno, p, reqid, C\rangle$ *no espaço de tuplas;*

2. *Para qualquer tupla* $\langle REQUEST, seqno, p, reqid, C\rangle$ *no espaço de tuplas, com* $seqno > 1$*, existe no espaço exatamente uma tupla* $\langle REQUEST, seqno-1, p', reqid', C'\rangle$ *ou uma tupla* $\langle CKPT, ckptnum, seqno-1, S, D(S)\rangle$.

Prova (esboço): Estas duas invariantes são conseqüências diretas dos algoritmos 2 e 3 de suas política de acesso:

1. A partir da política de acesso descrita na figura 5.2, pode-se verificar que uma tupla REQUEST 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 REQUEST com o mesmo número de seqüência *seqno* 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 REQUEST com o mesmo número de seqüência no espaço de tuplas.

2. Novamente, a partir da política de acesso da figura 5.2 (considerando a premissa 6, des- crita anteriormente) é possível ver que a operação *cas* só pode ser executada para inserir uma operação (tupla REQUEST) na posição *seqno* se existe uma tupla

REQUEST no espaço com número de seqüência uma unidade menor ou uma tupla CKPT cujo terceiro campo é igual a $seqno - 1$. Isto garante que existe uma tupla $\langle \text{REQUEST}, seqno - 1, p', reqid', C' \rangle$ ou uma tupla $\langle \text{CKPT}, ckptnum, seqno - 1, S, D(S) \rangle$ no espaço quando da inserção da operação na posição *seqno*. A garantia de que não existe mais de uma tupla dessa é conseqüência direta da primeira parte do lema. ■

Lema 2 *Se uma tupla $\langle CKPT, ckptnum, seqno, S, D(S) \rangle$, que representa o checkpoint com estado S (resultante da execução de todos os comandos em requisições com número de seqüencia menor igual a seqno em ordem crescente) é armazenado no espaço, então pelo menos um processo correto participou na geração deste checkpoint.*

Prova (esboço): A regra R_{cas} da política definida na figura 5.3 define (entre outras coisas) que uma tupla como a descrita acima só pode ser inserida no espaço de tuplas através de uma operação *cas* e apenas se existirem pelo menos $f + 1$ tuplas $\langle \text{CKPTPROP}, s, ckptnum, seqno, D(S) \rangle$ inserida por diferentes réplicas s. Considerando que, por premissa, temos que a função D gera um resumo criptográfico resistente a colisão e que temos no máximo f réplicas faltosas, podemos concluir que pelo menos um servidor correto inseriu uma tupla do tipo CKPTPROP validando o estado S como o estado correto após a execução de todas as requisições com número de seqüencia menor igual a *seqno*. ■

Lema 3 *Assumindo que a função local_execute executada nos servidores é determinista, o REPEATS garante que um comando inserido no PEATS em uma tupla REQUEST terá uma mesma resposta de pelo menos $f + 1$ servidores.*

Prova (esboço): Se o comando C_i foi inserido, então ele será processado por algum servidor correto. Temos de analisar dois casos:

- **A requisição é processada por $f + 1$ réplicas corretas**: Se $f + 1$ réplicas lêem a requisição do espaço, estas executam o comando requisitado e, tendo em vista que a função *local_execute* é determinista e que todas as réplicas processaram a mesma seqüencia de comandos antes de C_i, as $f + 1$ réplicas corretas enviam a mesma resposta ao cliente.

- **A requisição não é processada por $f + 1$ réplicas corretas**: Neste caso a requisição é apagada antes de $f + 1$ réplicas corretas a processarem. Como por premissa

temos que uma requisição é apagada apenas se o estado do último *checkpoint* armazenado no espaço contém um estado onde esta requisição já foi processada, a tupla $\langle \text{CKPT}, ckptnum, seqno, S, D(S)\rangle$, com $seqno > i$ foi inserida no espaço. De acordo com o lema 2, tal *checkpoint* só pode ser gerado se pelo menos uma réplica correta participa em sua geração. Este processo correto validou o estado S como contendo o resultado de todos os comandos com número de seqüencia menor igual a *seqno*, inclusive C_i. Isto implica que o estado S computa a execução de C_i e que a sua resposta está na tabela que contém a(s) última(s) resposta(s) para cada cliente. Assim, outras réplicas corretas que não processaram esta requisição antes de sua remoção do espaço podem, portanto, ler o estado associado ao *checkpoint ckptnum* armazenado no espaço para então ter a resposta para o comando C_i. Quando o temporizador no cliente vencer, ele re-envia a requisição e os servidores corretos terminam por recebê-la, e como percebem que é uma retransmissão de C_i, enviam a resposta contida no estado S para o cliente. Em ambos os casos, o cliente termina por receber $f + 1$ respostas iguais. ■

Tendo estes lemas em consideração, podemos provar as propriedades de *safety* e *liveness* definidas na seção 5.2.2.

Teorema 1 *O RePEATS garante que todas as réplicas corretas computam todos os comandos inseridos no espaço em ordem total.*

Prova (esboço): Pelo algoritmo 2 é possível ver que os servidores corretos lêem e executam comandos do espaço em ordem crescente de número de seqüencia das tuplas REQUEST. Isto somado ao lema 1, que prova que existe apenas uma tupla REQUEST para cada número de seqüencia no espaço, demonstra que todas as requisições são computadas por todas as réplicas corretas na mesma ordem. Devido ao algoritmo de coleta de lixo, pode ser que alguma tupla REQUEST ainda não processada por uma réplica correta seja removida do espaço, no entanto o efeito do comando contido nesta tupla se faz presente no estado contido no último *checkpoint* armazenado no sistema, e, portanto, quando uma réplica obtém este estado, ela automaticamente computa os comandos que levaram o sistema do estado inicial a este estado. ■

Teorema 2 *O RePEATS é livre de bloqueio.*

Prova (esboço): Este lema é provado por contradição. Considere, sem perda de generalidade, uma execução onde apenas dois clientes corretos p_1 e p_2 executam os comandos C_1 e C_2, res-

pectivamente. Suponha que eles permanecem parados para sempre, não recebendo resposta para essas invocações (nenhum dos dois desbloqueia). Vamos mostrar que isto não pode acontecer. Uma inspeção no algoritmo mostra que os clientes permanecem avançando na lista de requisições no espaço até que eles obtenham o número de seqüencia da operação mais recente introduzida no PEATS, com campo posição igual à *seqno*. Neste ponto, p_1 e p_2 vão tentar adicionar suas invocações no espaço na posição *seqno* + 1 através da execução da operação *cas* (linha 8). Como assumimos que o PEATS é linearizável, as duas invocações à operação *cas* devem acontecer uma após a outra, assim uma tupla REQUEST com C_1 ou com C_2 será inserida no espaço na posição *seqno* + 1. O processo bem sucedido executando *cas* para esta posição vai inserir seu comando, e vai ficar esperando uma resposta dos servidores (linha 9). Pelo lema 3, temos que uma requisição inserida no espaço terá $f+1$ respostas iguais de servidores diferentes, e portanto, o cliente que inseriu o comando consolidará um resposta e terminará o algoritmo. Este fato contradiz nossa afirmação de que ambos os clientes ficam bloqueados e não terminam o algoritmo. ■

5.3 Conclusões do Capítulo

Neste capítulo introduzimos um modelo de replicação fundamentado no modelo de coordenação generativa, o qual foi concretizado por meio da proposta de uma arquitetura para replicação tolerante a faltas bizantinas denominada REPEATS. No REPEATS é apresentada a vantagem em se replicar serviços por meio de uma abstração de memória compartilhada, no que concerne aos aspectos relacionados à coordenação de réplicas. Além do mais, o REPEATS nos permitiu concretizar serviços tolerantes a faltas bizantinas com resistência ótima ($2f + 1$), desde que exista um PEATS como suporte de comunicação e coordenação para as réplicas que compõe o sistema, e para a interação dos clientes com o serviço confiável.

A arquitetura proposta permite separar os aspectos relacionados à parte funcional da aplicação, dos aspectos concernentes à coordenação de réplicas e do protocolo de replicação, seguindo o princípio de separação do acordo e execução proposto por [5]. O uso do espaço de tuplas em nossa proposta nos permite tirar vantagens em relação aos protocolos de replicação existentes tais como: construção de um mecanismo de *logging* estável, confiável e seguro para o sistema replicado, nossa camada de acordo pode ser usada para diversas outras finalidades e aplicações, não sendo uma aplicação dedicada como a camada de acordo proposta por [5].

Por fim, o RePEATS permite elevar a qualidade dos serviços replicados a partir do uso das políticas de acesso fornecidas pelo PEATS. Esta funcionalidade permite que nossa arquitetura tenha a capacidade de inibir o comportamento malicioso das réplicas faltosas, não permitindo que as ações emitidas por elas sejam refletidas no sistema, e assim provendo um sistema muito mais robusto e confiável.

