Universidade Federal da Paraíba
Centro de Ciências e Tecnologia
Departamento de Sistemas e Computação
Coordenação de Pós-Graduação em Informática

Provendo Serviços para Tolerância a Faltas em Sistemas Distribuídos

Sheila Regine Almeida Vasconcelos

Dissertação de Mestrado

Campina Grande - PB
Julho/1997

Sheila Regine Almeida Vasconcelos

Provendo Serviços para Tolerância a Faltas em Sistemas Distribuídos

Dissertação apresentada ao Curso de Mestrado em Informática da Universidade Federal da Paraíba como requisito parcial à obtenção do título de Mestre em Informática.

Área de Concentração: Redes de Computadores e Sistemas Distribuídos

Orientador: Francisco Vilar Brasileiro

Universidade Federal da Paraíba
Campina Grande
Julho de 1997

V331p Vasconcelos, Sheila Regine Almeida
Provendo servicos para tolerancia a faltas em sistemas distribuidos / Sheila Regine Almeida Vasconcelos. - Campina Grande, 1997.
168 f.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Redes de Computacao 2. Sistemas Distribuidos 3. Dissertacao I. Brasileiro, Francisco Vilar, Dr. II. Universidade Federal da Paraiba - Campina Grande (PB)

CDU 004.738(043)

Sheila Regine Almeida Vasconcelos

Provendo Serviços para Tolerância a Faltas em Sistemas Distribuídos

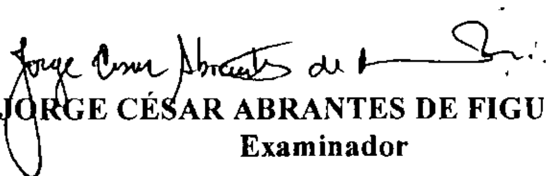
Julho de 1997

PROVENDO SERVIÇOS PARA TOLERÂNCIA A FALTAS EM SISTEMAS DISTRIBUÍDOS

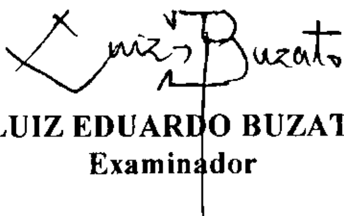
SHEILA REGINE ALMEIDA VASCONCELOS

DISSERTAÇÃO APROVADA EM 02.07.1997

PROF. FRANCISCO VILAR BRASILEIRO, Dr.
Presidente



PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO, D.Sc
Examinador



PROF. LUIZ EDUARDO BUZATO, Ph.D
Examinador

CAMPINA GRANDE - PB

Agradecimentos

A Deus, em primeiro lugar, a quem devo mais esta vitória, por ter-me dado força e luz nos tantos momentos de dificuldade encontrados durante o desenvolvimento deste trabalho.

A meu pai, Heronides, por estar espiritualmente presente em cada instante da minha vida, dando-me proteção e ânimo.

A minha mãe, Maria, e a minha irmã, Magdala, pela compreensão nos muitos momentos em que não estive disponível devido à falta de tempo.

A meu marido, José Barreto, pelos inúmeros momentos de lazer sacrificados em nome deste trabalho.

A meu orientador, Prof. Francisco Vilar Brasileiro, por ter-me confiado esta responsabilidade, e a toda a equipe do Projeto Seljuk, especialmente a Érica, pelo apoio.

Aos professores Jorge Figueiredo e Eduardo Buzzato, componentes desta banca, pela presteza em avaliar o trabalho em tão curto prazo.

Aos professores e funcionários do Departamento de Sistemas e Computação e da Coordenação de Pós-Graduação em Informática, por seu trabalho em prol dos alunos.

Aos amigos do Núcleo Setorial de Computação, pelo apoio, em especial a José Jonas de Oliveira e Camilo de Lélis Gondim, por terem me cedido tempo para desenvolvimento deste trabalho, e a Jailda, pela incansável amizade e presteza.

Enfim, a todos que, direta ou indiretamente, contribuíram, com atos ou palavras, para a concretização deste trabalho.

Sumário

A obtenção de confiança no funcionamento em aplicações distribuídas depende da utilização de mecanismos apropriados para tolerância a faltas. A arquitetura Seljuk se propõe a fornecer um ambiente propício ao desenvolvimento e à execução de aplicações distribuídas robustas. Alguns dos principais serviços para tolerância a faltas usados na construção de aplicações distribuídas robustas são discutidos, e a forma como estes serviços são oferecidos pelo ambiente operacional Seljuk-Amoeba é descrita. Uma proposta de terminologia em português para área da confiança no funcionamento e outras áreas da Ciência da Computação também é apresentada.

Palavras-chaves: sistemas distribuídos, tolerância a faltas, confiança no funcionamento, replicação, comunicação em grupo, diagnóstico de faltas, reconfiguração do sistema.

Abstract

Fault tolerance mechanisms must be introduced into systems if a dependability is to be attained. The Seljuk architecture provides a number of services to facilitate the implementation and support the execution of dependable distributed applications. This dissertation discusses the main fault tolerance services used by dependable distributed applications and how these services are provided by the Seljuk-Amoeba operating environment. Furthermore, a terminology proposal in portuguese language for the area of dependability and other areas in the Computing Science is presented.

Keywords: distributed systems, fault tolerance, dependability, replicated processing, group communication, fault diagnosis, system reconfiguration.

Índice

Agradecimentos i

Sumário ii

Abstract iii

Índice iv

Lista de Figuras viii

Lista de Tabelas ix

Capítulo 1: Introdução 01

1.1. Provendo Facilidades para Implementação de Tolerância a Faltas 03

1.1.1. Ferramentas para Construção de Sistemas Tolerantes a Faltas 03

1.1.2. Impondo Restrições à Semântica de Falha dos Componentes do Sistema 05

1.2. O Projeto Seljuk 07

1.2.1. Objetivos 08

1.2.2. Ambiente de Desenvolvimento 08

1.3. Estrutura da Dissertação 09

Capítulo 2: Conceitos de Tolerância a Faltas 11

2.1. Introdução 11

2.2. Sistemas Tolerantes a Faltas 12

2.3. Computação Distribuída e Tolerância a Faltas 14

2.4. Confiança no Funcionamento 16

2.5. Fases da Tolerância a Faltas 17

2.5.1. Detecção, Compensação e Correção do Erro 18

2.5.2. Confinamento e Avaliação do Dano 21

2.5.3. Recuperação do Sistema 22

2.5.4. Tratamento da Falta 23

2.6. Serviços para Tolerância a Faltas 25

Capítulo 3: Replicação de Componentes de *Software* 27

3.1. Introdução 27

3.2. Semântica de Falha do Serviço de Processamento e das Réplicas 28

3.3. Determinismo de Réplicas e do Grupo de Réplicas 31

3.4. Domínio de Replicação 32

3.5. Estratégias de Processamento do Erro 33

3.6. Replicação Ativa 34

3.6.1. A Abordagem da Máquina de Estado 35

3.6.2. Processamento do Erro 37

3.7. Replicação Passiva 39

3.7.1. Estratégias de Salvaguarda 40

3.7.2. Otimizando os Mecanismos de Salvaguarda 44

3.7.3. Processamento do Erro 45

3.8. Replicação Semi-Ativa 46

3.8.1. O Modelo de Replicação Líder-Seguidor 47

3.8.2. Sistema de Comunicação 48

3.8.3. Processamento do Erro 49

3.9. Sumário 50

Capítulo 4: Comunicação em Grupo 52

4.1. Introdução 52

4.2. Noções sobre Grupos de Processos 53

4.3. Noções sobre Comunicação em Grupo 55

4.3.1. Suporte para Replicação 56

4.3.2. Suporte para Grupos 57

4.4. Requisitos para Transparência de Grupos de Réplicas 58

4.5. Propriedades da Comunicação em Grupo 60

4.5.1. Acordo 61

4.5.2. Ordenação 62

4.6. Questões de Desenho em Comunicação em Grupo 65

4.7. Sumário 67

Capítulo 5: Diagnóstico da Falta 69

5.1. Introdução 69

5.2. Diagnóstico de Falta Distribuído 70

5.3. Diagnóstico de Falta para Sistemas Distribuídos com Falha e Reparo Dinâmicos 72

5.3.1. Modelo do Sistema 72

5.3.2. Algoritmo NEW_SELF 73

5.3.3. Análise do Algoritmo NEW_SELF 77

5.4. Diagnóstico Adaptativo a Nível de Sistema Distribuído 78

5.4.1. Modelo do Sistema 79

5.4.2. Algoritmo DSD Adaptativo 79

5.4.3. Análise do Algoritmo Adaptativo 82

5.5. Diagnóstico de Nodos com Semântica de Falha Bizantina 82

5.5.1. Modelo Básico 84

5.5.2. Algoritmo de Diagnóstico Parcial de Faltas Bizantinas 85

5.5.3. Análise do Algoritmo D 87

5.6. Sumário 88

Capítulo 6: Reconfiguração do Sistema 90

6.1. Introdução 90

6.2. Conceito de Unidades de Substituição 91

6.3. Realocação de Componentes de *Software* 92

6.4. Reconfiguração com Replicação Ativa 93

6.4.1. Condição de Combinação e Tolerância Ilimitada 94

6.4.2. Gerência da Configuração 95

6.4.3. Integração de um Objeto Recuperado 97

6.5. Reconfiguração com Replicação Passiva e Semi-Ativa 98

6.6. Sumário 99

Capítulo 7: Tolerância a Faltas no Seljuk-Amoeba 100

7.1. Introdução 100

7.2. A Arquitetura Seljuk 101
7.2.1. Objetivos 101
7.2.2. Estrutura 101
7.2.3. Ambiente de Desenvolvimento 102
7.3. O Sistema Operacional Distribuído Amoeba 103
7.3.1. Arquitetura do Amoeba 104
7.3.2. O Modelo de *Software* do Amoeba 105
7.3.3. Comunicação Interprocessos 107
7.4. Pressupostos 109
7.5. Serviço de Comunicação em Grupo 110
7.5.1. Comunicação em Grupo no Amoeba 110
7.5.2. Comunicação em Grupo no Seljuk-Amoeba 112
7.6. Serviço de Replicação 119
7.6.1. Serviço de Replicação Ativa 126
7.6.2. Serviço de Replicação Passiva 127
7.6.3. Serviço de Replicação Semi-Ativa 130
7.6.4. Exemplos de Processamento Replicado no Seljuk-Amoeba 134
7.7. Serviço de Diagnóstico de Falta e Detecção de Falhas 137
7.8. Serviço de Reconfiguração do Sistema 139
7.9. Sumário 141
Capítulo 8: Conclusões 144
8.1. Discussão 144
8.2. Trabalhos Relacionados 149
8.3. Direções para Trabalhos Futuros 150
8.4. Contribuições 151
Apêndice A: Terminologia Estendida para Confiança no Funcionamento 153
Referências Bibliográficas 159

Lista de Figuras

Figura 5.1: O Algoritmo DSD Adaptativo 80

Figura 5.2: O Algoritmo de Diagnóstico 81

Figura 7.1: Estrutura do Ambiente Operacional Seljuk 102

Figura 7.2: Camadas do Sistema de Comunicação do Seljuk-Amoeba 118

Figura 7.3: Sistema de Arquivos do Seljuk-Amoeba 121

Figura 7.4: Replicação de Processos no Seljuk-Amoeba 126

Figura 7.5: Linha de Execução Ativa de um Servidor Replicado Passivamente 136

Figura 7.6: Linha de Execução Passiva de um Servidor Replicado Passivamente .. 136

Figura 7.7: Aplicação Replicada de Forma Semi-Ativa 137

Lista de Tabelas

Tabela 3.1: Principais Características das Técnicas de Replicação 51

Tabela 4.1: Principais Questões de Desenho para Comunicação em Grupo 66

Tabela 7.1: Primitivas para Comunicação em Grupo do Amoeba 111

Tabela 7.2: Primitivas para Comunicação em Grupo do Seljuk-Amoeba 119

Tabela 7.3: Funções para Tolerância a Faltas do Seljuk-Amoeba 135

Tabela 7.4: Suporte do Seljuk-Amoeba para Processamento Replicado 142

Tabela A.1: Terminologia Proposta por Lemos e Veríssimo 153

Tabela A.2: Terminologia Proposta por Vasconcelos e Brasileiro 157

Capítulo 1:

Introdução

A partir de meados dos anos 80, o uso de computadores vem sofrendo uma revolução graças a dois extraordinários avanços tecnológicos. O primeiro deles foi o desenvolvimento de poderosos microprocessadores, muitos dos quais com poder computacional equivalente ou mesmo superior ao de um *mainframe*, porém custando apenas uma fração do seu preço.

O segundo avanço ocorreu na área de comunicações, com o surgimento das redes locais de alta velocidade, que permitem que dezenas, ou até centenas, de máquinas sejam conectadas de tal forma que grandes quantidades de dados possam ser transferidos entre elas a uma velocidade de 10 milhões de bits/segundo ou mais [Tanenbaum 92, pp. 362]. Como exemplo, podemos citar as novas tecnologias baseadas em fibra ótica, que podem atingir taxas de transferência superiores a 100 Mbps. É o caso da tecnologia ATM (*Asynchronous Transfer Mode*), que surge como uma solução para a interconexão de altíssima velocidade em ambientes locais (até alguns Gbps), além de dar suporte à integração de diferentes serviços, satisfazendo às exigências dos diversos tipos de tráfego (texto, áudio, vídeo etc.).

A aliança entre estes dois avanços tecnológicos propiciou o surgimento dos chamados *sistemas distribuídos*, compostos por inúmeros processadores interconectados por uma rede de comunicação de alta velocidade. A partir de então, tais sistemas vêm ganhando mais e mais popularidade.