Capítulo 6

Aspectos de Implementação e Resultados

Neste capítulo apresenta-se os aspectos relacionados à implementação de um protótipo do RePEATS, bem como os resultados obtidos a partir de *microbenchmarks* e *macrobenchmarks* realizados sobre o protótipo. Estes *benchmarks* são baseados em comparações do modelo e arquitetura propostas em relação a outros protocolos e arquiteturas de replicação. Além do mais, também é apresentada a implementação de um sistema de arquivos distribuído (*Network File System*) através do RePEATS e os respectivos resultados em termos de latência e *throughput*.

A idéia da implementação de um NFS se deu em virtude deste ser um sistema/serviço largamente utilizado, e pelo fato dos trabalhos relacionados ao RePEATS também o terem feito quando de suas experimentações. Esta implementação do NFS permitiu a realização de comparações mais factíveis para este trabalho.

6.1 Arquitetura e Protótipo de Implementação

Os algoritmos e mecanismos descritos nos capítulos anteriores foram implementados usando como base a implementação do **PEATS** (espaço de tuplas tolerante a faltas bizantinas) denominada DepSpace[1] [81], o qual foi apresentado detalhadamente no capítulo 4 desta dissertação. Resumidamente, o DepSpace consiste de uma biblioteca de classes escrita em Java e provê todo o suporte necessário para a implementação do RePEATS (i.e suporte ao uso de políticas de granularidade fina para controle de acesso ao espaço).

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

Em virtude do DepSpace estar disponível somente na linguagem em Java, todos os componentes do RePEATS foram implementados nesta linguagem, sendo que as ferramentas criptográficas usadas para a implementação do protótipo foram as presentes na API *Java Cryptography Extensions* da JVM 1.6.0_7 da SUN. O RePEATS consiste basicamente em uma biblioteca com algumas classes, que implementam uma camada de abstração (*stub*) entre as entidades do serviço (clientes e servidores) e o espaço de tuplas. Mais especificamente, os processos clientes e servidores acessam estas classes que implementam os algoritmos descritos no capítulo 5, que por sua vez fornecem a conectividade com o DepSpace. É válido salientar que este acesso (em nível de substrato de rede) é feito através de canais confiáveis e autenticados, suportados por *sockets* TCP e MACs. Este mesmo tipo de canal é usado também para envio de resposta dos servidores aos clientes (comunicação direta).

Tendo em vista o desempenho do sistema no que concerne à latência das operações e escalabilidade do sistema como um todo, algumas das novas funcionalidades providas pela API NIO [89] foram usadas na implementação do RePEATS, dentre as quais destacamos principalmente os *socketchannels* e os *sockets* não-bloqueantes. Também foram usadas as classes presente na nova API *java.util.concurrent* para a implementação de semáforos e monitores, visando a concepção de um controle de concorrência mais eficiente e otimizado.

A figura 6.1 apresenta a arquitetura definida para o protótipo de concretização do RePEATS. A partir desta figura podemos verificar que entre as entidades do sistema (ao lado esquerdo o cliente e ao lado direito os servidores) existe um sistema subjacente de comunicação que é representado pelo espaço de tuplas (PEATS). Nosso modelo de replicação foi constituído no intuito de ser menos intrusivo possível, isto é, tendo em vista evitar ao máximo a realização de alterações nas especificações e chamadas das aplicações clientes e dos serviços. Neste sentido, foram adicionados componentes de conformidade para prover a compatibilidade entre as chamadas padrões das aplicações cliente e servidor, onde no lado cliente tal componente é implementado como um *wrapper* que transforma as chamadas dos clientes em tuplas de requisição ao serviço, incluindo para tanto a operação invocada pelo cliente e os seus argumentos em tuplas. Em seguida, a tupla gerada é enviada aos servidores por meio do *proxy*, que por sua vez fornece acesso ao PEATS. No lado servidor a requisição é obtida também por meio do *proxy*, e, uma vez recebida uma requisição, o código que implementa a réplica

verifica a consistência desta requisição e encaminha ao serviço por meio de um despachante (*dispatcher*) que por sua vez implementa as chamadas padrões do serviço em questão. Note que o uso do despachante evita a necessidade de alteração do código do serviço final (aplicação).

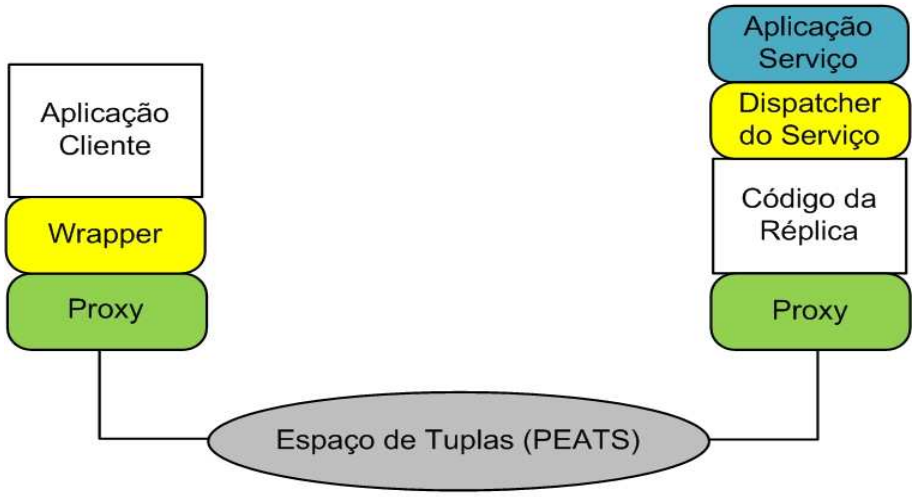


Figura 6.1: Arquitetura do Protótipo

6.1.1 Ambiente de Execução

O ambiente de execução para avaliação de desempenho foi constituído por máquinas Dell Optiplex 755, todas com a mesma configuração de *hardware* e *software*. No que concerne ao *hardware*, cada uma das máquinas foi composta por 1 processador Intel®Core™2 Duo 2.33GHz, 2GB RAM e uma interface Ethernet Intel 82566DM-2 Gigabit. A conectividade entre as máquinas foi realizada por meio de um *switch* 3Com SuperStack3 100/1000Mb/s. A composição do ambiente de *software* de todas as máquinas foi constituída pelo sistema operacional SUSE Linux SLES 10 (*Kernel* 2.6.16.21-0.8-smp x86-64) e pela máquina virtual Java IBM 1.6.0 com a opção JIT (*Just In Time*) ativada. Foi optado por utilizar a JVM JIT da IBM por ela apresentar melhor desempenho em relação a JVM Sun.

Em relação à quantidade de máquinas necessárias para os experimentos, a configuração variou de acordo com o parâmetro f, isto é, de acordo com número de faltas permitidas para

cada experimento. Contudo, a configuração mínima exigida para a execução do protótipo consiste em $n_s = 2f_s + 1$ réplicas do serviço e $n_{ts} = 3f_{ts} + 1$ réplicas do espaço de tuplas (um requisito do modelo de replicação adotado pelo DepSpace [81]), tolerando assim f_s réplicas faltosas do serviço e f_{ts} réplicas faltosas do espaço de tuplas. No entanto, há de se considerar que o serviço fornecido pelas mesmas n_{ts} máquinas pode ser utilizado concorrentemente por diferentes conjuntos de serviços replicados (diferentes aplicações), o que reduz consideravelmente o custo em se replicar serviços através do RePEATS. As mesmas $n_{ts} = 3f_{ts} + 1$ máquinas foram usadas (em momentos distintos) para a execução dos demais protocolos avaliados.

6.1.2 Avaliação de Desempenho

A avaliação de desempenho de sistemas de computação distribuída pode ser realizada de forma analítica (teórica) ou empírica (pragmática). A abordagem de avaliação analítica possui fundamentos mais teóricos e tem o objetivo de medir a eficiência de um sistema, através do estudo das complexidades algoritmicas. Por outro lado, a abordagem empírica é baseada em amostras de dados, que são extraídos através das execuções dos algoritmos do sistema. Para esta avaliação, como os algoritmos de coordenação de réplicas do RePEATS são implementado por meio de memória compartilhada, a despeito dos demais trabalhos avaliados que o fazem por meio de passagem de mensagens (coordenação direta), a concepção de uma avaliação analítica não teria sentido algum, já que na literatura as métricas empregadas para a avaliação da complexidade de algoritmos distribuídos em memória compartilhada são diferentes das métricas usadas para medir a complexidade de algoritmos distribuídos baseados em trocas de mensagens [24]. Por esta razão, nossa avaliação foi realizada sob fundamentos pragmáticos e concretizadas por meio de experimentações em ambiente real (avaliação empírica).

A metodologia de avaliação empregada contou com a definição de métricas, de modo a permitir a coleta de dados representativos quanto ao comportamento dos protocolos de replicação, em diversas condições de carga. Neste caso, optamos por computar as medidas de desempenho concernentes a latência e *throughput* dos protocolos, as quais foram obtidas a partir da realização de *microbenchmarks* e *macrobenchmarks*. A medidas de latência foram obtidas a partir da transmissão de um conjunto de mensagens, e da espera das respostas dos receptores. O tempo deste processamento é referenciado como *round-trip*. Já as medidas de *throughput* significam a capacidade máxima de processamento de requisições por unidade de

tempo. Para a obtenção dos valores destas medidas nos baseamos no tempo de processamento das requisições desde o envio, até a recepção de todas as respostas advindas das réplicas.

A fim de avaliar o desempenho do REPEATS realizamos a implementação dos protocolos PBFT [65], BFT com separação de acordo e execução [5] e Zyzzyva [6]. A avaliação foi realizada em termos de latência (tempo médio para execução de uma operação) e *throughput* (quantidade média de operações por segundo). Para computar as medidas de desempenho, tomamos como base os algoritmos 4 (latência) e 5 (*throughput*), descritos a seguir. Cabe ressaltar que a computação destes experimentos foi baseada no tempo médio de *round-trip* para uma requisição, isto é, tomando como base o tempo total de processamento de uma operação desde o envio da mensagem pelo cliente até a chegadas de $f + 1$ confirmações de diferentes servidores, onde cada confirmação indica que a mensagem foi entregue por aquele servidor, após ter sido ordenada.

Algoritmo 4 Algoritmo de computação da latência

Variáveis locais:

1: $inicio = \perp$ {marcador do tempo de início}
2: $final = \perp$ {marcador do tempo final}

tarefa de envio/recepção

3: $inicio \leftarrow clock()$
4: **while** $(i \leq execucoes)$ **do**
5: $send(requisicao)$
6: **for** $(j \leftarrow 0; j < f + 1; j++)$ **do**
7: $receive(resposta)$
8: **end for**
9: $i \leftarrow i + 1$
10: **end while**
11: $final \leftarrow clock()$
12: $latencia \leftarrow (final - inicio)/execucoes$

É valido salientar que o REPEATS é totalmente diferente dos demais trabalhos avaliados, no sentido de que em nossa proposta todos as comunicações são realizadas por meio de memória compartilhada, já os outros trabalhos realizam a mesma tarefa a partir do método convencional de passagem de mensagens. Isto de certa forma tem influência no desempenho do sistema, já que uma abstração de memória compartilhada em nível de *middleware*, é implementada a partir de passagens de mensagens através de *sockets* TCP/IP.

Em primeira instância, a avaliação do desempenho foi realizada por meio de *micro-benchmarks* no intuito de analisar a eficiência dos algoritmos de ordenação, sem a influência de

Algoritmo 5 Algoritmo de computação do *throughput*

Variáveis compartilhadas:

1: $inicio = \perp$ {marcador do tempo de início}
2: $final = \perp$ {marcador do tempo final}
3: $execs = 0$ {número de execuções}

tarefa de envio

4: $inicio \leftarrow clock()$
5: $i \leftarrow 1$
6: **while** $(i \leq execs)$ **do**
7: $\quad send(requisicao_i)$
8: $\quad i \leftarrow i + 1$
9: **end while**

tarefa de recepção

10: $i \leftarrow 1$
11: **while** $(i \leq execs)$ **do**
12: $\quad$ **for** $(j \leftarrow 0; j < f + 1; j++)$ **do**
13: $\quad\quad receive(resposta_i)$
14: $\quad$ **end for**
15: $\quad i \leftarrow i + 1$
16: **end while**
17: $final \leftarrow clock()$
18: $throughput \leftarrow execs/(final - inicio)$

uma aplicação. Um aspecto de grande relevância para a avaliação de desempenho é tomar conhecimento do comportamento do protocolo para diferentes configurações e níveis de atividades, tais como a escalabilidade em relação ao número de réplicas e tamanho das mensagens. Deste modo, as execuções dos protocolos avaliados foram realizadas fazendo uso de um serviço sem estado e com operações **nulas** (que não fazem nada), variando os tamanhos das requisições e dos resultados em 0 e 4Kbs.

6.1.2.1 Execuções Normais

Os primeiros experimentos foram realizados com o sistema em execuções normais, isto é, sem a ocorrência de faltas. Foram efetuadas 10000 execuções da operação, realizadas por 30 clientes concorrentes. Para a tabulação dos resultados foi desprezado 1% dos valores com maior desvio, conforme percebido pelos clientes. A figura 6.2 apresenta os resultados preliminares do *throughput* do sistema em relação ao tamanho das requisições e respostas. Este experimentos foram realizados com 3 servidores de aplicação ($2f + 1$, sendo $f = 1$), e 4 réplicas do DepSpace ($3f + 1$, sendo $f = 1$). Ao avaliar os resultados observa-se que o RePEATS apresenta o menor desempenho em relação aos demais protocolos, no entanto, sendo muito próximo do resultado reportado para a arquitetura de "separação do acordo e execução" [5]. É valido salientar que este resultado já era esperado, pois o RePEATS é baseado

na abordagem de organização do esquema de replicação em camadas, isto é, camadas de acordo e de execução [5], e, portanto, requer mais passos de comunicação que os protocolos PBFT e Zyzzyva. Note que a comparação mais plausível para nossa proposta tem de ser feita em relação à arquitetura de "separação de acordo e execução" de [5], pois ambas são baseadas no mesmo principio de funcionamento.

Novamente ressaltamos que mesmo sendo baseado no mesmo princípio de funcionamento, o RePEATS é sensivelmente diferente da arquitetura de [5] no sentido de que em nosso modelo todas as comunicações são realizadas por meio de memória compartilhada e os demais trabalhos o fazem a partir de passagem de mensagens. Nos resultados preliminares pode-se verificar que o RePEATS tem desempenho muito próximo (mas pouco inferior) ao da arquitetura de [5]. Por outro lado se analisarmos o *throughput* em relação ao tamanho das requisições e respostas, pode-se verificar que o sistema RePEATS apresenta escalabilidade aceitável, tendo em vista que o decréscimo do número de operações/segundo é pequeno quando se aumenta o tamanho das requisições e respostas.

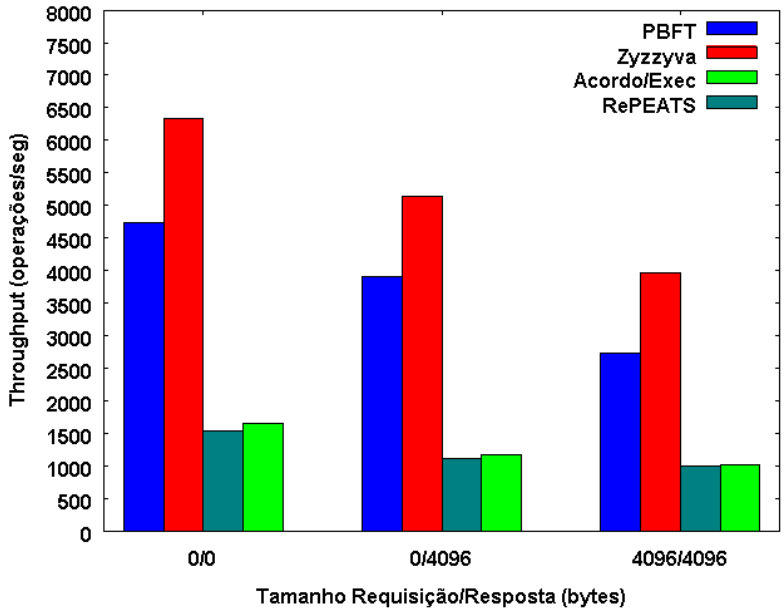


Figura 6.2: *Throughput* das operações para os sistemas avaliados

Tomando em conta as medidas de latência, os resultados dos experimentos são reportados na figura 6.3. Da mesma forma que na avaliação do *throughput*, para estes experimentos também foram executadas 10000 operações para cada protocolo, realizadas por 30 clientes

concorrentes. Os valores reportados compreendem o valor médio observado para as execuções, desprezando-se 1% dos valores com maior desvio. Os resultados (figura 6.3) demonstram que a latência das operações executadas pelo RePEATS em alguns casos é equivalente ou menor que as operações executadas pela arquitetura de separação de [5]. O fato para que os resultados sejam muito próximos é que o DepSpace [80] (implementação do espaço de tuplas usado nos experimentos) é baseado no algoritmo de replicação Paxos Bizantino, que por sua vez é equivalente ao PBFT. A explicação para o RePEATS apresentar latência pouco menor que o trabalho de [5] reside basicamente no custo de acesso ao PEATS. Por fim, da mesma forma que ocorre com o *throughput*, estes resultados mostram que o sistema tem escalabilidade razoável quando se aumenta o tamanho das requisições e respostas, pois o acréscimo na latência é menos de duas vezes. O caso particular das requisições e respostas com 4096 bytes é considerável, pois o modelo de replicação adotado requer um passo de comunicação a mais, bem como o envio e transmissão da requisição ordenada aos servidores de execução. O que por sua vez onera parcialmente o sistema, quando o tamanho das mensagens é muito grande.