A principal razão para a crescente tendência em torno das soluções baseadas em processamento distribuído reside no fato de os sistemas deste tipo apresentarem uma relação custo/benefício muito melhor do que aquela oferecida por um único e grande sistema centralizado. Uma coleção de microprocessadores pode não apenas oferecer uma razão custo/desempenho melhor do que um único *mainframe*, como também pode produzir um desempenho absoluto que nenhum *mainframe* poderá alcançar a preço algum.

Uma segunda razão é que algumas aplicações apresentam características inerentemente distribuídas. Em um sistema de automação industrial, por exemplo, que controla robôs e máquinas numa linha de montagem, é possível associar a cada um destes componentes o seu próprio computador; cada computador, por sua vez, precisa interagir com alguns ou todos os demais computadores, o que conduz à necessidade de interconectá-los, formando assim um sistema industrial distribuído [Tanenbaum 92, pp. 364].

Finalmente, um sistema distribuído, desde que bem projetado, pode potencialmente oferecer um nível de confiança no funcionamento muito maior do que aquele fornecido por um sistema centralizado. Confiança no funcionamento é o termo sugerido em [Lemos-Veríssimo 91] como tradução para o termo em inglês *dependability* introduzido em [Laprie 85]. Entretanto, se não forem utilizadas técnicas adequadas para tolerância a faltas, a distribuição da computação sobre múltiplas unidades de processamento pode até mesmo conduzir ao decréscimo da confiança no funcionamento visto que, na maioria dos casos, a computação só poderá prosseguir se todas as unidades envolvidas estiverem operacionais.

Diversos mecanismos vêm sendo desenvolvidos para garantir tolerância a faltas sob uma variedade de ambientes. Porém, na grande maioria dos casos, a implementação destes mecanismos fica a cargo da aplicação, com pouco ou nenhum suporte do sistema operacional, o que aumenta consideravelmente a complexidade do desenvolvimento de tais aplicações. Realmente, um dos principais problemas encontrados no desenvolvimento de aplicações distribuídas robustas, isto é, aquelas aplicações que possuem requisitos consideráveis de confiança no funcionamento, reside justamente na dificuldade introduzida pela necessidade de se tolerar e tratar faltas [Cristian 91].

Uma possível maneira de reduzir esta complexidade é oferecer os mecanismos para tolerância a faltas sob a forma de serviços, que poderiam ser utilizados diretamente pela aplicação sem que ela própria tivesse que implementá-los.

Nossa proposta é facilitar o desenvolvimento de sistemas críticos, provendo meios para que os seus projetistas construam aplicações distribuídas tolerantes a faltas sem terem de se preocupar com os serviços básicos que apóiam estas construções. Ou seja, pretendemos tornar os serviços para tolerância a faltas disponíveis e transparentes para a aplicação e, dessa forma, minimizar o trabalho do projetista. Buscamos também oferecer

flexibilidade suficiente para que a utilização de tais serviços gere ônus apenas para as aplicações que desejem, ou necessitem, fazer uso deles.

1.1. Provendo Facilidades para Implementação de Tolerância a Faltas

Como foi dito anteriormente, na maioria das vezes, os mecanismos para tolerância a faltas são implementados pela própria aplicação, tornando difícil a tarefa de construir aplicações distribuídas robustas. A literatura corrente, entretanto, mostra que os projetistas deste tipo de aplicação têm buscado reduzir a complexidade do seu trabalho através da utilização de duas abordagens complementares:

(i) a utilização de ferramentas de apoio à construção de sistemas tolerantes a faltas, tais como: bibliotecas de funções [Huang-Kintala 93], *toolkits* de programação [Birman 85, Shrivastava et al. 91a] e serviços especializados do sistema operacional [Ng 90, Mullender et al. 90]; e

(ii) a restrição da semântica de falha dos componentes que formam a infra-estrutura de processamento e comunicação sobre a qual a aplicação será executada [Cristian 91].

Nas subseções seguintes, discutiremos como as duas estratégias têm sido empregadas por alguns sistemas apresentados na literatura.

1.1.1. Ferramentas para Construção de Sistemas Tolerantes a Faltas

Os primeiros sistemas tolerantes a faltas eram soluções *ad-hoc* aplicáveis a problemas específicos [Hopkins et al. 78, Wensley et al. 78, Lala 86]. Com o aumento do número de aplicações com requisitos de confiança no funcionamento, surge a necessidade de se introduzir ferramentas que possibilitem a construção destes sistemas com o reaproveitamento do esforço desprendido na construção de outros sistemas com características similares. A seguir, apresentamos resumidamente algumas ferramentas propostas na literatura para tal fim.

<u>Bibliotecas de Funções para Tolerância a Faltas</u>

Uma forma bastante comum de se promover reusabilidade de *software* é através da utilização de bibliotecas de funções. Com pouco esforço de programação, é possível

utilizar mecanismos para tolerância a faltas que já tenham sido usados e, principalmente, testados por outras aplicações.

[Huang-Kintala 93] apresenta uma biblioteca de funções C, denominada `libft`, que oferece funções para especificar e realizar salvaguarda (*checkpoint*) de dados críticos da aplicação, recuperar a informação das salvaguardas, atualizar diário de eventos (*log*), localizar e reconectar servidores, realizar tratamento de exceções etc. A função `critical()`, por exemplo, pode ser usada para especificar dados críticos da aplicação, que deverão fazer parte da salvaguarda ativada pela função `checkpoint()`. Além disso, tal biblioteca provê suporte para programação diversificada [Avizienis 85] e uso de blocos de recuperação [Randell 75]. Tais facilidades podem ser usadas por qualquer aplicação sem necessidade de se mudar o sistema operacional ou adicionar novos pré-processadores C. [Huang-Kintala 93] apresenta alguns exemplos de utilização das funções da biblioteca `libft` na construção de aplicações tolerantes a faltas.

Toolkits de Programação

A consolidação do paradigma de orientação a objetos para o desenvolvimento de sistemas de *software* fez emergir ambientes especializados, denominados *toolkits* de programação, que se baseiam na idéia de usar as facilidades de herança das linguagens de programação orientadas a objetos para prover classes de objetos que implementam mecanismos para tolerância a faltas. A aplicação pode então ter acesso a estes mecanismos através da utilização de subclasses derivadas daquelas classes providas pelo *toolkit* [Brasileiro 97].

O sistema Arjuna [Shrivastava et al. 91a] é um exemplo deste tipo de ferramenta. O seu modelo computacional se baseia na utilização do conceito de transações atômicas aninhadas que controlam as operações realizadas sobre objetos persistentes. A complexidade do tratamento da distribuição e da persistência dos objetos que implementam a aplicação é escondido do programador pela infra-estrutura provida pelo Arjuna.

Recentemente, o conceito de reflexão computacional [Maes 87] tem se unido ao modelo de orientação a objetos para auxiliar a construção de aplicações distribuídas robustas. De uma forma simplificada, a idéia é dividir o sistema em uma parte funcional

(*objetos da aplicação*) e outra parte de controle e gerência (*meta-objetos*). Em [Lisbôa-Cavalheiro 95], esta técnica é utilizada para prover tolerância a faltas de *software*.

Serviços Especializados de Sistemas Operacionais Distribuídos

O projeto dos sistemas operacionais modernos, especialmente dos sistemas operacionais distribuídos, tem sido influenciado pela necessidade de se desenvolver aplicações que tolerem faltas que possam afetar a sua infra-estrutura de execução. A maior parte destes sistemas implementa serviços de comunicação com variável grau de confiabilidade [Mullender et al. 90, Cheriton-Zwaenpoel 85, Rozier 92], enquanto outros vão mais além e oferecem serviços de processamento confiável [Ng 90].

No sistema ROSE [Ng 90], por exemplo, um serviço de detecção de falhas é provido pelo núcleo do sistema. Um processo pode usar este serviço para detectar a falha de um outro processo, possivelmente executando em um outro processador do sistema distribuído. O sistema também oferece um serviço de replicação do espaço de endereçamento, que permite tornar uma porção do espaço de endereçamento de um processo disponível para outros processos. Finalmente, o sistema provê uma abstração de processo resiliente, que, apoiada nos dois serviços anteriores, possibilita a execução replicada de processos para propósitos de tolerância a faltas.

1.1.2. Impondo Restrições à Semântica de Falha dos Componentes do Sistema

A complexidade dos mecanismos para tolerância a faltas reflete as considerações feitas sobre a semântica de falha dos componentes do sistema utilizados na execução destes mecanismos [Cristian 91]. Quanto mais restritiva a semântica de falha dos componentes, mais simples será a implementação dos mecanismos. Levando-se ao extremo, se os componentes nunca falham (semântica de falha mais restritiva possível), não há necessidade de adicionar à aplicação qualquer mecanismo para tolerância a faltas.

Muitos sistemas distribuídos apresentados na literatura (p. ex. [Birman 85, Kopetz-Merker 85, Ng 90, Shrivastava et al. 91a, Powell 92]) foram construídos assumindo que os componentes que formam a sua infra-estrutura de processamento e comunicação apresentam um comportamento de falha bem definido e previsível, isto é, possuem uma semântica de falha controlada (*fail-controlled* [Laprie 89]).

Quando a infra-estrutura disponível não pode prover, com probabilidade suficientemente alta, a semântica de falha assumida, faz-se necessário construir unidades de processamento, denominadas *nodos*, que de fato apresentem a semântica requerida. Nodos com semântica de falha controlada podem ser construídos através do agrupamento de processadores convencionais que falham de maneira independente. A computação é replicada e executada simultaneamente em cada um dos processadores formando o nodo. Os resultados da computação de cada um dos processadores são, então, validados por um mecanismo apropriado (p. ex., comparação, votação majoritária) que garante a semântica de falha do nodo [Brasileiro 95].

A implementação de um nodo envolve basicamente um protocolo de sincronização dos processadores redundantes e um protocolo de validação do resultado das operações realizadas por estes processadores. Tal implementação pode ser feita tanto em *hardware* [Hopkins et al. 78, Lala 86, Bernstein 88, Webber-Beirne 91] quanto em *software* [Wensley et al. 78, Brasileiro 95, Brasileiro et al. 96].

A abordagem em *software* é muito mais flexível do que aquela em *hardware*, permitindo inclusive a utilização de diferentes tipos de processadores (*diversidade de projeto*) em um mesmo nodo, o que possibilita o tratamento de faltas de projeto (daqui por diante chamadas de *faltas de concepção* [Lemos-Veríssimo 91]) dos processadores. Além disso, a atualização tecnológica dos nodos é relativamente fácil, pois os protocolos implementados em *software* precisam apenas ser recompilados para a nova plataforma. Por outro lado, o principal problema da abordagem de *software* é a possibilidade de uma queda acentuada no desempenho do sistema [Brasileiro 97].

[Brasileiro 95] apresenta uma forma eficiente de se construir em *software* diversos tipos de nodos com semântica de falha controlada e identifica classes de aplicações para as quais a queda de desempenho do sistema, decorrente da utilização de mecanismos para tolerância a faltas, é bastante reduzida. Os protocolos apresentados são implementados por uma coleção de funções pertencentes a uma biblioteca, que são ligadas ao código da aplicação. A aplicação deve, então, ser executada no número de processadores necessários para garantir o grau de confiança no funcionamento requerido. Grande parte da gerência dos mecanismos para tolerância a faltas, porém, continuam sob a responsabilidade do programador, o que além de ser trabalhoso, é susceptível à introdução de faltas de concepção. Estes problemas podem ser

solucionados se o ambiente operacional no qual à aplicação está sendo desenvolvida/executada oferecer serviços que tornem tais mecanismos transparentes para a aplicação.

1.2. O Projeto Seljuk

Esta dissertação faz parte do projeto de pesquisa intitulado "*Implementando Aplicações Distribuídas Robustas utilizando os Serviços para Tolerância a Faltas de Micronúcleos Distribuídos*" [Brasileiro 96], cujo objetivo principal é construir uma plataforma de desenvolvimento que facilite a construção e execução de aplicações distribuídas robustas.

Durante os séculos X e XI, os seljúquios desenvolveram um exército altamente efetivo, responsável, entre outras conquistas, pela subjugação da armada Bizantina. Além disso, os seljúquios estabeleceram um Estado altamente coeso e bem administrado. Em homenagem ao povo seljúquio e a seu Estado, a plataforma em consideração foi batizada de *Seljuk*.

Tal plataforma deverá incorporar os protocolos necessários para garantir diferentes semânticas de falha para a infra-estrutura de processamento do sistema distribuído, liberando as aplicações da tarefa de implementar os mecanismos que garantam a semântica de falha por elas requerida. Adicionalmente, a plataforma oferecerá serviços que implementam mecanismos básicos para tolerância a faltas, os quais poderão ser usados diretamente pelos projetistas na construção das aplicações distribuídas robustas.

Finalmente, o projeto incluirá o desenvolvimento e a implementação de aplicações distribuídas robustas, que serão usadas para avaliar a redução do grau de complexidade de programação *versus* o custo de processamento resultantes do uso dos serviços oferecidos pela plataforma. A plataforma Seljuk deverá ser bastante flexível, possibilitando que aplicações com diferentes requisitos de confiança no funcionamento coexistam num mesmo sistema distribuído, mas pagando apenas por aqueles serviços de tolerância a faltas por elas requisitados.

1.2.1. Objetivos

O objetivo específico deste trabalho de dissertação é estudar e projetar uma maneira de oferecer os serviços de *replicação dos componentes de software, comunicação em grupo, diagnóstico de faltas* e *reconfiguração do sistema*, de forma a torná-los o mais transparentes possível para a aplicação, permitindo que os desenvolvedores de sistemas críticos se libertem da obrigação de construir, eles próprios, os mecanismos para tolerância a faltas e possam voltar seus esforços para a implementação do serviço real que o sistema se propõe a fornecer.