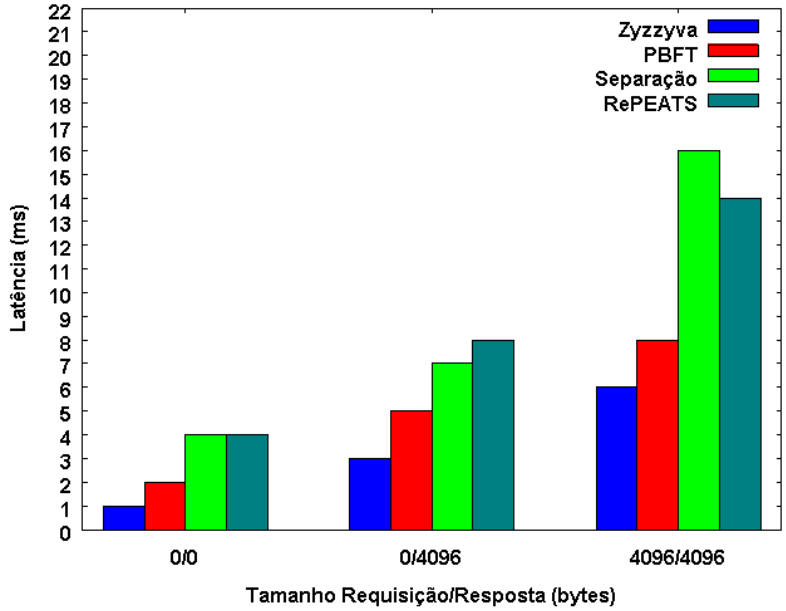


Figura 6.3: Latência das operações para os sistemas avaliados

6.1.2.2 Execuções Com Faltas/Falhas

Para avaliar a eficiência do RePEATS em casos onde há ocorrências de faltas, foram realizados os mesmos experimentos da seção anterior considerando a influência de faltas no ambiente. Para a realização destes experimentos foram introduzidas faltas nos algoritmos PBFT, Zyzzyva, na arquitetura de separação e também no RePEATS. Cabe ressaltar que para todos os protocolos as simulações contemplaram faltas nas réplicas que exerceram a função de líder (por este ser o pior caso). No caso do RePEATS e da arquitetura de [5], foram consideradas faltas na camada de acordo, já que as faltas da camada de execução têm influencia muito pequena no desempenho do sistema, já que as mesmas são mascaradas pelo sistema. Este feito se deve ao fato de que se até f réplica de um total de $2f + 1$ falharem, o cliente ainda receberá um número de respostas suficientes para consolidar a operação ($n - f \geq f + 1$).

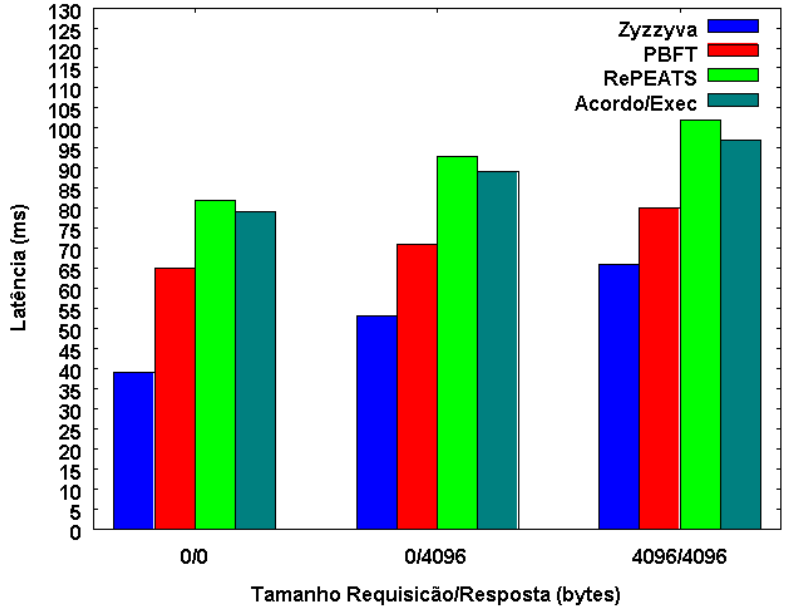


Figura 6.4: Latência das operações com falhas nos sistemas ($f = 1$)

No primeiro momento das execuções com faltas foi avaliada a latência da execução de operações nulas nos protocolos avaliados. Como já era esperado, tanto o RePEATS como a arquitetura de [5] apresentaram maior latência em relação aos demais protocolos, em virtude da camada de acordo dos referidos protocolos ser baseada no algoritmo PBFT (ou Paxos Bizantino), e deste modo, a latência apresentada contempla o tempo de execução do PBFT mais o tempo necessário para o envio das mensagens ordenadas para os servidores de execução.

Como pode ser observado na figura 6.4, apesar do desempenho de nossa solução não ser o melhor, neste quesito o REPEATS é levemente inferior a arquitetura de [5]. A razão para este fato é existe uma sobrecarga adicional em se acessar o espaço de tuplas, que por sua vez é implementado pelo PAXOS Bizantino em nível subjacente de comunicação, já que o mesmo é replicado.

O mesmo ocorre com a avaliação do *throughput* que é apresentada na figura 6.5, onde o REPEATS ainda apresenta desempenho pouco inferior ao da arquitetura de [5]. Novamente, a razão para isso é a sobrecarga que incorre nos acessos ao espaço de tuplas.

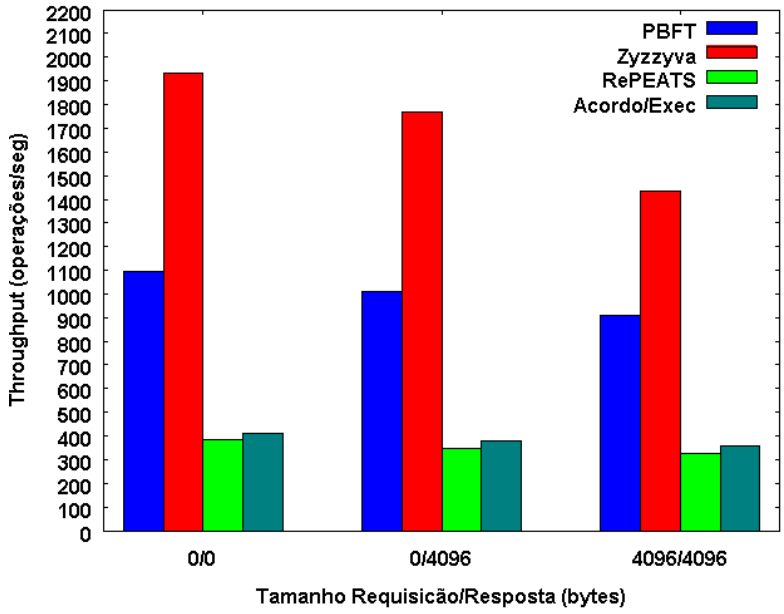


Figura 6.5: *Throughput* das operações com falhas nos sistemas ($f = 1$)

6.2 Estudo de Caso: Implementação de um Sistema de Arquivos NFS Tolerante a Faltas Bizantinas

A fim de avaliar o desempenho REPEATS em uma aplicação real, implementamos um serviço NFS [90] replicado, nos mesmos moldes de trabalhos anteriores [3, 5, 6]. Nossa implementação foi inspirada nas especificações do WebNFS [91, 92], que na verdade não é um substituto do NFS padrão, mais sim uma extensão da especificação do protocolo NFS,

que permite o uso mais eficiente dos sistemas de arquivos pelas aplicações. Para acessar aos arquivos remotos por meio do NFS é necessário realizar uma seqüencia de passos padrão, onde primeiramente se deve montar o diretório remoto a partir do cliente no qual se deseja usá-lo, e a partir dai é realizado o acesso aos arquivos. No caso do WebNFS é possível obter o acesso aos objetos remotos (arquivos, diretórios, *links*) por parte das aplicações, sem a necessidade de interação com o sistema operacional, i.e., não é necessário realizar a montagem de sistemas de arquivos remotos no VFS (*Virtual File System*) do sistema operacional do cliente. Todo o acesso a arquivos, diretórios e ligações é feito através de uma API (biblioteca de classes Java/C++) carregada pela aplicação cliente.

Em nossos experimentos foi usado um servidor NFS escrito em Java e conhecido como "jnfs"[2], que consiste em uma implementação aberta do protocolo NFS versão 2. Para prover o funcionamento das funcionalidades requeridas para o WebNFS foi necessária uma pequena modificação no núcleo do "jnfs", no entanto permitindo a compatibilidade/portabilidade com o NFS padrão. É sabido que a aplicação NFS pode ser executada tanto em nível de *kernel* como em nível de usuário (*daemon* de aplicação comum). Como nossos experimentos foram realizados em um servidor NFS escrito em Java, o NFS foi executado como aplicação em nível de usuário. Para o funcionamento replicado junto ao REPEATS foi necessária a implementação de uma entidade para realizar o despacho das requisições que chegam às réplicas (em tuplas) ao servidor NFS (componente *Dispatcher*, conforme figura 6.1). A figura 6.6 apresenta uma classe abstrata e o método que executa as chamadas ao servidor NFS para cada requisição recebida.

6.2.1 Avaliação de Desempenho

Esta seção apresenta alguns resultados experimentais comparando o NFS tolerante a faltas bizantinas construído com base no REPEATS com serviços NFS replicados pelos protocolos PBFT, Zyzzyva, arquitetura de [5] e por um serviço NFS não replicado. A métrica utilizada para medir o desempenho foi o tempo total de execução de várias operações comuns sobre arquivos. O ambiente de experimentação e testes é o mesmo descrito na seção 6.1.1. Nos experimentos foram analisadas as seguintes operações sobre o NFS: (1) criação de arquivos e

[2]Disponível em `http://www.void.org/~steven/jnfs/`.

```
public class NFSDispatcher implements Dispatcher {   public byte[] dispatch (Request args) {
    try {  int execOp= args.getOper();
           byte[] response= null;
           switch (execOp) {
              case NFSPROC_NULL:
                   /* do nothting */ response= null; break;
              case NFSPROC_GETATTR:
                   response= nfsProcGetAttr(args.getSource()); break;
              case NFSPROC_SETATTR:
                   response= nfsProcSetAttr(args.getSource()); break;
              case NFSPROC_ROOT:
                   /* do nothting */ response= null; break;
              case NFSPROC_LOOKUP:
                   response= nfsProcLookup(args.getSource()); break;
              case NFSPROC_READLINK:
                   response= nfsProcReadLink(args.getSource()); break;
              case NFSPROC_READ:
                   response= nfsProcRead(args.getSource()); break;
              case NFSPROC_WRITECACHE:
                   /* do nothting */ response= null; break;
              case NFSPROC_WRITE:
                   response= nfsProcWrite(args.getSource(), args.getData()); break;
              case NFSPROC_CREATE:
                   response= nfsProcCreate(args.getSource()); break;
              case NFSPROC_REMOVE:
                   response= nfsProcRemove(args.getSource()); break;
              case NFSPROC_RENAME:
                   response= nfsProcRename(args.getSource(), args.getDest()); break;
              case NFSPROC_LINK:
                   response= nfsProcLink(args.getSource(), args.getDest()); break;
              case NFSPROC_SYMLINK:
                   response= nfsProcSymLink(args.getSource(), args.getDest()); break;
              case NFSPROC_MKDIR:
                   response= nfsProcMkdir(args.getSource(), args.getDest()); break;
              case NFSPROC_RMDIR:
                   response= nfsProcRmdir(args.getSource(), args.getDest()); break;
              case NFSPROC_READDIR:
                   response= nfsProcReadDir(args.getSource()); break;
              case NFSPROC_STATFS:
                   response= nfsProcStatFS(args.getSource()); break;
           } return response;
    } catch (Exception e) {e.printStackTrace();}
    return null;   }
}
```

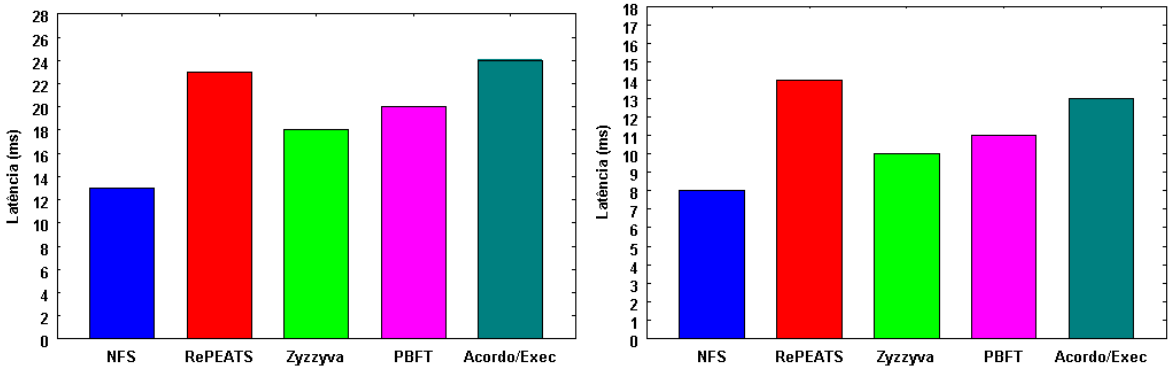
Figura 6.6: Classe e Método de Acesso ao NFS

diretórios (figura 6.7(a)), (2) exclusão de arquivos e diretórios (figura 6.7(b)), (3) listagem do conteúdo de diretórios com 100 objetos recursivos (figura 6.7(c)), (4) escrita de 2k, 4k, 8k, 16k e 32k em arquivos remotos (figura 6.7(d)), e (5) leitura dos dados de arquivos com 2k, 4k, 8k, 16k e 32k (figura 6.7(e)). Para cada um destes experimentos foram realizadas 10000 execuções da operação em questão, onde a latência reportada compreende o tempo médio necessário para a execução da operação, excluindo-se 1% dos valores com maior desvio.

A partir dos gráficos das figuras 6.7(a) e 6.7(b) podemos observar a viabilidade de uso do RePEATS, onde se verifica que o custo adicional do uso de nossa proposta incorre em 30 a 60% em relação ao acesso a um serviço NFS não replicado (i.e não tolerante a faltas). Este custo adicional se deve basicamente a latência extra em se acessar o PEATS para a execução do protocolo de ordenação de requisições. Este acesso incorre em torno de 3 ms em operações de escrita e 1 ms em operações de leitura. Este custo é moderado se considerarmos os benefícios em termos de confiabilidade e segurança que nossa solução replicada oferece. Também é digno de nota o fato da latência apresentar um crescimento leve, quando o tamanho da resposta do serviço é aumentado. Isto ocorre pelo fato das respostas serem enviadas diretamente aos clientes e não passarem pelo PEATS.

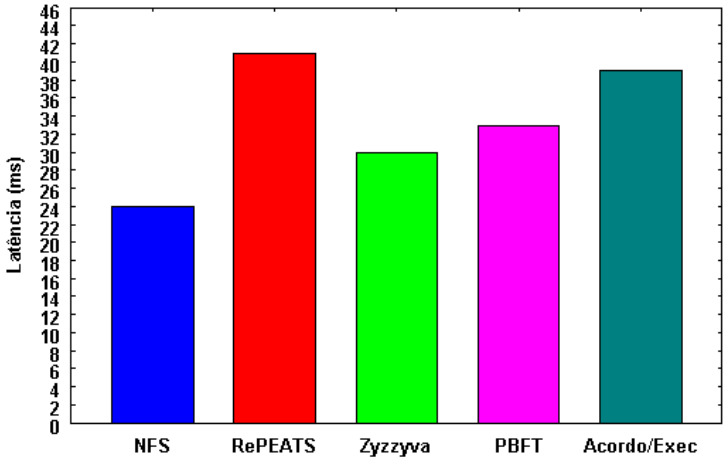
Uma última avaliação realizada sobre nossa proposta foi implementação de diversos serviços partilhando o mesmo suporte de acordo, neste caso, o mesmo PEATS. Neste caso podemos observar que o desempenho do RePEATS cai a medida que se implementa mais serviços em uma única camada de acordo. Isto ocorre pelo fato de que o PEATS é baseado no algoritmo PAXOS Bizantino, que por sua vez apresenta um baixo desempenho em situações onde há grande concorrência no sistema, como é o caso da execução dos N serviços simultâneos.

Algumas outras vantagens pertinentes do RePEATS em relação aos demais protocolos avaliados são apresentadas na tabela 6.1. O primeiro caso é em relação quantidade de réplicas de aplicação, isto é, que mantém o estado do serviço. Neste caso podemos observar que tanto o RePEATS como a arquitetura de separação de [5] compartilham das mesmas características, já que ambas são concretizadas a partir do mesmo princípio. O PBFT e o Zyzzyva requerem $3f + 1$ réplicas de aplicação, visto que o mesmo conjunto de réplicas é usado para implementar o protocolo de acordo bizantino e a aplicação. Obviamente que se verificarmos, o RePEATS e a arquitetura de [5], no total usam mais réplicas devido ao fato da camada de acordo requerer

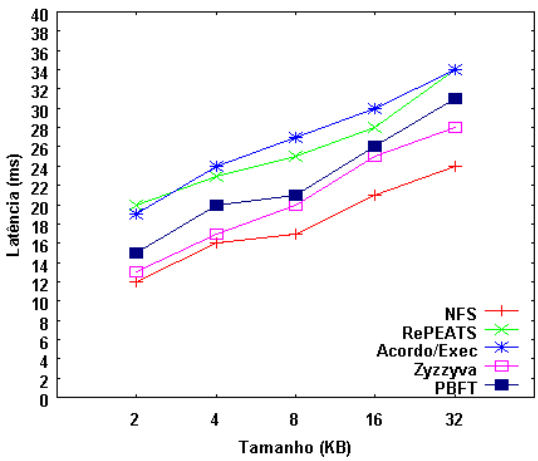


(a) Criação de arquivos.