Um segundo objetivo é fornecer um material de leitura fácil e agradável, a ser utilizado por pessoas interessadas na área de *tolerância a faltas em sistemas distribuídos*, que desejem obter informação organizada sobre conceitos básicos de tolerância a faltas e sobre alguns dos principais serviços para tolerância a faltas. Os cinco capítulos seguintes tratam em detalhes destes dois aspectos.

Por último, um outro objetivo deste trabalho é sugerir uma terminologia em português que estende aquela apresentada em [Lemos-Veríssimo 91], como uma possível proposta de formalização, para a língua portuguesa, da nomenclatura na área de confiança no funcionamento, a exemplo do que já existe para outros idiomas [Laprie 92]. Aqui, apresentamos sugestões apenas para os termos, conceitos e definições que aparecem naturalmente no decorrer deste texto e que são normalmente referenciados no nosso cotidiano pelos seus nomes em inglês. No Apêndice A, é apresentado um glossário dos termos apresentados em [Lemos-Veríssimo 91] e dos que serão introduzidos ao longo do texto, relacionando-os com os seus respectivos termos em inglês e com uma pequena descrição dos mesmos.

1.2.2. Ambiente de Desenvolvimento

A plataforma Seljuk [Brasileiro 97] está sendo desenvolvida a partir de um sistema operacional distribuído de domínio público, baseado na tecnologia de micronúcleos, cujo princípio básico é minimizar o tamanho da parte do sistema operacional que executa em modo supervisor com o objetivo de aumentar a flexibilidade do sistema. Toda a funcionalidade do sistema operacional que não é provida pelo micronúcleo fica a cargo de processos servidores que executam em modo usuário e, portanto, podem ser

modificados/adaptados/configurados muito mais facilmente para atender às exigências das diferentes aplicações [Tanenbaum 92, pp. 388-389 e 594].

Dada a importância que a comunicação tem nos sistemas operacionais distribuídos baseados em micronúcleos, a maioria destes sistemas oferece um serviço de comunicação com algum grau de confiabilidade. No entanto, serviços de processamento confiáveis são praticamente inexistentes. Na maioria dos casos, o processamento é no máximo tão confiável quanto as unidades nas quais ele é executado. Nossa abordagem será no sentido de agregar novos serviços para tolerância a faltas àqueles fornecidos por micronúcleos distribuídos, de forma a possibilitar a concepção de uma plataforma de desenvolvimento que apresente as características anteriormente discutidas.

Dentre os micronúcleos apresentados na literatura, o Amoeba [Mullender et al. 90], projetado e, na sua maior parte, implementado na Universidade Vrije, Amsterdã, destaca-se por sua penetração na comunidade científica e pela grande quantidade de informação disponível (inclusive código-fonte). Por estes motivos, o Amoeba foi escolhido como sistema hospedeiro para a primeira implementação do ambiente operacional Seljuk, a qual chamamos de *Seljuk-Amoeba.*

1.3. Estrutura da Dissertação

Esta dissertação está organizada da seguinte forma. No Capítulo 2 serão apresentados alguns conceitos básicos de tolerância a faltas, aplicados sobretudo a ambientes distribuídos, cujo conhecimento é imprescindível para o perfeito entendimento dos demais capítulos deste texto. Tais conceitos procuram seguir, sempre que possível, as definições apresentadas em [Laprie 89], que são adotadas a nível internacional, e a terminologia em português proposta por Lemos e Veríssimo [Lemos-Veríssimo 91] que mapeia aquela sugerida por Laprie. O capítulo termina identificando quatro serviços básicos para tolerância a faltas, quais sejam: replicação dos componentes de *software*, comunicação em grupo, diagnóstico de faltas e reconfiguração do sistema.

Nos Capítulos de 3 a 6, aspectos diversos relacionados a cada um dos serviços identificados no capítulo anterior, bem como possíveis mecanismos para implementá-los, serão explorados detalhadamente. A meta destes capítulos é comprovar a relevância de cada um destes serviços para a construção de sistemas tolerantes a faltas, bem como

formar uma sólida base de conhecimento que conduza à perfeita compreensão da proposta de implementação apresentada no Capítulo 7. Tais capítulos, juntamente com o Capítulo 2, podem também servir como fonte de pesquisa para pessoas que tenham interesse em obter informação organizada sobre conceitos básicos de tolerância a faltas e serviços para tolerância a faltas.

Especificamente, no Capítulo 3, as diversas técnicas de replicação de componentes de *software* serão estudadas, enfatizando os requisitos, vantagens e desvantagens de cada uma delas. A forma como o processamento do erro é executado por cada modelo de replicação será também discutida.

As técnicas de replicação apresentadas no Capítulo 3 possuem diferentes requisitos quanto a interação entre as réplicas, que podem ser satisfeitos por propriedades específicas do serviço de comunicação. O Capítulo 4 apresenta as diversas propriedades dos protocolos de comunicação em grupo. Algumas questões de desenho que diferenciam os sistemas operacionais existentes quanto ao suporte à comunicação em grupo serão também discutidas naquele capítulo, que conclui caracterizando os protocolos para comunicação em grupo de alguns dos sistemas distribuídos de maior destaque na literatura.

O Capítulo 5 destaca a importância da fase de diagnóstico no contexto da tolerância a faltas e apresenta alguns dos protocolos mais conhecidos para a realização de diagnóstico em sistemas distribuídos. O Capítulo 6 discute a necessidade de se reconfigurar o sistema após faltas terem sido diagnosticadas e apresenta algumas estratégias que podem ser utilizadas para a realização da tarefa de reconfiguração.

Uma proposta de implementação dos serviços para tolerância a faltas sobre o ambiente operacional distribuído Seljuk-Amoeba é apresentada no Capítulo 7. Tal proposta visa atender a requisitos de transparência e flexibilidade, e sua implementação será parte efetiva da construção da plataforma Seljuk-Amoeba - um ambiente para suporte ao desenvolvimento e à execução de aplicações distribuídas robustas. Finalmente, o Capítulo 8 traz nossas conclusões, com ênfase na relevância deste projeto de pesquisa, e discute possíveis direções para trabalhos futuros.

No Apêndice A, um glossário da terminologia proposta ao longo do texto é apresentado, relacionando o termo sugerido em português com o respectivo termo em inglês e o seu significado no contexto dos sistemas de computação.

Capítulo 2:

Conceitos de Tolerância a Faltas

2.1. Introdução

Sistemas de computador vêm sendo encarregados de responsabilidades cada vez maiores, que podem exigir alta confiabilidade e disponibilidade. Tais sistemas, porém, consistem de uma série de componentes de *hardware* e *software* que podem eventualmente falhar.

Em muitos casos, o comportamento errôneo do sistema ou a indisponibilidade do serviço são aceitáveis, desde que ocorram de forma temporária. Há, entretanto, um número crescente de aplicações para as quais o custo de falhas imprevisíveis, potencialmente perigosas, pode ser muito significante, trazendo conseqüências indesejáveis como:

- perda de dinheiro (p. ex., aplicações financeiras);
- perda de produção (p. ex., aplicações industriais);
- perda de clientes (p. ex., devido à perda de confiança no serviço);
- perda de confidencialidade (p. ex., sistemas com informações sigilosas); ou até
- perda de vida humana (p. ex., sistemas de monitoração de pacientes).

Para minimizar tais perdas, os projetistas destas aplicações fazem uso de mecanismos para tolerância a faltas, que propiciam a construção de sistemas nos quais os usuários podem confiar.

A meta deste capítulo é apresentar conceitos básicos da tolerância a faltas, sobretudo em ambientes distribuídos, cujo conhecimento é imprescindível para o perfeito entendimento dos demais capítulos deste texto. Tais conceitos, na sua maioria, seguem as definições apresentadas em [Laprie 89], que são adotadas a nível internacional, e a terminologia em português proposta em [Lemos-Veríssimo 91].

Iniciaremos o capítulo apresentando uma definição para *sistemas tolerantes a faltas*, que será usada ao longo deste trabalho. Em seguida, estenderemos o conceito de

tolerância a faltas para uso em sistemas distribuídos, uma vez que este será o nosso ambiente de desenvolvimento. Logo depois, apresentaremos o conceito de *confiança no funcionamento* e os seus diversos atributos. Por fim, discutiremos as diversas atividades envolvidas na obtenção de tolerância a faltas e alguns possíveis mecanismos para implementá-las. A partir destas atividades, identificaremos então alguns dos principais serviços para construção de aplicações tolerantes a faltas, que serão explorados profundamente nos quatro capítulos seguintes.

2.2. Sistemas Tolerantes a Faltas

Uma *falha* do sistema ocorre quando o seu comportamento não é consistente com suas especificações, ou seja, quando o serviço fornecido pelo sistema se desvia das condições estabelecidas na especificação do serviço. Uma falha ocorre porque o sistema estava incorreto: um *erro* é a parte do estado do sistema que está propensa a conduzir a uma falha subseqüente. A causa de um erro é uma *falta*. Portanto, um erro é uma manifestação de uma falta no sistema e uma falha é uma manifestação de um erro no serviço [Laprie 89].

Do ponto de vista estrutural, um sistema pode ser definido como um conjunto de subsistemas que interagem a fim de fornecerem um serviço [Lee-Anderson 90]. Cada um destes subsistemas, por sua vez, possui sua própria estrutura interna e um comportamento externo perceptível pelo resto do sistema. Sendo assim, um subsistema é por si só um sistema. Tal hierarquia sistema/subsistema continua até um nível em que não é mais possível ou desejável especificar os detalhes do sistema. Os subsistemas deste nível são então chamados de *componentes do sistema* ou *componentes atômicos*. Seguindo este raciocínio, uma falta em um sistema nada mais é do que a falha de um dos seus subsistemas.

É certo que quanto mais complexo é um componente, mais susceptível ele é a ocorrência de uma falha. Do mesmo modo, quanto maior o número de componentes de um sistema, maior é a possibilidade de algum deles apresentar uma falha. A evidente complexidade e o grande número de componentes dos sistemas de computação modernos, portanto, aumentam consideravelmente a probabilidade de uma falta ocorrer

no sistema, devido a falha de algum dos seus componentes, e, conseqüentemente, do sistema como um todo falhar.

Uma forma de melhorar a confiabilidade do sistema é utilizar a abordagem denominada *prevenção de faltas*, que tenta prevenir, por construção, a introdução ou a ocorrência de faltas no sistema [Laprie 89]. Porém, muito freqüentemente, o uso de técnicas de desenho e de fabricação sofisticadas na construção de componentes de *hardware* de alta qualidade não é suficiente para reduzir, a um nível aceitável, a probabilidade da ocorrência de falha destes componentes e, por conseguinte, de faltas no sistema. Neste caso, a confiabilidade do sistema pode ser melhorada através da utilização de técnicas de *tolerância a faltas*, que propiciam a construção de sistemas capazes de continuar a prover o serviço adequado apesar da ocorrência de faltas no sistema.

Um sistema é dito ser *tolerante a faltas* se ele apresenta um comportamento consistente com suas especificações apesar da ocorrência de falha de alguns de seus componentes. Isto significa que a falha de um componente do sistema não é refletida no comportamento externo deste sistema.

Redundância é a chave para tolerância a faltas e pode ser definida como a parte do sistema que não é necessária para o funcionamento do mesmo se nenhuma falta ocorre. Redundância pode ser introduzida no *hardware* (processadores, memória, canais de comunicação etc.), no *software* (p. ex., diversidade de projeto) ou no tempo (tempo extra para execução de tarefas ou reexecução das mesmas) [Avizienis 76].

A forma mais comum e mais abrangente de se tolerar faltas é a introdução de componentes de *hardware* redundantes que podem assumir a função daqueles que apresentarem falhas. A redundância de *software*, por sua vez, é empregada na tolerância a faltas de concepção de *software*, que são introduzidas durante a própria concepção ou manutenção do *software*. Finalmente, quando apenas faltas temporárias (aquelas que estão presentes no sistema apenas por um tempo limitado [Laprie 89]) são consideradas, pode ser suficiente utilizar computação redundante em componentes não-redundantes, onde a mesma operação é executada várias vezes seguidas, caracterizando assim redundância de tempo.

2.3. Computação Distribuída e Tolerância a Faltas

Um *sistema distribuído* pode ser definido como uma coleção de sistemas de computação autônomos, porém interconectados através de uma rede de comunicação. Os seus componentes são fisicamente separados e independentes, não compartilham memória nem relógio global e se comunicam apenas através de passagem de mensagens [Jalote 94, pp. 46].

Software distribuído é freqüentemente estruturado em termos de *clientes* e *serviços*. Cada serviço compreende um ou mais *servidores* que atendem a *pedidos* dos clientes. A maneira mais simples de se implementar um serviço é usar um servidor centralizado único que responde a todas as solicitações dos clientes; porém, o serviço resultante desta implementação é apenas tão confiável quanto o processador que executa aquele servidor e os canais de comunicação entre este e os seus clientes. Se este nível de tolerância a faltas é inaceitável, múltiplos servidores que falhem independentemente devem então ser usados [Schneider 90].

A idéia desta última abordagem é implementar o serviço por meio de um grupo de servidores redundantes, chamados de *réplicas*, que são executados em processadores distintos do sistema distribuído e que mantêm informação redundante sobre o estado global do serviço. Quando um membro do grupo falha, ou devido a falha de um componente de *hardware* ou por causa de uma falta de concepção no programa que implementa o servidor, as réplicas sobreviventes têm informação suficiente para continuarem a prover o serviço. Protocolos especiais precisam ser usados para coordenar as interações dos clientes com as réplicas do servidor correspondente.

A principal causa de falhas de componentes de *hardware* é a ocorrência de fenômenos físicos adversos, sejam eles externos (p. ex., radiação, alta temperatura etc.) ou internos ao componente (p. ex., desgaste natural, curto circuito etc.); tais falhas caracterizam as chamadas *faltas físicas* de um sistema. *Software*, por outro lado, não tem propriedades físicas e, portanto, faltas físicas não existem nesta área - falhas dos componentes de *software* caracterizam sempre *faltas de concepção*.

Quando apenas faltas físicas precisam ser toleradas, a simples replicação dos servidores da aplicação em processadores diferentes é suficiente para se alcançar a confiabilidade desejada. O isolamento físico e elétrico dos processadores, característico

dos sistemas distribuídos, garante que as falhas dos servidores replicados, decorrentes de faltas de *hardware*, ocorram de forma independente umas das outras, permitindo assim que o serviço adequado seja fornecido. Se, por outro lado, faltas de concepção do *hardware* devem ser consideradas, é ainda necessário que tais processadores tenham sido fabricados a partir de projetos distintos.

Com relação às faltas de concepção do *software*, a abordagem de replicação pode também ser empregada, embora com certa cautela. Neste caso, é necessário que se use ambientes operacionais distintos para cada réplica, apostando na suposição de que o simples fato de várias réplicas estarem sendo executadas em ambientes diversificados pode levar as faltas a se manifestarem independentemente nas diferentes réplicas. Isto porém não é confiável o suficiente para ser empregado com segurança.

Trabalhos em diversidade de projeto de *software* perseguem a meta de produzir uma metodologia que garanta que grupos de servidores, rodando programas implementados a partir de projetos diversos, não sofram falhas majoritárias apesar da possível existência de faltas de concepção nestes programas. Várias abordagens foram propostas para organizar projetos diversos na construção de *software tolerante a faltas* [Campbell et al. 79, Kim 84, Sullivan-Masson 90], sendo as mais conhecidas: a abordagem do bloco de recuperação [Randell 75] e a programação N-versões [Chen-Avizienis 78, Avizienis 85]. Talvez porque não haja técnicas aceitáveis para se estimar com segurança o aumento na confiabilidade resultante do uso de programação diversa, o trabalho em diversidade de projeto tenha gerado controvérsia até hoje [Cristian 91].

A arquitetura sobre a qual este trabalho se baseia [Brasileiro 97] permite a construção de sistemas capazes de tolerar faltas físicas e faltas de concepção, tanto do *hardware* quanto do *software*. Porém, no caso de faltas de concepção, considera-se que os componentes do sistema possuem projetos e implementações diferentes. Uma visão geral de diversas técnicas para tolerância a faltas de concepção do *software* pode ser encontrada em [Jalote 94, pp. 355-399], bem como as referências para os textos originais que apresentam tais técnicas e suas implementações.

2.4. Confiança no Funcionamento

Confiança no funcionamento pode ser definida como a propriedade de um sistema de computador tal que confiança possa ser justificavelmente depositada no serviço que ele entrega [Laprie 89]. Confiança no funcionamento é, portanto, um conceito global que inclui os atributos usuais de *confiabilidade* (continuidade do serviço), *disponibilidade* (presteza para uso), *segurança contra faltas acidentais* (prevenção de catástrofes) e *segurança contra faltas intencionais* (prevenção de acesso não autorizado ao sistema ou a suas informações). Segurança contra faltas acidentais e segurança contra faltas intencionais são os termos adotados por [Lemos-Veríssimo 91] como tradução para os termos em inglês *safety* e *security*, respectivamente.

Em termos de tolerância a faltas, os atributos mais significantes são confiabilidade e disponibilidade. Confiabilidade pode ser definida como a *probabilidade do sistema continuar funcionando corretamente durante todo o tempo de sua missão.* Disponibilidade, por sua vez, é a *probabilidade do sistema estar operante em qualquer instante do tempo.* Por exemplo, um sistema de controle de uma usina nuclear deve apresentar alta confiabilidade, enquanto um sistema de caixa eletrônico de uma rede bancária deve apresentar alta disponibilidade.

A meta do desenho tolerante a faltas é aumentar a confiança no funcionamento por permitir o sistema realizar a função pretendida apesar da presença de um dado número de falhas de seus componentes (isto é, de faltas no sistema).

Se operação confiável é necessária (ou desejável), o sistema distribuído deve possuir atributos arquiteturais específicos. Neste sentido, considerações importantes são [Powell 92]:

- o número tipicamente grande de recursos de *hardware* empregados em sistemas de processamento de informações de larga escala naturalmente aumenta a probabilidade de falha, pelo menos parcial, do sistema; e
- *hardware* de prateleira (*off-the-shelf*) pode falhar de uma maneira arbitrária e não possui mecanismos especiais para detectar erros, com probabilidade suficientemente alta, ou para conter os seus efeitos de modo que eles não se propaguem subseqüentemente pelo sistema de uma maneira incontrolada e imprevisível.

O requisito para operação confiável do sistema pode ser satisfeito pelo uso de redundância, através da replicação de componentes de *software* em unidades de processamento[1] distintas do sistema distribuído, conforme descrito na seção anterior. Isto requer suporte à replicação de componentes lógicos, como, por exemplo, disseminação coerente de informação e, em certos casos, algum esquema de votação majoritária ou comparação das saídas replicadas. Além disso, mecanismos avançados de administração de sistema são requeridos para implementar diagnóstico de faltas e reconfiguração do sistema. Finalmente, atividades de validação extensivas precisam ser executadas para justificar a confiança nos serviços fornecidos pelo sistema.

2.5. Fases da Tolerância a Faltas

Tolerância a faltas é realizada ao longo de várias fases relacionadas tanto ao processamento do erro quanto ao tratamento da falta. Enquanto o processamento do erro tem por objetivo a remoção de erros do estado computacional, se possível antes da ocorrência de uma falha no serviço entregue pelo sistema, o tratamento da falta tenta prevenir que as faltas que geraram tais erros venham a ser ativadas novamente [Lee-Anderson 90]. Geralmente, não há uma ordem fixa de execução das ações relacionadas a cada uma destas fases e nem sempre é possível associar um mecanismo particular com cada uma delas. Além disso, em certas situações, algumas fases podem ser executadas implicitamente. Entretanto, no âmbito deste trabalho, é importante saber da existência destas fases e das ações a elas associadas.

Fases do Processamento do Erro

- Detecção/Compensação/Correção. Detecção do erro e sua correção dinâmica.
- Confinamento. Prevenção da propagação do erro além dos limites definidos.
- Recuperação. Restauração do sistema para um estado correto.

Fases do Tratamento da Falta

- Diagnóstico. Identificação do componente responsável pelo erro detectado.
- Apassivação. Prevenção da reativação das faltas diagnosticadas.
- Reconfiguração. Restauração do nível de redundância do sistema.

[1] O termo unidade de processamento refere-se tanto a um processador quanto a um conjunto deles (nodo).

2.5.1. Detecção, Compensação e Correção do Erro

Erros em um sistema computacional podem ocorrer tanto no *domínio do tempo* (aqueles que conduzem ao atraso ou ausência de respostas/mensagens) quanto no *domínio do valor* (aqueles que conduzem à produção de mensagens com conteúdo incorreto).

Detectar erros ocorrendo em componentes de armazenagem de dados, tais como memória, registradores e discos, ou erros gerados durante a transmissão dos dados através de canais de comunicação é muito mais fácil do que detectar erros originados em componentes que geram ou transformam dados. A compensação ou a correção de erros, por sua vez, é mais difícil do que sua detecção e requer o emprego de algum tipo de redundância de modo que dados corretos possam ser extraídos da informação redundante. Mecanismos para tolerância a faltas podem ser implementados tanto em *hardware* quanto em *software*. A seguir veremos como a detecção e correção de erros no domínio do tempo e no domínio do valor podem ser conseguidas usando ambas as abordagens: em *hardware* e em *software*.

<u>Mecanismos para Tolerância a Faltas em *Hardware*</u>

Com respeito à tolerância a faltas implementada em *hardware*, alguns mecanismos já vêm embutidos nos próprios sistemas básicos de processamento e comunicação. É o caso da teoria da codificação, que provê os métodos mais comuns para detecção e correção de erros no domínio do valor em sistemas digitais. A idéia básica em torno da codificação é adicionar *bits* de checagem à informação tal que os erros em alguns *bits* da informação possam ser detectados e possivelmente corrigidos [Jalote 94, pp. 22-25].

Códigos de detecção e correção de erros variam muito em termos de: propriedades de detecção e correção, complexidade da codificação/decodificação, grau de cobertura e eficiência do código. A maioria dos códigos incluem simples checagem de paridade para detectar erros em barramentos, memória e registradores. *Códigos Hamming* [Jalote 94, pp. 23-24], baseados em paridade, detectam e corrigem erros na memória; *códigos de redundância cíclica* (CRC) detectam e corrigem erros em canais de comunicação e discos; *códigos 'm-out-of-n'* detectam erros em armazenagens de controle de microprograma e outras ROMs; e *códigos aritméticos* detectam erros originados dentro de unidades lógicas e aritméticas [Nelson 90].

É fácil observar que as técnicas de tolerância a faltas baseadas em codificação apenas detectam/corrigem informações danificadas durante seu tempo de armazenagem ou durante sua transmissão pelos canais de comunicação. Com componentes que geram ou transformam informação, a replicação do módulo completo, com as saídas dos módulos idênticos sendo comparadas, é freqüentemente a única abordagem efetiva para detecção e compensação de erros.

Alguns sistemas comerciais têm sido construídos com pares de microprocessadores de prateleira, dotados de circuitos comparadores em suas interfaces com o barramento que detectam falhas dos processadores. Operação continuada, por sua vez, é freqüentemente conseguida pelo uso do voto majoritário das saídas de três ou mais módulos idênticos, mascarando falhas de uma minoria dos módulos. A forma mais comum desta técnica de tolerância a faltas é a conhecida *redundância modular tripla* (TMR) [Jalote 94, pp. 20-21], na qual o processador é triplicado e todas os três processadores trabalham em paralelo, tendo suas saídas submetidas a um elemento *votador* que entrega o voto majoritário como saída final. Neste caso, a falha de um entre os três processadores é completamente mascarada.

Votadores e comparadores, embora sejam tipicamente muito mais confiáveis do que os módulos redundantes que eles protegem, representam pontos potenciais de falha. O grau de tolerância a faltas pode, então, ser aumentado através da replicação dos próprios comparadores e votadores.

Quanto a detecção de erros no domínio do tempo, um mecanismo bastante simples e geralmente usado é a checagem de tempo - um evento que deixa de ocorrer dentro de um intervalo de tempo predefinido usualmente indica a ocorrência de uma falha e, portanto, a existência de um erro. Tais ocorrências podem ser monitoradas por um temporizador disparado no início de cada evento e instruído para parar após algum tempo T, interromper o sistema e sinalizar uma falha, que é a conseqüência de um erro. Se o evento se completa antes de T ter se esgotado, o temporizador pára e reinicializa para o próximo evento [Nelson 90].

Mecanismos para Tolerância a Faltas em *Software*

Mecanismos para tolerância a faltas implementados em *software* baseiam-se, em geral, na replicação da computação em unidades de processamento distintas do sistema

distribuído e são utilizados principalmente para detectar/mascarar erros no sistema decorrentes de componentes que geram um valor incorreto ou simplesmente deixam de gerar o valor esperado. Como estamos particularmente interessados em tolerar, via *software*, faltas que conduzam a este tipo de erro, tais mecanismos serão alvo de exploração ao longo de todo este trabalho. No Capítulo 3, as diversas técnicas de replicação de componentes de *software* serão estudadas detalhadamente. Por enquanto, daremos apenas uma visão geral de estratégias para tolerância a faltas implementadas em *software*.

Como na implementação em *hardware*, a detecção de erros no domínio do tempo é normalmente feita através do uso de temporizadores. Porém, embora a expiração de um temporizador seja de fato um erro, isto não necessariamente significa que o componente que deveria gerar o valor esperado falhou e deve ser irrevogavelmente removido do sistema, visto que o problema pode derivar, por exemplo, de uma sobrecarga de processamento do próprio componente ou mesmo de um congestionamento na rede de comunicação; além do mais, grande parte dos sistemas distribuídos são assíncronos de modo que não se pode definir um limite de tempo finito dentro do qual a função pretendida seria de fato realizada caso nenhuma falta ocorresse no sistema [Mishra-Schlicthig 92]. Protocolos de processamento de erro apropriadamente projetados devem mascarar tais erros, mas registrá-los para o sistema de administração, que, por sua vez, poderá primeiro tentar aliviar aquele componente, balanceando a carga, e apenas o removerá do sistema se for diagnosticado que o número de erros registrados excede um determinado limite.

Erros no domínio do valor, por outro lado, são detectados comparando-se as mensagens de saída equivalentes das diferentes réplicas de um componente de *software*. Para isto ser possível, é necessário que as réplicas localizadas nas unidades de processamento corretas permaneçam consistentes de modo a produzirem as mesmas mensagens de saída. É também necessário que haja alguma maneira de identificar mensagens equivalentes, ou seja, mensagens que deveriam conduzir o mesmo valor. Finalmente, um esquema de votação majoritária [Jalote 94, pp. 274 e 295-298] deve ser adotado para se escolher o valor correto entre todos os valores gerados pelas diferentes réplicas.