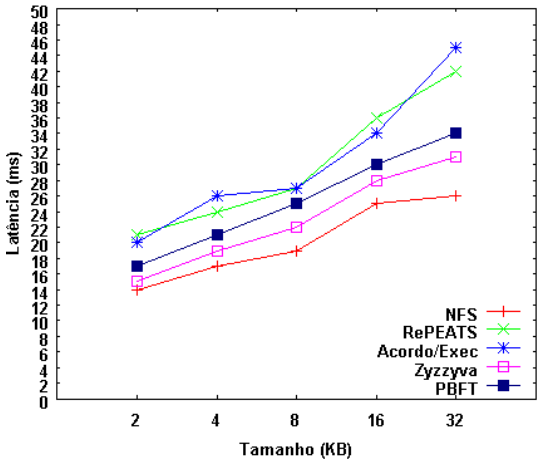
(b) Exclusão de arquivos.



(c) Listagem de diretórios.

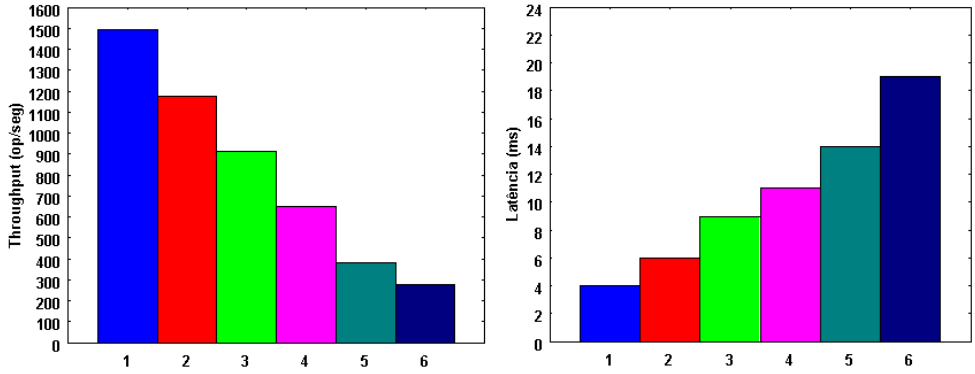


(d) Leitura em arquivos.



(e) Escrita em arquivos.

Figura 6.7: Latência de operações sobre objetos nos serviços NFS comparados: NFS nativo (API *java.io* + cliente NFS Linux), REPEATS, Arquitetura de Separação, Zyzzyva e PBFT. O servidor NFS usado em todos os experimentos é o jnfs.



(a) *Throughput*. (b) Latência.

Figura 6.8: Desempenho do RePEATS com diversos serviços em um único Espaço de Tuplas.

$3f + 1$ réplicas, totalizando $(3f + 1)_{\mathcal{A}} + (2f_E + 1)_{\mathcal{E}}$ para um serviço. No entanto, este número de réplicas cai na medida em que se implementa mais serviços sobre o mesmo suporte de acordo, por exemplo, para 6 serviços tolerando uma única réplica faltosa o PBFT e o Zyzzyva requerem 24 réplicas, já o RePEATS e a arquitetura de [5] requerem 22 réplicas para os mesmos 6 serviços. Um outro ponto no qual o RePEATS se destaca em relação aos demais, é que para N serviços implementados a mesma instância do PEATS é compartilhada entre eles, já a arquitetura de separação de [5] também compartilha as réplicas de acordo para os N serviços, no entanto, tendo que usar uma instância do protocolo de acordo para cada um dos serviços, em virtude das particularidades de cada serviço. Esta característica faz com que as réplicas de acordo tenham de ser mais robustas, caso se necessite executar N serviços sobre a mesma camada de acordo, enquanto que no RePEATS a recíproca não é afirmativa. Por fim, o RePEATS não requer mudança de visão, pois o protocolo de acordo não é baseado em líder. Cabe ressaltar que o algoritmo de mudança de visão (de todos os protocolos) requer que o serviço fique inativo até que o protocolo seja re-estabelecido e possa continuar a receber requisições.

Tabela 6.1: Comparação entre Algoritmos de Replicação BFT

	RePEATS	**Acordo/Exec**	**PBFT**	**Zyzzyva**
Réplicas do Estado do Serviço	$2f + 1$	$2f + 1$	$3f + 1$	$3f + 1$
Qtd Serviços por Camada Acordo	N	$N*$	1	1
Requer Mudança de Visão	N	S	S	S

6.3 Conclusões do Capítulo

Neste capítulo apresentamos os principais resultados referentes à implementação da arquitetura de replicação proposta. Decidimos por realizar um plano de testes em nível de *microbenchmarks* para mensurar de forma mais precisa o custo da arquitetura sem a influência de uma aplicação. Tudo isso para que pudéssemos compreender melhor o impacto da replicação de um serviço qualquer por meio do RePEATS. Estes *microbenchmarks* mostraram que o desempenho do RePEATS não é tão favorável, se comparado com os protocolos já existentes no âmbito de replicação com faltas bizantinas. No entanto, esta perda de desempenho é compensada se considerarmos o elevado nível de qualidade de serviço oferecido por nossa arquitetura. Além do mais, como as comunicações do RePEATS se dão através de memória compartilhada, isto influencia o desempenho do sistema, pois em nível de *middleware* o gerenciamento da memória compartilhada é realizado por meio de passagens de mensagens em *sockets* TCP/IP.

O *macrobenchmark* realizado a partir da implementação do sistema de arquivos distribuído (NFS), o qual foi usado como exemplo de aplicação para os experimentos, permitiu a construção de um ambiente mais realista para a avaliação de nossa proposta, além de nos auxiliar a mensurar de forma mais precisa os benefícios e custos associados ao nosso esquema de replicação. Estes testes mostraram que o custo de acesso ao serviço replicado a partir do RePEATS incorre em sobrecarga adicional de 30 a 60% em relação a um serviço não replicado, o que é justificado pelos aspectos relacionados a coordenação das réplicas do serviço.

Por fim, é válido salientar que todas as ferramentas e tecnologias empregadas na construção do protótipo são baseadas em padrões abertos, o que favorece em muito no quesito interoperabilidade e independência de plataforma para o uso do sistema desenvolvido.

Capítulo 7

Conclusões e Perspectivas Futuras

O trabalho de investigação proposto nesta dissertação atingiu os objetivos estabelecidos ao inicio do projeto. O estudo dos principais algoritmos para tolerância a falta bizantinas, bem como dos requisitos e premissas necessárias para implementá-los, nos permitiu especificar com maior precisão o modelo de sistema que amparou nossa proposta. A idéia, bem como a opção pela admissão de uma abstração de espaço de tuplas para a concretização do trabalho se deu por meio de diversas motivações, dentre as quais podemos destacar a flexibilidade de seu modelo de comunicação, a adequação do mesmo ao contexto de faltas bizantinas através do PEATS, a modularidade e simplicidade dos algoritmos especificados para este modelo de coordenação.

O estudo apresentado sobre a viabilidade de implementação de sistemas replicados sujeitos a faltas bizantinas, tendo como sistemas subjacente de comunicação e coordenação um espaço de tuplas confiável e seguro para as tarefas que concernem à coordenação e manutenção das réplicas, se confirma pelos resultados obtidos no protótipo de implementação. Vimos que, apesar dos custos de execução ser em algumas situações, maiores que os dos trabalhos como PBFT e Zyzzyva, a vantagem do modelo se dá quando consideramos a implementação de mais de um serviço sobre o mesmo suporte de coordenação e acordo. Os resultados obtidos nos permitiram formar uma opinião mais concreta sobre o uso de espaços de tuplas no contexto de sistemas distribuídos, além de que os resultados nos proporcionaram uma experiência mais precisa sobre o poder de um espaço de tuplas para a coordenação de serviços bizantinos por meio de replicação.

O modelo de interação entre as réplicas e clientes proposto neste trabalho, além do

princípio de funcionamento empregado no modelo de replicação (principio de separação de acordo e execução [5]), introduzem ganhos significativos ao sistema, já que fica a cargo das réplicas implementar apenas a aplicação e não o protocolo de acordo e aplicação, como acontece com o PBFT e Zyzzyva, e, no caso do modelo de interação, um serviço muito mais genérico do que simples primitivas de comunicação pode ser usados por diversas aplicações e serviços independentes, sem terem relações uns com os outros.

Por fim, o protótipo implementado e os testes realizados demonstraram que, apesar de moderado, o custo de execução do RePEATS não é muito favorável quando comparado aos sistemas de replicação tradicionais como *Zyzzyva* e PBFT, o que era esperado já que o RePEATS é inspirado no modelo de "separação de acordo e execução" e por isso requer um passo a mais de comunicação. No entanto, a eficiência do RePEATS é compensada se comparado com o trabalho que cunhou a arquitetura de separação de acordo e execução [5]. Ainda assim, é passível de comentário que, quando avaliado em um ambiente real o custo de acesso ao serviço incorre em sobrecarga adicional de 30 a 60% maior em termos de latência que o custo de acesso ao NFS não replicado, o que é bastante satisfatório dado o elevado nível de qualidade de serviço oferecido por nossa arquitetura, o que demonstra e atesta a viabilidade e aplicabilidade do esquema/arquitetura de replicação propostos em ambientes reais.