A correção de erro sem redundância é difícil. Entretanto, para a maioria das faltas temporárias, desde que seja possível restaurar o estado do sistema para aquele do início da operação, a simples repetição da operação (redundância no tempo) depois da falta desaparecer produzirá resultados corretos, visto que a causa do erro não mais existe no sistema.

2.5.2. Confinamento e Avaliação do dano

A fim de proteger recursos críticos do sistema e minimizar o tempo de recuperação, os erros devem ser confinados aos módulos onde eles foram gerados. Os limites do confinamento podem ser estabelecidos de duas maneiras: (1) cada módulo pode checar suas próprias saídas; ou (2) cada módulo pode checar todas as informações de entrada. A abordagem mais comum é requerer que cada módulo suspeite de toda informação de entrada e corrija ou bloqueie o dado incorreto na interface do módulo.

Nesta configuração, toda informação de entrada para um módulo passa primeiro por uma unidade de detecção/correção de erros (comparador/votador/autenticador de assinaturas) que filtra os valores de entrada, deixando passar para o módulo apenas informações corretas. Tal unidade pode ser implementada em *hardware* ou em *software*.

Na outra abordagem, onde o módulo é responsável por checar suas próprias saídas, um circuito de detecção/correção de erros (comparador/votador/verificador de código) é colocado na interface entre o módulo e o canal de comunicação, juntamente com um circuito capaz de desabilitar a saída do módulo. Se a correção do erro não é possível, o módulo falho é isolado para prevenir a propagação do erro - seu processamento é efetivamente parado. A desvantagem desta configuração é que a interface do módulo freqüentemente não pode proteger o sistema de falhas dos próprios circuitos da interface.

Como nem sempre é possível dispor de recursos especiais para limitar os efeitos de uma falta (isto é, para confinar o dano) uma outra abordagem, direcionada para a avaliação do dano causado por uma falta, pode ser usada. Esta estratégia baseia-se na assertiva de que erros se propagam no sistema através da comunicação entre os seus componentes e, portanto, para se avaliar a quantidade de dano no sistema depois do erro ter sido detectado, o fluxo de informação entre os diferentes componentes deve ser examinado.

Freqüentemente, a atividade de avaliação do dano não é realizada explicitamente: a estrutura do sistema é usada na fase de recuperação do erro para decidir a quantidade de recuperação necessária, indiretamente fazendo suposições sobre o confinamento do dano. A fim de minimizar os danos causados por uma falta, o sistema deve ser estruturado de uma maneira tal que o fluxo de informação entre seus componentes seja controlado e bem definido [Brasileiro 95].

2.5.3. Recuperação do Sistema

Os mecanismos para correção de erros originados em módulos que geram ou transformam dados requerem redundância do módulo e outros recursos especiais (comparadores, votadores etc.) nem sempre disponíveis (em geral, por questões de custo). Os mecanismos de detecção de erro e confinamento do dano por si só não corrigem o estado do sistema. Portanto, uma vez que um erro foi detectado, mas não corrigido, e a sua extensão foi identificada, é necessário transformar o estado errôneo do sistema em um estado válido. Esta tarefa é referenciada como *recuperação de erro.*

A maioria dos esquemas de recuperação restauram a operação do sistema a um estado anterior livre de erro, denominado *ponto de salvaguarda* (*checkpoint*). Um processador é retrocedido (*rolled back*) para um ponto de salvaguarda por meio da restauração dos seus registradores e da memória ao estado salvo anteriormente e da invalidação da memória *cache*, forçando os dados desta a serem restaurados a partir da memória global. Dados globais são tipicamente protegidos através de protocolos que permitem que as atualizações sejam completadas com sucesso ou, em caso de falha, sejam desfeitas e repetidas.

Em sistemas distribuídos, tolerância a faltas muitas vezes é conseguida através da manutenção de processadores extras, denominados de *suplentes* (*spares*), que podem substituir um processador *mestre* quando este falha. Os processadores suplentes são periodicamente atualizados em pontos de salvaguarda pré-definidos, de modo que, quando um processador mestre falha e o controle de uma tarefa é dado a um processador suplente, o processamento pode continuar a partir do ponto de salvaguarda mais recente, eliminando a necessidade de executar toda a tarefa novamente. O grau de retrocesso (*rollback*) é limitado pelo uso de ações atômicas - pequenos e indivisíveis passos de processamento, completados e verificados antes das atualizações globais serem

feitas e da próxima ação ser iniciada. A recuperação de um erro que ocorre antes dos resultados da ação serem salvos é usualmente realizada pela repetição da ação inteira.

A estratégia descrita acima é denominada de *recuperação de erro para trás* e pode ser empregada independente da natureza da falha. Sua principal desvantagem é a carga extra de processamento e comunicação incorrida na operação de *salvaguarda (checkpointing).*

Uma outra técnica de recuperação de erro, denominada *recuperação de erro para a frente*, não requer conhecimento de nenhum estado prévio nem realiza retrocesso do sistema; ao contrário, tal estratégia tenta tornar o estado livre de erros através da realização de algumas ações corretivas. Esta técnica é menos geral que a anterior, pois requer que a natureza exata do erro seja conhecida para que as ações corretas sejam tomadas. Isto torna este tipo de recuperação dependente do sistema e da aplicação e, por isso, esta estratégia não é tão comumente usada como a anterior [Jalote 94, pp. 15].

As denominações "recuperação de erro para trás" e "recuperação de erro para a frente" são as respectivas traduções sugeridas por [Lemos-Veríssimo 91] para os termos em inglês "*backward error recovery*" e "*forward error recovery*".

2.5.4. Tratamento da Falta

Como vimos, tolerância a faltas se dá basicamente em duas fases gerais: processamento do erro e tratamento da falta. Enquanto processamento de erro ajuda a evitar que erros se tornem visíveis para os usuários, através da manifestação de uma falha, tratamento da falta tenta prevenir que as faltas que geraram tais erros voltem a se manifestar.

Se o erro é causado por uma falta temporária, a simples reinicialização do sistema a partir de um estado válido eliminaria completamente o erro do sistema, visto que a falta não mais existiria. Porém, se a falta é permanente, o erro reincidiria, já que a falta que o gerou permanece no sistema. A fim de evitar isto, o componente apresentando a falta deve ser identificado e não mais utilizado na computação subseqüente à recuperação do erro.

O tratamento da falta, portanto, envolve diagnóstico e apassivação da falta e, idealmente, reconfiguração do sistema [Powell 92]. O diagnóstico tem por objetivo

identificar faltas no sistema, isto é, os componentes do sistema que apresentam falhas. A apassivação procura prevenir que as faltas diagnosticadas sejam ativadas novamente através da eliminação ou reconfiguração do componente falho. Finalmente, a reconfiguração do sistema é necessária para restaurar o nível de redundância de modo que o sistema seja hábil a tolerar faltas subseqüentes.

Em ambientes distribuídos tolerantes a faltas, o tratamento da falta é uma tarefa crítica, devendo portanto ser tão automatizada quanto possível.

O diagnóstico é imprescindível tanto para a apassivação da falta quanto para a reconfiguração do sistema, pois, a menos que o componente falho seja identificado, nenhum mecanismo pode ser empregado nem para impedir que a falta representada pela falha do componente gere um erro novamente, nem para reparar a falta através da reconfiguração do sistema.

Após o diagnóstico, os componentes falhos devem ser removidos do sistema ou então usados numa configuração diferente a fim de não mais conduzirem a falhas do sistema - isto é feito na fase de apassivação da falta. Porém, a exclusão destes componentes, quando ocorre, vai pouco a pouco degradando o grau de redundância do sistema e, por conseguinte, os serviços para tolerância a faltas usados pela aplicação. Surge, então, a necessidade de reconfigurar o sistema, o que se dá com a inclusão de novos componentes ao sistema. Tais componentes podem ser módulos suplentes ou os próprios módulos que falharam, desde que eles já tenham sido recuperados.

As atividades relativas à apassivação da falta são intimamente ligadas à reconfiguração do sistema, estando geralmente embutidas nesta última fase. Por este motivo, é comum tratá-las como atividades de reconfiguração, não fazendo distinção explícita entre as duas fases. Por exemplo, unidades de processamento com semântica de *falha silenciosa*[2] [Powell et al. 88, Shrivastava et al. 91b, Shrivastava et al. 92, Brasileiro 95, Brasileiro et al. 96], por definição, executam a apassivação da falta automaticamente e autonomamente, visto que a unidade simplesmente pára de funcionar na presença da falta.

[2] Unidades de processamento com semântica de falha silenciosa são aquelas que ou entregam o serviço especificado ou não entregam serviço algum. Ou seja, quando uma falha ocorre, tais unidades simplesmente param de produzir saídas. Para maiores detalhes, vide Seção 3.2.

Após o diagnóstico, os componentes apresentando a falta devem ser removidos do sistema. Porém, a exclusão destes componentes vai paulatinamente degradando o nível de redundância do sistema e, conseqüentemente, a capacidade de se tolerar faltas posteriores. Surge, então, a necessidade de reconfiguração do sistema, que se dá com a inclusão de novas réplicas no grupo responsável pelo fornecimento do serviço.

A reconfiguração do sistema somente pode ser considerada se há recursos redundantes suficientes, uma vez que esta tarefa acarreta a realocação e reinicialidação das réplicas que falharam a fim de restaurar o nível de redundância necessário para os protocolos de processamento de erro continuarem funcionando corretamente diante de faltas posteriores.

Com base nas considerações acima, podemos identificar quatro serviços básicos para a construção de aplicações tolerantes a faltas:

- Replicação dos componentes de software
- Comunicação em grupo
- Diagnóstico de faltas e
- Reconfiguração do sistema

Nos capítulos seguintes, os diversos aspectos relacionados a cada um destes serviços, bem como possíveis formas de implementá-los, serão explorados detalhadamente. Nossa meta é comprovar a relevância de cada um destes serviços, bem como formar uma base sólida de conhecimento para o perfeito entendimento da proposta de implementação apresentada no Capítulo 7 desta dissertação.

Capítulo 3:

Replicação de Componentes de *Software*

3.1. Introdução

A replicação de dados e computação em diferentes unidades de processamento é o único meio pelo qual um sistema distribuído pode seguramente continuar a fornecer o serviço adequado mesmo na presença de falhas de alguns dos seus componentes. Embora a *armazenagem estável* [Lampson 81; Jalote 94, pp. 99-107] possa ser usado para permitir ao sistema recuperar-se de falhas no serviço de processamento, tal técnica usada isoladamente não permite ao sistema distribuído alcançar melhor confiabilidade/ disponibilidade do que aquele centralizado. Na verdade, se uma computação é distribuída sobre múltiplas unidades de processamento sem qualquer forma de replicação, a distribuição pode até mesmo conduzir ao decréscimo da confiabilidade, visto que, na maioria dos casos, a computação só pode proceder se todos as unidades envolvidas estiverem operacionais.

Um *componente de software replicado* é definido como um componente de *software* que possui uma representação ou *réplica* em duas ou mais unidades de processamento do sistema. Daqui por diante, o termo *componente de software* poderá ser usado para referenciar tanto um componente individual quanto a entidade lógica como um todo, isto é, o *grupo de réplicas*, o que ficará subentendido no contexto.

Técnicas de tolerância a faltas distribuídas baseadas na replicação de componentes de *software* apresentam muitas vantagens [Chérèque et al. 92, Powell 92]:

- O custo de desenho de *hardware* é minimizado: uma vez que tolerância a faltas distribuída é implementada principalmente em *software*, o *hardware* especializado requerido é o mínimo possível.
- Separação geográfica de recursos não precisa ser adicionada: as mesmas técnicas de tolerância a faltas podem ser usadas independente de as réplicas estarem perto ou longe umas das outras.

- Certas faltas de concepção de *software* podem ser toleradas sem recorrer a um projeto diferente já que leves diferenças no ambiente local das réplicas podem levar tais faltas a se manifestarem independentemente nas diferentes réplicas.
- A sincronização fraca das réplicas (através de passagem de mensagens) conduz ao melhoramento da tolerância a faltas temporárias que poderiam afetar simultaneamente várias réplicas no mesmo ponto da execução.

O grau de replicação dos componentes de *software* depende principalmente de quão crítico é o componente, mas depende também de quão fácil e rápido é adicionar novos membros ao grupo existente (para substituir réplicas falhas). Em geral, é desejável figurar grupos de tamanhos variados, embora o grau de replicação possa freqüentemente ser limitado a 2 ou 3 (ou até mesmo 1, isto é, nenhuma replicação, para componentes não-críticos) [Powell 92].

Há duas questões principais relacionadas à replicação de componentes de *software* para prover tolerância a faltas [Chérèque et al. 92]:

(i) Gerência da afiliação do grupo (*group membership*): como um componente de *software* é instanciado como um grupo de réplicas e como a afiliação do grupo é atualizada como conseqüência de falhas e reparos?

(ii) Coordenação inter-réplicas: como a atividade do grupo de réplicas é coordenada a fim de processar erros e dar a ilusão para outros componentes do sistema de um componente de *software* único, livre de falha?

Estas questões e diversos outros aspectos relacionados à replicação de componentes de *software* serão tratadas ao longo deste capítulo, que dá destaque especial às características dos diferentes modelos de replicação: ativa, passiva e semi-ativa.

3.2. Semântica de Falha do Serviço de Processamento e das Réplicas

Várias suposições sobre a semântica de falha do serviço de processamento são possíveis. Em um extremo, representando a semântica menos restritiva, está a *semântica de falha arbitrária* e no outro, representando a semântica mais restritiva, está a *semântica de falha silenciosa.* Suposições intermediárias podem ser introduzidas

dependendo principalmente das propriedades do sistema de comunicação, como por exemplo *falhas por omissão* e *falhas por temporização* [Cristian et al. 86].