Com a realização deste trabalho, algumas perspectivas para novos trabalhos que poderão vir a ser desenvolvidos:

- Realização da implementação de outras técnicas de replicação, já que nos concentramos apenas em replicação ativa (máquina de estados);
- Implementação da arquitetura proposta em outras aplicações. Como sugestão, seria particularmente interessante a implementação em um serviço de banco de dados (possivelmente heterogêneo);
- Realização de um plano de testes mais completo em um ambiente com a influência de faltas.

Referências Bibliográficas

[1] LAMPORT, L., "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, v. 21, n. 7, p. 558–565, July 1978.

[2] SCHNEIDER, F. B., "Implementing Fault-Tolerant Service Using the State Machine Approach: A Tutorial", *ACM Computing Surveys*, v. 22, n. 4, p. 299–319, Dec. 1990.

[3] CASTRO, M., LISKOV, B., "Practical Byzantine Fault-Tolerance and Proactive Recovery", *ACM Transactions Computer Systems*, v. 20, n. 4, p. 398–461, Nov. 2002.

[4] CASTRO, M., RODRIGUES, R., LISKOV, B., "BASE: Using Abstraction to Improve Fault Tolerance", *ACM Transactions Computer Systems*, v. 21, n. 3, p. 236–269, Aug. 2003.

[5] YIN, J., MARTIN, J.-P., VENKATARAMANI, A., et al., "Separating Agreement from Execution for Byzantine Fault Tolerant Services". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles - SOSP'03*, p. 253–267, Oct. 2003.

[6] KOTLA, R., ALVISI, L., DAHLIN, M., et al., "Zyzzyva: speculative byzantine fault tolerance". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles - SOSP'07*, p. 45–58, ACM: New York, NY, USA, 2007.

[7] CHUN, B.-G., MANIATIS, P., SHENKER, S., et al., "Attested append-only memory: making adversaries stick to their word". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles - SOSP'07*, p. 189–204, ACM: New York, NY, USA, 2007.

[8] FRAGA, J., POWELL, D., "A fault and intrusion-tolerant file system". In: *Proceedings of the 3rd International Conference on Computer Security*, p. 203–218, 1985.

[9] VERÍSSIMO, P., NEVES, N. F., CORREIA, M. P., "Intrusion-Tolerant Architectures: Concepts and Design", In: LEMOS, R., GACEK, C., ROMANOVSKY, A. (eds), *Architecting Dependable Systems*, v. 2677, *LNCS*, Springer-Verlag, 2003.

[10] AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., et al., "Basic Concepts and Taxonomy of Dependable and Secure Computing", *IEEE Transactions on Dependable and Secure Computing*, v. 1, n. 1, p. 11–33, March 2004.

[11] FISCHER, M. J., LYNCH, N. A., PATERSON, M. S., "Impossibility of Distributed Consensus with One Faulty Process", *Journal of the ACM*, v. 32, n. 2, p. 374–382, April 1985.

[12] LAMPORT, L., SHOSTAK, R., PEASE, M., "The Byzantine Generals Problem", *ACM Transactions on Programing Languages and Systems*, v. 4, n. 3, p. 382–401, July 1982.

[13] CORREIA, M., NEVES, N. F., LUNG, L. C., et al., "Worm–IT – A wormhole-based intrusion-tolerant group communication system", *Journal of Systems and Software*, v. 80, n. 2, p. 178–197, 2007.

[14] CABRI, G., LEONARDI, L., ZAMBONELLI, F., "Mobile-Agent Coordination Models for Internet Applications", *Computer*, v. 33, n. 2, p. 82–89, Feb. 2000.

[15] GELERNTER, D., CARRIERO, N., "Coordination Languages and their Significance", *Communications of ACM*, v. 35, n. 2, p. 96–107, Feb. 1992.

[16] GELERNTER, D., "Generative Communication in Linda", *ACM Transactions on Programing Languages and Systems*, v. 7, n. 1, p. 80–112, Jan. 1985.

[17] BESSANI, A. N., CORREIA, M., DA SILVA FRAGA, J., et al., "Sharing Memory between Byzantine Processes using Policy-enforced Tuple Spaces". In: *Proceedings of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*, July 2006.

[18] BESSANI, A. N., CORREIA, M., DA SILVA FRAGA, J., et al., "Sharing Memory between Byzantine Processes using Policy-Enforced Tuple Spaces", *IEEE Transactions on Parallel and Distributed Systems*, p. 1–12, 2008.

[19] KSHEMKALYANI, A. D., SINGHAL, M., *Distributed Computing: Principles, Algorithms and Systems*. Cambridge University Press, 2008.

[20] GARG, V. K., *Elements of Distributed Computing*. Wiley-IEEE Press, 2002.

[21] JALOTE, P., *Fault Tolerance in Distributed Systems*. Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 1994.

[22] COULOURIS, G., DOLLIMORE, J., KINDBERG, T., *Distributed Systems: Concepts and Design*. 4th ed. Addison-Wesley, 2005.

[23] GUERRAOUI, R., RODRIGUES, L., *Introduction to Reliable Distributed Programming*. Springer-Verlag: Berlin, Germany, 2006.

[24] ATTIYA, H., WELCH, J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. 2nd ed. *Wiley Series on Parallel and Distributed Computing*, Wiley-Interscience, 2004.

[25] LYNCH, N. A., *Distributed Algorithms*. Morgan Kauffman, 1996.

[26] CHANDY, K. M., LAMPORT, L., "Distributed snapshots: determining global states of distributed systems", *ACM Transactions Computer Systems*, v. 3, n. 1, p. 63–75, 1985.

[27] RAYNAL, M., MUKESH, S., "Logical Time: Capturing Causality in Distributed Systems", *Computer*, v. 29, n. 2, p. 49–56, Feb 1996.

[28] LUNG, L. C., *Experiências com Tolerância a Faltas no CORBA e Extensões no FT-CORBA para Sistemas Distribuídos de Larga Escala*, Tese de doutorado em engenharia elétrica, Universidade Federal de Santa Catarina, Florianópolis, 2001.

[29] CRISTIAN, F., "Understanding fault-tolerant distributed systems", *Communications of the ACM*, v. 34, n. 2, p. 56–78, 1991.

[30] CRISTIAN, F., FETZER, C., "The Timed Asynchronous Distributed System Model", *IEEE Transactions on Parallel and Distributed Systems*, v. 10, n. 6, p. 642–657, June 1999.

[31] DWORK, C., LYNCH, N. A., STOCKMEYER, L., "Consensus in the Presence of Partial Synchrony", *Journal of ACM*, v. 35, n. 2, p. 288–322, April 1988.

[32] POSTEL, J., "User Datagram Protocol (RFC 768)", IETF Request For Comments, Aug. 1980.

[33] HADZILACOS, V., TOUEG, S., *A Modular Approach to the Specification and Implementation of Fault-Tolerant Broadcasts and Related Problems*, Tech. Rep. TR 94-1425, Department of Computer Science, Cornell University, New York - USA, May 1994.

[34] RABIN, M. O., "Randomized Byzantine Generals". In: *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, p. 403–409, 1983.

[35] BRACHA, G., TOUEG, S., "Asynchronous consensus and broadcast protocols", *Journal of ACM*, v. 32, n. 4, p. 824–840, 1985.

[36] CHANDRA, T. D., TOUEG, S., "Unreliable Failure Detectors for Reliable Distributed Systems", *Journal of the ACM*, v. 43, n. 2, March 1996.

[37] ZIELINSKI, P., *Paxos at War*, Tech. Rep. UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK, June 2004.

[38] CHARRON-BOST, B., DÉFAGO, X., SCHIPER, A., "Broadcasting Messages in Fault-Tolerant Distributed Systems: the benefit of handling input-triggered and output-triggered suspicions differently". In: *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems - SRDS'02*, p. 244–249, 2002.

[39] GUERRAOUI, R., OLIVEIRA, R., SCHIPER, A., *Stubborn Communication Channels*, Tech. rep., LSE, D'epartement d'Informatique, Ecole Polytechnique F'ed'erale de.

[40] BASU, A., CHARRON-BOST, B., TOUEG, S., "Simulating reliable links with unreliable links in the presence of process crashes". In: *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG96)*, p. 105–122, 1996.

[41] MENEZES, A. J., VAN OORSCHOT, P. C., VANSTONE, S. A., *HandBook of Applied Cryptography*. CRC Press, Inc., October 1996.

[42] STALLINGS, W., *Cryptography and Network Security: Principles and Practice*. 2nd ed. Prentice Hall, 1999.

[43] DIERKS, T., ALLEN, C., "The TLS Protocol Version 1.0 (RFC 2246)", IETF Request For Comments, Jan. 1999.

[44] CHARRON-BOST, B., “Agreement Problems in Fault-Tolerant Distributed Systems”. In: *Proceedings of the 28th Conference on Current Trends in Theory and Practice of Informatics Piestany (SOFSEM)*, p. 10–32, Springer-Verlag: London, UK, 2001.

[45] PEASE, M., SHOSTAK, R., LAMPORT, L., “Reaching Agreement in the Presence of Faults”, *Journal of ACM*, v. 27, n. 2, p. 228–234, April 1980.

[46] RAYNAL, M., SINGHAL, M., “Mastering Agreement Problems in Distributed Systems”, *IEEE Software*, v. 18, n. 4, p. 40–47, 2001.

[47] CHARRON-BOST, B., SCHIPER, A., “Uniform consensus is harder than consensus”, *Journal of Algorithms*, v. 51, n. 1, p. 15–37, 2004.

[48] FISHER, M. J., *The Consensus Problem in Unreliable Distributed Systems (A Brief Survey)*, Tech. Rep. YALEU/DCS/TR-273, Yale University, USA, June 1983.

[49] CHAUDHURI, S., “More choices allow more faults: Set consensus problems in totally asynchronous systems”, *Information and Computation*, v. 105, n. 1, p. 132–158, July 1993.

[50] DOLEV, D., STRONG, H. R., “Authenticated Algorithms for Byzantine Agreement”, *SIAM Journal on Computing*, v. 12, n. 4, p. 656–666, 1983.

[51] TOUEG, S., “Randomized Byzantine Agreements”. In: *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, p. 163–178, 1984.

[52] DOUDOU, A., GARBINATO, B., GUERRAOUI, R., “Encapsulating Failure Detection: From Crash-Stop to Byzantine Failures”. In: *International Conference on Reliable Software Technologies*, p. 24–50, Springer-Verlag: Viena - Austria, 2002.

[53] DOUDOU, A., SCHIPER, A., “Muteness detectors for consensus with Byzantine processes”. In: *Proceedings of the 17th annual ACM Symposium on Principles of Distributed Computing (Brief announcements)*, p. 315, 1998.

[54] GUERRAOUI, R., SCHIPER, A., “Software-Based Replication for Fault Tolerance”, *Computer*, v. 30, n. 4, p. 68–74, 1997.

[55] HERLIHY, M., WING, J. M., "Linearizability: A Correctness Condition for Concurrent Objects", *ACM Transactions on Programing Languages and Systems*, v. 12, n. 3, p. 463–492, July 1990.

[56] BUDHIRAJA, N., MARZULLO, K., SCHNEIDER, F. B., et al., "The primary-backup approach", , p. 199–2161993.

[57] ALPERN, B., SCHNEIDER, F. B., "Recognizing Safety and Liveness", *Distributed Computing*, v. 2, n. 3, p. 117–126, 1987.

[58] GARCIA-MOLINA, H., "Elections in a Distributed Computing System", *IEEE Transactions on Computers*, v. C-31, n. 1, p. 48–59, 1982.

[59] CHANG, E., ROBERTS, R., "An improved algorithm for decentralized extrema-finding in circular configurations of processes", *Communications of the ACM*, v. 22, n. 5, p. 281–283, 1979.

[60] DÉFAGO, X., SCHIPER, A., URBÁN, P., "Total order broadcast and multicast algorithms: Taxonomy and survey", *ACM Computing Surveys*, v. 36, n. 4, p. 372–421, Dec. 2004.

[61] POWELL, D., *Delta-4 Architecture Guide*. Esprit II P2252, august 1991.

[62] CHÉRÈQUE, M., POWELL, D., REYNIER, P., et al., "Active Replication in Delta-4". In: *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing (FTCS)*, p. 28–37.

[63] DÉFAGO, X., SCHIPER, A., SERGENT, N., "Semi-Passive Replication". In: *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, p. 43, IEEE Computer Society: Washington, DC, USA, 1998.

[64] DÉFAGO, X., SCHIPER, A., "Semi-passive replication and Lazy Consensus", *Journal of Parallel and Distributed Computing*, v. 64, p. 1380–1398, 2004.

[65] CASTRO, M., LISKOV, B., "Practical Byzantine Fault Tolerance". In: *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, p. 173–186, USENIX Association, Feb. 1999.

[66] ÖZALP BABAOĞLU, DAVOLI, R., MONTRESOR, A., "Group membership and view synchrony in partitionable asynchronous distributed systems: specifications", *SIGOPS Operating Systems Review*, v. 31, n. 2, p. 11–22, 1997.

[67] SCHNEIDER, F. B., "Replication management using the state-machine approach", , p. 169–1971993.

[68] YIN, J., MARTIN, J.-P., VENKATARAMANI, A., et al., "Towards a Practical Approach to Confidential Byzantine Fault Tolerance", In: *Future Directions in Distributed Computing*, v. 2584, p. 51–56, *LNCS*, Springer Berlin/Heidelberg, 2003.

[69] NIGHTINGALE, E. B., CHEN, P. M., FLINN, J., "Speculative execution in a distributed file system". In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles - SOSP'05*, p. 191–205, ACM: New York, NY, USA, 2005.

[70] BESSANI, A. N., DA SILVA FRAGA, J., LUNG, L. C., "Coordenação Desacoplada Tolerante a Faltas Bizantinas". In: *Anais do 6o Workshop de Testes e Tolerância a Faltas - WTF 2005*, may 2005.

[71] PAPADOPOLOUS, G., ARBAB, F., "Coordination Models and Languages", In: *The Engineering of Large Systems*, v. 46, *Advances in Computers*, Academic Press, Aug. 1998.

[72] GELERNTER, D., BERNSTEIN, A. J., "Distributed Communication via Global Buffer". In: *Proceedings of the 1st Annual ACM Symposium on Principles of Distributed Computing - PODC'82*, p. 10–18, Aug. 1982.

[73] BAUMANN, J., HOHL, F., ROTHERMEL, K., et al., "Mole – Concepts of a mobile agent system", *World Wide Web*, v. 1, n. 3, p. 123–137, 1998.

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

[75] XU, A., LISKOV, B., "A design for a fault-tolerant, distributed implementation of Linda". In: *Proceedings of the 19th Symposium on Fault-Tolerant Computing - FTCS'89*, p. 199–206, June 1989.

[76] BAKKEN, D. E., SCHLICHTING, R. D., "Supporting Fault-Tolerant Parallel Programing in Linda", *IEEE Transactions on Parallel and Distributed Systems*, v. 6, n. 3, p. 287–302, March 1995.

[77] VITEK, J., BRYCE, C., ORIOL, M., "Coordination Processes with Secure Spaces", *Science of Computer Programming*, v. 46, n. 1-2, p. 163–193, Jan. 2003.

[78] SEGALL, E. J., "Resilient Distributed Objects: Basic Results and Applications to Shared Spaces". In: *Proceedings of the 7th Symposium on Parallel and Distributed Processing - SPDP'95*, p. 320–327, Oct. 1995.

[79] HERLIHY, M., "Wait-Free Synchronization", *ACM Transactions on Programing Languages and Systems*, v. 13, n. 1, p. 124–149, Jan. 1991.

[80] BESSANI, A. N., ALCHIERI, E. P., CORREIA, M., et al., "DepSpace: a byzantine fault-tolerant coordination service". In: *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, p. 163–176, ACM: New York, NY, USA, 2008.

[81] BESSANI, A. N., ALCHIERI, E., CORREIA, M. P., et al., "DepSpace: Um Middleware para Coordenação em Ambientes Dinâmicos e Não Confiáveis". In: *Salão de Ferramentas do XXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, SBC, May 2007.

[82] SCHOENMAKERS, B., "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*, p. 148–164, Aug. 1999.

[83] CODEHAUS, "Groovy programing language homepage", Disponível em *http://groovy.codehaus.org/*, 2006.

[84] LUIZ, A. F., BESSANI, A. N., LUNG, L. C., et al., "RePEATS: Uma Arquitetura para Replicação Tolerante a Faltas Bizantinas baseada em Espaço de Tuplas". In: *Anais do XXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, SBC, May 2008.

[85] BIRMAN, K. P., JOSEPH, T. A., "Exploiting virtual synchrony in distributed systems". In: *Proceedings of the eleventh ACM Symposium on Operating systems principles*, p. 123–138, 1987.

[86] AGUILERA, M. K., MERCHANT, A., SHAH, M., et al., "Sinfonia: a new paradigm for building scalable distributed systems". In: *Proceedings of the 21st ACM Symposium on Operating Systems Principles - SOSP'07*, 2007.

[87] BIRMAN, K. P., JOSEPH, T. A., RAEUCHLE, T., et al., "Implementing Fault-Tolerant Distributed Objects", *IEEE Transactions on Software Engineering*, v. 11, n. 6, p. 502–508, 1985.

[88] ELNOZAHY, M., ALVISI, L., WANG, Y.-M., et al., "A survey of rollback-recovery protocols in message-passing systems", *ACM Computing Surveys*, v. 34, n. 3, p. 375–408, 2002.

[89] HITCHENS, R., *Java NIO*. O'Reilly, 2002.

[90] SUN MICROSYSTEMS, I., "NFS: Network File System Protocol Specification (RFC 1094)", IETF Request For Comments, Mar 1989.

[91] CALLAGHAN, B., "WebNFS Client Specification (RFC 2054)", IETF Request For Comments, Oct. 1996.

[92] CALLAGHAN, B., "WebNFS Server Specification (RFC 2055)", IETF Request For Comments, Oct. 1996.