Uma falha por omissão é aquela que leva um componente a não responder a algumas mensagens; elas são muitas vezes resultantes de perda de mensagens nos canais de comunicação. Já uma falha por temporização é aquela que leva um componente a responder muito cedo ou muito tarde a certas solicitações; este tipo de falha é algumas vezes chamado de *falha de desempenho.*

Semântica de falha silenciosa é a suposição simplista de que as unidades de processamento quando falham simplesmente param de enviar mensagens[4]. Como tais unidades só enviam mensagens corretas, a detecção de erro pode ser conseguida de forma simples e direta: as unidades de processamento trocam mensagens da forma "*I'm alive*" e uma unidade inoperante é detectada por sua falha em enviar tal mensagem dentro de um determinado intervalo de tempo. Porém, tal detecção só pode ser considerada verdadeiramente correta quando o sistema em questão é síncrono, uma vez que em sistemas assíncronos não há como diferenciar a situação em que o sistema realmente falhou daquela em que ele apenas está muito lento. Para sistemas assíncronos, o máximo que se pode fazer é *supor* a ocorrência da falha [Mishra-Schlicthig 92].

Já a semântica de falha arbitrária é a suposição mais ampla possível, em que as unidades de processamento não possuem qualquer mecanismo local de detecção de erro, podendo assim produzir comportamento arbitrário, inclusive malicioso. Na literatura, as faltas que produzem tais falhas são também chamadas de faltas Bizantinas [Lamport 82; Cristian et al. 85; Jalote 94, pp. 52]. Uma unidade de processamento deste tipo pode: deixar de enviar (algumas) mensagens, atrasar ou adiantar o envio de mensagens, enviar mensagens extras, enviar mensagens com conteúdo errôneo, enviar mensagens com conteúdo diferente para diferentes destinos ou recusar o recebimento de mensagens. Ainda, uma unidade de processamento com semântica de falha Bizantina pode tentar "enganar" o sistema, fazendo-se passar por uma outra unidade. Há, porém, uma variação deste tipo de semântica de falha, denominada de *Bizantina autenticada* [Strong et al. 90], onde a utilização de identidade falsa não é exeqüível.

[4] Neste trabalho, a exemplo da grande maioria das publicações na área, assumimos que os componentes de um sistema distribuído comunicam-se apenas através da passagem de mensagens.

Visto que um desenho baseado em suposições sobre o comportamento de falha de componentes corre o risco de falhar se estas suposições não forem satisfeitas e como nem sempre é possível garantir um comportamento bem definido em caso de falha, é prudente que sistemas críticos sejam projetados para tratar falhas Bizantinas [Schneider 90].

Se apenas faltas físicas (i.e., faltas de *hardware*) são consideradas, pode-se assumir que as réplicas de um componente de *software* falham da mesma maneira definida pelo comportamento de falha do serviço de processamento sobre o qual elas executam. Se uma unidade de processamento com semântica de falha silenciosa apresentar uma falha, então todas as réplicas executando naquela unidade pareceriam ter falhado silenciosamente. Se, por outro lado, ocorrer uma falha em uma unidade de processamento com semântica de falha arbitrária, então algumas ou todas as réplicas que estão sendo ali executadas poderão falhar arbitrariamente [Chérèque et al. 92].

Falhas do próprio componente de *software* são derivadas de faltas de concepção, que normalmente são toleradas pelo uso de diversidade de projeto. Porém, abstraindo-se o fato de as réplicas de um componente de *software* precisarem possuir projetos e implementações diferentes se faltas de concepção de *software* devem ser toleradas, as mesmas técnicas de processamento de erro usadas para tolerar faltas físicas são perfeitamente aplicáveis na tolerância de faltas dos componentes de *software*. Lembremos, porém, que não é objetivo deste trabalho discutir questões relacionadas à diversidade de projeto. Portanto, quando faltas de concepção de um componente de *software* são aqui tratadas, estamos assumindo que as réplicas do componente são implementadas a partir de projetos diferentes. Do mesmo modo, para faltas de concepção do *hardware*, assumimos que as réplicas são executadas em componentes de *hardware* derivados de diferentes projetos.

Por questões de generalidade, ao longo deste trabalho, nós nos referiremos a *falha de uma réplica* quando na verdade estaremos tratando da falha do conjunto *hardware-software* (ou seja, unidade de processamento-réplica), uma vez que, sob o ponto de vista do usuário, a falha da unidade de processamento na qual uma réplica está sendo executada se apresenta como uma falha da própria réplica. A mesma observação é válida em se tratando da *semântica de falha da réplica*. Note que o conjunto *hardware-software* apresenta semântica de falha silenciosa apenas quando ambos os componentes

(unidade de processamento e réplica) falham silenciosamente. Caso contrário, se algum dos componentes (ou ambos) apresenta semântica de falha arbitrária, esta mesma semântica de falha é assumida para o conjunto como um todo.

3.3. Determinismo de Réplicas e do Grupo de Réplicas

[Chérèque et al. 92] define determinismo de réplicas e do grupo de réplicas da seguinte forma:

Determinismo de Réplicas. Uma réplica é dita ser determinística se, na ausência de faltas, qualquer execução começando do mesmo estado inicial e consumindo o mesmo conjunto ordenado de mensagens de entrada produz o mesmo conjunto ordenado de mensagens de saída.

Determinismo do Grupo de Réplicas. Um grupo de réplicas é determinístico se, na ausência de faltas, dado o mesmo estado inicial e o mesmo conjunto ordenado de mensagens de entrada para todas as réplicas, cada réplica do grupo produz o mesmo conjunto ordenado de mensagens de saída.

Se todas as réplicas do grupo consomem mensagens idênticas e na mesma ordem, então o determinismo de cada réplica é condição suficiente para se alcançar o determinismo do grupo de réplicas.

O determinismo de réplicas pode ser complicado pela heterogeneidade. Por exemplo, se um componente de *software* é compilado com diferentes compiladores, as réplicas resultantes podem não se comportar consistentemente. Para se garantir determinismo em um ambiente verdadeiramente heterogêneo, as localizações das réplicas devem ficar restritas a um subconjunto de unidades de processamento que, se livres de falha, garantam que a consistência de processamento é mantida. Tal subconjunto é denominado *domínio de replicação do componente de software* [Powell 92] (vide Seção 3.4).

Se as réplicas não são determinísticas, o determinismo do grupo só pode ser conseguido pela negociação entre as réplicas [Barrett et al. 90, Tully-Shrivastava 90]. Alternativamente, a potencialidade para o não-determinismo pode ser removida adotando-se um modelo restritivo de computação baseado em *máquinas de estado* [Scheneider 90].

Se o determinismo da réplica não pode ser garantido de forma alguma, então é necessário impor suposições fortes sobre o seu modo de falha e restringir a escolha das possíveis estratégias de processamento de erro.

3.4. Domínio de Replicação

O *domínio de replicação* de um componente de *software* é definido como o conjunto de unidades de processamento em que as réplicas daquele componente podem residir [Powell 92].

Obviamente, as réplicas de um dado componente de *software* só podem ser alocadas em unidades de processamento que possuam os recursos necessários para sua execução. Entretanto, há geralmente outras razões para se restringir o domínio de replicação do componente e talvez a principal delas seja a necessidade de se garantir determinismo da réplica, como mencionado na seção anterior.

Um outro fator que pode afetar a definição de um domínio de replicação é a semântica de falha assumida pela técnica de replicação selecionada. Por exemplo, se as réplicas devem apresentar semântica de falha silenciosa com um alto grau de confiança, então suas localizações seriam restritas a unidades de processamento implementadas a partir de *hardware* autochecável ou de um conjunto de processadores cuja atividade garanta a semântica requerida (conceito de *nodo* - vide Seção 1.1.2).

Finalmente, equivalência de velocidade de execução é ainda um outro critério na definição de um domínio de replicação. Embora réplicas possam produzir saídas idênticas quando executadas em um dado conjunto de unidades de processamento, uma dispersão na velocidade de execução, acima de um determinado nível, poderia complicar a detecção de erros no domínio do tempo. Além disso, a necessidade de sincronização inter-réplicas forçaria todas as réplicas a procederem, em média, à velocidade da réplica mais lenta.

3.5. Estratégias de Processamento de Erro

No contexto da replicação de componentes de *software*, o processamento de erro consiste das técnicas utilizadas para coordenar a computação replicada, que permitem a comunicação e a computação procederem apesar da falha de algumas réplicas.

Três modelos básicos de computação replicada estão disponíveis:

a) **Modelo de Réplicas Ativas**. Neste modelo, todas as réplicas processam concorrentemente todas as mensagens de entrada de modo que seus estados são sincronizados e, na ausência de faltas, todas elas produzem as mesmas mensagens de saída e na mesma ordem. Isto requer que as réplicas apresentem comportamento determinístico na ausência de faltas. A técnica de replicação ativa pode ser usada para tolerar faltas decorrentes de falhas no serviço de processamento ou no próprio componente de *software* sob suposições tanto de falha silenciosa quanto de falha arbitrária. A fim de tolerar falhas arbitrárias, as saídas de todas as réplicas são comparadas e a decisão majoritária é usada.

Se, por outro lado, as réplicas possuem semântica de falha silenciosa, a replicação ativa pode ser usada sem votação, uma vez que qualquer mensagem produzida por uma réplica deste tipo pode ser seguramente assumida como correta. Como resultado, os requisitos do sistema de comunicação são simplificados e melhor desempenho é alcançado, visto que os resultados podem ser propagados imediatamente após terem sido gerados, em vez de ficarem pendentes esperando o processo de votação.

b) **Modelo de Réplicas Passivas**. Nesta abordagem, uma das réplicas (a *réplica primária*) processa as mensagens de entrada e provê mensagens de saída - na ausência de faltas; os estados internos das demais réplicas (as *réplicas suplentes*) são regularmente atualizados por meio de salvaguardas (*checkpoints*) da réplica primária. Se a réplica primária falha, uma suplente é ativada e começa a executar a partir de seu ponto de salvaguarda mais recente. Para muitas aplicações, a principal vantagem desta técnica é que ela não requer que as réplicas sejam determinísticas. Além do mais, os requisitos de processamento são minimizados: capturar salvaguardas geralmente requer menos recursos do que execução replicada.

No modelo de replicação ativa, a falha de uma réplica é detectada quando uma mensagem por ela produzida não combina com aquelas geradas pela maioria das réplicas

do seu grupo ou simplesmente não é produzida. A replicação passiva, por sua vez, não possui mecanismo implícito para detecção da falha da réplica primária. Por este motivo, a abordagem de replicação passiva supõe que o componente de *software* replicado possui semântica de falha silenciosa.

c) **Modelo de Réplicas Semi-ativas**. Esta técnica pode ser vista como um híbrido das duas técnicas anteriores: todas as réplicas processam todas as mensagens de entrada, mas apenas uma delas (a *réplica líder*) provê mensagens de saída. As decisões que afetam o determinismo são tomadas pela líder e comunicadas às demais réplicas (as *seguidoras*). A suposição de falha silenciosa, assumida por este modelo, garante que as mensagens propagadas pela líder não são errôneas, dispensando assim qualquer validação.

As próximas seções tratam de cada uma destas técnicas detalhadamente.

3.6. Replicação Ativa

O modelo de *replicação ativa* envolve a operação paralela de um certo número de cópias idênticas do componente de *software*, com as saídas de todas elas sendo comparadas e a decisão majoritária sendo usada. Porém, se é garantido que apenas mensagens corretas são produzidas, isto é, que as réplicas apresentam semântica de falha silenciosa, a saída de qualquer uma delas pode ser usada, não havendo necessidade de qualquer comparação.

Com esta técnica, a recuperação quase instantânea dos erros detectados pode ser conseguida desde que seja possível garantir que todas as réplicas livres de falha produzam as mesmas mensagens de saída na mesma ordem - condição esta denominada de *consistência de saída* [Powell 92]. São condições suficientes para obtenção de consistência de saída:

C1. ***Consistência de entrada***: o conjunto de mensagens de entrada deve ser idêntico para todas as réplicas livres de falha; e,

C2. ***Determinismo do grupo de réplicas***: começando de estados iniciais idênticos e processando o mesmo conjunto ordenado de mensagens de entrada, todas as réplicas livres de falha produzem mensagens de saída idênticas e na mesma ordem.

Consistência de entrada implica que qualquer mensagem enviada para um componente de *software* seja entregue a todas ou a nenhuma das réplicas livres de falha daquele componente. A suposição requerida para o sistema de comunicação é então que ele implemente um protocolo de *comunicação em grupo* que garanta *unanimidade*[5] [Powell 92] entre os receptores livres de falha [Chang-Maxemchuk 84; Birman-Joseph 87; Jalote 94, pp. 142-150]. O Capítulo 4 desta dissertação tratará de protocolos que garantam tal propriedade, entre outras.

Já o determinismo do grupo de réplicas pode ser conseguido por:

- garantir que réplicas livres de falha recebam as mesmas mensagens de entrada numa mesma ordem, isto é, numa *ordem total* [Powell 92]; e
- forçar o determinismo da réplica, estruturando os componentes de *software* como *máquinas de estados* e, no caso de ambientes heterogêneos, estabelecendo e respeitando o domínio de replicação para cada componente.

Ordem total pode ser conseguida por um protocolo de comunicação que exiba a propriedade de *ordenação*[6] [Chang-Maxemchuk 84; Cristian et al. 85; Birman-Joseph 87; Jalote 94, pp. 150-170; Kaashoek-Tanenbaum 94]. Quanto ao determinismo das réplicas, o modelo de replicação ativa é de fato baseado na abordagem da máquina de estado, conforme veremos a seguir.

3.6.1. A Abordagem da Máquina de Estado

Uma *máquina de estado* consiste de *variáveis de estado*, que guardam o estado da máquina, e *comandos*, que transformam este estado. Cada comando é implementado por um programa *determinístico*: a execução do comando é atômica com respeito a outros comandos e modifica variáveis de estado e/ou produz alguma saída [Schneider 90]. Um *cliente* da máquina de estado faz um *pedido* para executar um comando.

Um sistema consistindo de um conjunto de componentes distintos é dito ser *t-resiliente* se ele satisfaz sua especificação desde que não mais do que *t* destes componentes falhem durante um dado intervalo de tempo.

---

[5] A propriedade da *unanimidade* é também referenciada na literatura como *atomicidade* [Cristian et al. 85] ou *confiabilidade* [Jalote 94].

[6] A propriedade de *ordenação* é alternativamente referenciada como *ordenação total* [Chang-Maxemchuk 84] ou *ordenação consistente* [Jalote 94].

3.6.2. Processamento de Erro

Diferentes filosofias para processar erros nas mensagens de saída de réplicas ativas podem ser seguidas dependendo da suposição sobre o modo de falha das réplicas [Chérèque et al. 92]:

Réplicas ativas com semântica de falha silenciosa. Neste caso, uma vez que a mensagem de saída enviada por qualquer réplica do grupo pode ser assumida como tendo um valor correto, é possível escolher qualquer das saídas e descartar as demais. Técnicas de replicação para operação continuada na presença de t falhas de réplicas com semântica de falha silenciosa necessitam utilizar apenas $t + 1$ réplicas.

Do ponto de vista das mensagens de saída, a atividade de processamento de erro é reduzida a uma arbitragem simples sobre as cópias múltiplas das mensagens de saída de modo que uma única cópia seja entregue ao destino pretendido. Esta atividade pode ser implementada de duas formas [Powell 92]:

(i) para cada mensagem de saída, um protocolo é executado entre as réplicas a fim de decidir qual delas enviaria a mensagem para o seu destino; ou,

(ii) todas as réplicas propagam toda mensagem de saída e o destino trata de selecionar uma delas e descartar todas as outras.

Para mensagens longas, a primeira abordagem requer obviamente menos demanda do serviço de comunicação. O protocolo de seleção de mensagem de saída usado nesta abordagem pode operar de duas formas distintas: no *modo competitivo* ou no *modo cíclico*. O modo competitivo dá preferência à réplica mais rápida e, por este motivo, pode permitir que as réplicas mais lentas do grupo sigam permanentemente atrás daquelas mais rápidas. É necessário, portanto, que se utilize algum método para limitar a quantidade de dessincronização entre as réplicas. Já no modo cíclico, onde o grupo de réplicas é configurado como um anel lógico com um *token* associado, todas as réplicas são tratadas igualmente e, neste caso, nenhum mecanismo de sincronização é necessário. Maiores detalhes sobre os modos de operação do protocolo de seleção de mensagens podem ser encontrados em [Powell 92, Chérèque et al. 92].

Como qualquer mensagem de saída produzida por réplicas com semântica de falha silenciosa conduz seguramente um valor correto, é possível relaxar a condição de consistência de saída exigida pela abordagem de replicação ativa e otimizar o uso de

réplicas individuais. Uma forma de otimização seria, por exemplo, enviar pedidos que não modificam o estado do componente para apenas uma réplica do grupo; se nenhuma resposta chega dentro de um certo intervalo de tempo, o pedido pode ser re-submetido a uma outra réplica. Tal otimização permite uma diminuição da carga de trabalho do serviço de processamento e do tráfego de mensagens sobre a rede às custas de impor a existência de um mecanismo de seriação leitura-escrita para garantir a consistência das réplicas. Todavia, o benefício decorrente desta otimização geralmente não é considerado, principalmente porque é interessante fornecer mecanismos similares para gerenciar réplicas ativas tanto sob suposição de falha silenciosa quanto de falha arbitrária.

Réplicas ativas com semântica de falha arbitrária. Uma vez que a falha de réplicas com esta semântica de falha pode se manifestar pelo envio de mensagens incorretas, técnicas de replicação para operação continuada na presença de falhas arbitrárias em t réplicas devem ser baseadas em, no mínimo, $2t + 1$ réplicas, de modo que uma minoria de mensagens incorretas possa ser mascarada.

Para processar erros no domínio do tempo, são usados temporizadores para checar que, no mínimo, $t + 1$ réplicas enviam mensagens equivalentes dentro do intervalo de tempo especificado. Erros no valor das mensagens, por sua vez, são processados comparando-se cada mensagem enviada pela réplica local com as mensagens equivalentes enviadas pelas demais réplicas do grupo. Esta checagem é denominada *validação de mensagem*. Tão logo $t + 1$ mensagens concordem, supondo-se que t faltas devem ser toleradas, pode-se assumir seguramente que a mensagem em consenso é correta.

Há dois modos de validação de mensagens [Powell et al. 88, Powell 92]:

a) Valida-antes-de-propagar. Esta técnica requer que, antes da mensagem ser propagada, mensagens de saída equivalentes de, no mínimo, m entre as n réplicas sejam comparadas, com $m > t$ e $n > 2t$, onde t é o número de faltas ativas a serem toleradas. Tão logo m mensagens concordem, pode-se estar certo que aquela mensagem é correta.

Para prevenir que faltas nas $n - m$ réplicas remanescentes passem desapercebidas por um longo tempo, é necessário garantir que todas as réplicas sejam regularmente ativadas e checadas, o que pode ser feito de duas formas:

(i) por rotação das m réplicas cujas mensagens são comparadas, ou seja, mensagens de um conjunto específico de m réplicas são comparadas e tal conjunto é periodicamente mudado para garantir a ativação de todas as réplicas; ou,

(ii) por comparação sistemática das mensagens de saída de todas as n réplicas.

Esta última abordagem é mais simples de se implementar do que a anterior e pode apresentar desempenho aceitável se as últimas $n - m$ mensagens só forem comparadas depois do valor concordado pelas m réplicas já ter sido propagado.

Como no caso de réplicas com semântica de falha silenciosa, o protocolo de seleção da mensagem de saída para a estratégia *valida-antes-de-propagar* pode operar tanto no modo competitivo quanto no modo cíclico.

b) Propaga-antes-de-validar. Desde que nenhuma falta ocorra, esta técnica permite que a computação proceda na velocidade da réplica mais rápida; porém, ela provê apenas a detecção de erro e, por este motivo, algum mecanismo mais complexo deve ser implementado para garantir a sua recuperação. Uma maneira de conseguir isto é criar salvaguardas globais de modo a permitir que a computação seja executada tentativamente e possa ser desfeita se qualquer mensagem usada naquela parte da computação for posteriormente detectada como incorreta. Um *modelo de transações* provê uma estrutura adequada para tal facilidade de salvaguarda global: antes de tornar o resultado da computação permanente e visível fora da transação, todas as mensagens enviadas pelas réplicas dos componentes envolvidos na transação devem ser checadas contra (e concordar com) no mínimo $m - 1$ outras réplicas; isto garante que apenas resultados corretos são liberados externamente à transação. Esta abordagem é atrativa quando a taxa de falhas é baixa; caso contrário, a carga extra de processamento necessária para desfazer a computação pode se tornar inaceitável.

3.7. Replicação Passiva

Quando as réplicas de um componente de *software* apresentam semântica de falha silenciosa, é possível utilizar uma técnica de replicação alternativa, denominada *replicação passiva*, que economiza o serviço de processamento por ativar as réplicas redundantes apenas quando elas são necessárias para garantir a recuperação e dar

continuidade ao serviço. Um componente de *software* replicado passivamente consiste de um grupo de réplicas em que apenas uma réplica - a *réplica primária* - processa todas as mensagens de entrada e provê todas as mensagens de saída [Powell 92]. A suposição de semântica de falha silenciosa garante que estas mensagens são corretas e, desse modo, nenhum protocolo de validação de mensagens é necessário.

As demais réplicas do grupo, denominadas *réplicas suplentes*, consistem de uma cópia do componente de *software* juntamente com uma cópia de alguns dos seus estados anteriores a partir dos quais a execução pode ser reassumida em caso de falha da réplica primária. Os estados internos das réplicas *suplentes* devem ser regularmente atualizados pela réplica primária através de uma operação chamada *salvaguarda*. Réplicas *suplentes* são passivas visto que, na ausência de faltas, elas não executam qualquer processamento, exceto a atualização de seus estados internos.

Este tipo de replicação permite que a capacidade de processamento seja economizada, mas, por outro lado, gera: (a) carga extra de comunicação, que é permanentemente necessária para prover as réplicas suplentes com a salvaguarda para a recuperação de erro; e (b) carga extra de processamento, que é temporariamente necessário quando uma réplica *suplente* assume o papel de réplica primária, tendo assim que reexecutar, a partir do último ponto de salvaguarda, as ações já realizadas pela antiga réplica primária antes de sua falha. [Powell et al. 88].

Na verdade, há um balanceamento entre a sobrecarga permanente de comunicação e a sobrecarga temporária de processamento que deve ser considerado ao se decidir a freqüência com que as salvaguardas devem ser estabelecidas: se a freqüência no estabelecimento de salvaguardas é diminuída, o tráfego nos canais de comunicação também diminui, mas a quantidade de retrocesso aumenta, o que requer mais processamento durante a recuperação; ao contrário, se a freqüência de salvaguardas é aumentada, a quantidade de processamento temporário diminui às custas de um aumento no tráfego da rede.

3.7.1. Estratégias de Salvaguarda

Conforme definido em [Powell 92], uma salvaguarda, no contexto da replicação passiva, consiste de uma fotografia do estado interno da réplica primária, devendo incluir

o seu espaço de dados e toda informação específica da réplica, quais sejam: registradores do processador, apontador de pilha, informação de *status* etc.

Uma réplica primária com grau de replicação $k > 1$ deve emitir salvaguardas de seu estado interno para suas k - 1 *suplentes*. Se o componente de *software* replicado apresenta comportamento não-deteminístico, uma salvaguarda deve ser emitida sempre que ocorrer uma mudança no estado da réplica primária. Como apenas as mudanças desde o último ponto de salvaguarda precisam ser transmitidas e como tais mudanças são tipicamente pequenas, é pouco provável que sua transmissão represente uma sobrecarga de comunicação particularmente custosa [Speirs-Barrett 89].

A freqüência da operação de salvaguarda pode ser diminuída se o determinismo do componente replicado pode ser assumido. Num sistema determinístico, as cópias *suplentes* poderiam, por exemplo, guardar toda mensagem recebida e enviada pela réplica primária desde o ponto de salvaguarda mais recente e, em caso de falha desta réplica, a réplica *suplente* que assumisse o processamento (isto é, a *réplica substituta*) processaria diretamente as mensagens de entrada, podendo assim atingir o mesmo estado alcançado pela réplica primária antes da falha. Tal diário (*log)* de mensagens é usado tanto para prover a réplica substituta com as entradas requeridas para continuar o processamento quanto para evitar a retransmissão das mensagens já enviadas com sucesso pela réplica primária.

Várias estratégias de salvaguarda são possíveis. Uma delas é implementar a técnica de *salvaguarda transacional* em que as interações entre grupos de componentes de *software* são estruturadas como transações. Neste caso, a réplica primária de cada componente de *software* envolvido na transação emite salvaguardas para suas *suplentes* apenas quando mudanças em seu estado são completadas com sucesso.

Na ausência de um modelo transacional de computação, uma outra estratégia de recuperação deve ser empregada. Neste caso, a operação de salvaguarda deve ser organizada de forma a prevenir a ocorrência do conhecido *efeito dominó* [Randell 75, Koo-Toueg 87], onde o retrocesso de um processo causa um "avalanche" de retrocessos, que pode até mesmo conduzir ao estado inicial do sistema [Jalote 94, pp. 189]. A causa deste avalanche é a existência no sistema de *mensagens perdidas* (aquelas "enviadas mas não recebidas", devido ao retrocesso do receptor para um ponto de salvaguarda anterior ao recebimento da mensagem), o que obriga o transmissor a fazer

um retrocesso para um ponto de salvaguarda anterior ao envio da mensagem, e *mensagens órfãs* (aquelas "recebidas mas não enviadas", devido ao retrocesso do transmissor para um ponto de salvaguarda anterior ao envio da mensagem), o que conduz ao retrocesso do receptor para um ponto anterior ao recebimento da mensagem. Note que, a cada retrocesso de um processo, novas mensagens perdidas e órfãs podem surgir no sistema, conduzindo ao retrocesso de outros processos, o que pode gerar novas mensagens perdidas ou órfãs e assim sucessivamente.

No caso da replicação passiva, o efeito dominó pode ser facilmente evitado se a técnica de salvaguarda é tal que: (1) evita a necessidade de a réplica substituta pedir o reenvio das mensagens de entrada previamente processadas pela réplica primária; e (2) previne que a réplica substituta envie mensagens de saída iguais àquelas já enviadas pela réplica primária. As condições 1 e 2 impedem, respectivamente, o aparecimento no sistema de mensagens perdidas e de mensagens órfãs.

A primeira condição pode ser satisfeita tomando-se uma nova salvaguarda cada vez que a réplica primária recebe uma mensagem e modifica seu estado. No caso de componentes de *software* determinísticos, uma outra possibilidade é forçar cada réplica passiva a manter, conforme colocado anteriormente, um diário das mensagens de entrada processadas pela réplica primária desde a última salvaguarda. Esta abordagem tem a vantagem da operação de salvaguarda ser menos freqüente e, conseqüentemente, da sobrecarga de comunicação ser mais baixa, especialmente se as filas de mensagens de entrada forem criadas concorrentemente com o fluxo normal de mensagens intercomponentes, por exemplo, por meio de um protocolo de comunicação em grupo que explora a capacidade de *difusão* (*broadcast*) ou *difusão restrita* (*multicast*) da própria rede de comunicação[7] [Tanenbaum 92, pp. 447].

A duplicação das mensagens de saída, por sua vez, pode ser evitada pelo uso de uma das seguintes técnicas: *salvaguarda sistemática* e *salvaguarda periódica* [Powell 92]. Salvaguarda sistemática envolve a criação de pontos de salvaguarda sempre que a réplica primária comunica algum dos seus dados internos para o mundo exterior, isto é,

[7] Em algumas redes de computadores, é possível criar um endereço especial para identificar grupos de máquinas. *Difusão irrestrita* ou simplesmente *difusão* (*broadcast*) é uma funcionalidade existente na maioria das redes locais onde uma mensagem enviada para um determinado endereço é entregue a todas as máquinas daquela rede. Já a *difusão restrita* (*multicast*) permite que um transmissor envie uma mensagem para todas as máquinas pertencentes ao grupo endereçado.

sempre que uma mensagem é enviada. Assim, o retrocesso para o último ponto de salvaguarda nunca requer reenvio das mensagens de saída.

Já a estratégia de salvaguarda periódica reduz o número de salvaguardas por emiti-las, digamos, a cada *n* mensagens de saída [Borg et al. 1983]. Cada réplica *suplente* deve registrar todas as mensagens de saída produzidas pela réplica primária. Durante a recuperação, quaisquer mensagens de saída geradas pela réplica substituta é primeiro checada contra o diário de mensagens previamente enviadas pela antiga réplica primária e apenas aquelas que não possuem equivalentes no diário são propagadas.

Uma restrição da técnica de salvaguarda periódica é que, para se conseguir recuperação correta, as réplicas devem ser determinísticas e as mensagens devem ser recebidas por todas as réplicas do grupo na mesma ordem (como para as estratégias de replicação ativa), de modo que a réplica substituta produza as mesmas mensagens de saída produzidas pela réplica primária antes da falha.

No caso de salvaguarda transacional ou sistemática, os requisitos de determinismo e ordenação são desnecessários visto que qualquer execução baseada numa ordem de mensagens de entrada que respeite a *causalidade* [Birman-Joseph 87; Schneider 90; Birman et al. 91; Jalote 94, pp. 142] é uma execução válida. A propriedade de preservação de causalidade requer que a ordem em que as mensagens são entregues aos destinos seja consistente com a relação causa-efeito entre os eventos de envio destas mensagens (vide Seção 4.6.2).

Embora a salvaguarda sistemática envolva mais sobrecarga de comunicação do que a transacional e a periódica, sua capacidade para acomodar processamento não-determinístico (ao contrário da salvaguarda periódica) e sua adequação para implementar tolerância a faltas independente do modelo computacional dos componentes de *software* (ao contrário da salvaguarda transacional) são vantagens muito importantes sobre as outras duas abordagens. A Arquitetura Delta-4, por exemplo, proposta para prover tolerância a faltas em sistemas distribuídos abertos, utiliza a abordagem de salvaguarda sistemática [Powell 92].

3.7.2. Otimizando os Mecanismos de Salvaguarda

É possível tornar a replicação praticamente transparente para o programador através da adoção de um modelo orientado a objetos: o grau de replicação do objeto e a localização das réplicas seriam especificados quando o objeto fosse instanciado, mas, a partir daí, o programador não mais se preocuparia com replicação e distribuição.

Usando este esquema, o tamanho da salvaguarda estaria diretamente relacionado à quantidade de dados armazenados dentro de um objeto, o que pode obviamente ser muito grande. [Speirs-Barrett 89] apresenta algumas otimizações para minimizá-lo:

a) Compressão da salvaguarda

Um algoritmo de compressão de dados pode ser aplicado as salvaguardas antes da transmissão. Para um objeto típico, razões de compressão de até 3 ou 4 para 1 podem ser conseguidas.

b) Salvaguarda seletiva

É raramente necessário fazer salvaguardas do espaço de dados completo de um objeto, pois grande parte dele pode não estar correntemente em uso. Se salvaguarda seletiva é empregada, a quantidade de dados transmitidos pode ser bastante reduzida.

c) Salvaguarda apenas das mudanças

Uma minimização no tamanho da salvaguarda pode ser conseguida através da cópia apenas daquelas locações de memória que foram alteradas depois do estabelecimento da última salvaguarda. Uma forma de conseguir isto é manter, em memória, uma fotografia do espaço de dados do objeto no tempo da salvaguarda para compará-lo com o estado corrente na próxima salvaguarda. Embora este mecanismo imponha alguma carga adicional no serviço de processamento, a sobrecarga de comunicação é minimizada.

d) Indicação de página "suja"

Com suporte apropriado do sistema operacional e do *hardware*, é possível associar a cada página de memória um campo (*flag*) cujo propósito é indicar se aquela página foi ou não atualizada desde o último ponto de salvaguarda. Usando salvaguarda incremental apenas as páginas modificadas seriam salvas. Esta otimização, porém, apresenta um custo de implementação alto.

e) Armazenagem local de salvaguardas

Em vez das salvaguardas serem transmitidas através da rede, elas seriam escritas num dispositivo de armazenagem de acesso compartilhado, estável, a partir do qual elas seriam lidas diretamente pelas réplicas *suplentes*, se requerido.

3.7.3. Processamento do Erro

Como a técnica de replicação passiva trabalha apenas sob suposição de semântica de falha silenciosa, a detecção da falha de uma réplica pode ser reduzida à detecção do seu silêncio, o que pode ser conseguido através da troca de mensagens "I'*m alive*". A falha de uma réplica seria detectada quando ela deixasse de enviar tal mensagem dentro de um intervalo de tempo pré-determinado. Esta estratégia de detecção pode ser empregada com segurança se o sistema em questão é síncrono. Para sistemas assíncronos, porém, em que não há garantia de um limite de tempo finito dentro do qual a função pretendida seria de fato realizada caso nenhuma falta no sistema ocorresse, não há como diferenciar a situação em que o sistema realmente falhou daquela em que ele apenas está muito lento. Para sistemas assíncronos só é possível *supor* a ocorrência de uma falha, mas não há como se ter certeza disso.

Se a réplica primária de um componente de *software* falha, uma réplica substituta deve ser selecionada para executar a recuperação a partir do último ponto de salvaguarda. Esta seleção pode ser realizada, por exemplo, por meio de eleição dinâmica (tal como o protocolo competitivo da replicação ativa) ou pode ser baseada numa ordem preestabelecida entre as réplicas *suplentes* (por exemplo, uma cadeia linear com a réplica *suplente* operante de numeração mais baixa se tornando a nova réplica primária). Vários métodos de eleição de uma nova réplica primária são propostos em [Garcia-Molina 82].

Formas alternativas para detecção de falhas da réplica primária são [Powell 92]:

(i) monitoração do intervalo de tempo entre um pedido por um serviço e a resposta correspondente (um modelo de computação *cliente-servidor* é necessário para isto ser possível);

(ii) utilização das facilidades construídas no sistema operacional local, como por exemplo sinais de terminação, verificação da tabela de processos ativos etc.

A ação de recuperação se dá resumidamente da seguinte forma: (1) as áreas de dados e de pilha requeridas para a réplica substituta são alocadas e iniciadas a partir da informação contida na salvaguarda; (2) os registradores do processador e o apontador de pilha são atualizados a partir dos valores armazenados na salvaguarda; e (3) a réplica substituta começa a execução no ponto exato do código do componente de *software* em que aquela salvaguarda foi emitida.

3.8. Replicação Semi-Ativa

[Barrett et al. 90] levanta algumas desvantagens dos modelos de replicação discutidos nas seções anteriores, sobretudo no que diz respeito ao seu uso em aplicações críticas de tempo real, conforme sumariado abaixo.

Replicação ativa:

- requer alguma forma de difusão restrita atômica, cujos protocolos são necessariamente complexos;
- todas as réplicas devem se comportar de maneira idêntica com respeito às mensagens consumidas e produzidas (determinismo do grupo de réplicas);
- se é requerido que o componente de *software* replicado responda rapidamente a eventos externos através de algum tipo de preempção, como é comum em sistemas de tempo real, a dificuldade é multiplicada; preempção é difícil de ser sincronizada em réplicas ativas visto que cada réplica deve sofrer a preempção exatamente no mesmo ponto do processamento, mas as velocidades das diversas unidades de processamento do sistema distribuído são geralmente diferentes.

Replicação passiva:

- quando a réplica primária falha, há um atraso na provisão do serviço enquanto a recuperação e reexecução são realizadas; tal atraso pode não ser compatível com os requisitos de tempo real de muitas aplicações.

Apesar das desvantagens supracitadas, a replicação ativa tem uma vantagem significativa sobre a replicação passiva com relação ao desempenho visto que: (a) a carga extra de comunicação devido à operação de salvaguarda é evitada; e (b) o tempo de recuperação é mais baixo já que as computações redundantes são executadas em

paralelo. Entretanto, a replicação passiva tem a importante vantagem de não requerer que as réplicas sejam determinísticas, além de economizar a capacidade de processamento do sistema. A técnica discutida nesta seção tenta eliminar as desvantagens e tirar proveito das vantagens de ambos os paradigmas de replicação a fim de prover recuperação rápida apesar do não-determinismo potencial do componente de *software* replicado.

3.8.1. O Modelo de Replicação Líder-Seguidor

Como o próprio nome sugere, a técnica de *replicação semi-ativa* é um híbrido entre as técnicas de replicação ativa e passiva. Como na replicação passiva, na ausência de faltas, apenas uma das réplicas, denominada de *líder*, produz mensagens de saída. Porém, ao contrário daquela técnica, as demais réplicas do grupo, denominadas de *seguidoras*, não são completamente passivas (daí o termo *semi-ativa)* - elas processam as mesmas entradas da réplica líder e autonomamente executam toda a computação determinística, atualizando seu estado local de acordo. A réplica líder é responsável por tomar as decisões que afetam o determinismo e informar às seguidoras sobre estas decisões. Graças aos nomes dados as réplicas, a técnica de replicação semi-ativa é alternativamente chamada de *replicação líder-seguidor.*

A suposição sobre o modo de falha das réplicas é que elas apresentam semântica de falha silenciosa e, portanto, a validação das mensagens de saída não é requerida. Isto permite que tais mensagens sejam enviadas imediatamente após terem sido geradas, ao contrário do modelo de réplicas ativas. Como foi dito, a princípio, apenas a réplica líder propaga mensagens, mas, se é desejado obter o máximo de continuidade de serviço e o mínimo de atraso na provisão de mensagens, uma alternativa é permitir que todas as réplicas possam fazê-lo. Neste caso, algum mecanismo deve ser empregado para garantir que uma única cópia da mensagem seja entregue ao destino pretendido, como descrito para a replicação ativa aplicada à semântica de falha silenciosa (vide Seção 3.6.2).

Embora na replicação semi-ativa todas as réplicas devam *consumir* as mesmas mensagens exatamente na mesma ordem, pois, do contrário, seus caminhos de execução podem divergir e o determinismo do grupo de réplicas pode ser comprometido, tal técnica não requer que as mensagens sejam *entregues* a todas as réplicas numa ordem idêntica, o que simplifica o requisito de suporte à comunicação. A própria réplica líder pode se encarregar de indicar a suas seguidoras a ordem em que as mensagens devem ser

consumidas. Isto pode ser feito da seguinte forma: quando a líder seleciona a próxima mensagem de entrada, ela constrói uma *mensagem de sincronização* atribuindo um novo número de seqüência àquela mensagem de entrada e a envia para suas seguidoras, que devem então consumir as mensagens recebidas na ordem ditada pelo líder, favorecendo assim o determinismo.

O não-determinismo que a abordagem líder-seguidor ajuda a resolver pode resultar de um requisito, bastante comum dos sistemas de tempo real, para se tratar preempção, que pode ocorrer em pontos diferentes da execução nas diferentes réplicas [Powell 92]. Neste caso, é necessário se dispor de algum mecanismo que force as réplicas seguidoras a sofrerem a preempção no mesmo ponto de execução que a líder.

A arquitetura Delta-4, por exemplo, adotou um modelo líder-seguidor que incorpora um mecanismo de sincronização de preempção que impõe uma carga extra de processamento e comunicação muito pequena. Este mecanismo faz uso do conceito de *pontos de preempção*, que são pontos pré-definidos do processamento nos quais o componente pode sofrer a preempção. A sincronização é conseguida da seguinte forma: cada vez que a réplica líder alcança um ponto de preempção, um contador é incrementado. Quando uma mensagem chega à líder, uma checagem é feita para determinar se esta mensagem requer que tal réplica sofra uma preempção. Em caso positivo, o ponto de preempção é selecionado (o valor do contador corrente mais 1 representa o próximo ponto de preempção) e uma mensagem de sincronização, contendo este valor e identificando a mensagem, é construída e despachada para as seguidoras. Ao chegar no ponto de preempção designado (isto é, quando seus contadores combinam com o valor indicado na mensagem de sincronização), cada réplica seguidora começa a processar a preempção [Barrett et al. 90].

Para que este mecanismo funcione, as seguidoras devem sempre estar executando no mínimo um passo atrás da líder, onde um *passo* constitui o recebimento de uma mensagem de sincronização devido ou a uma preempção ou à determinação da ordem de consumo de uma mensagem de entrada pela líder.

3.8.2. Sistema de Comunicação

Uma das vantagens do modelo líder-seguidor é que ele pode se apoiar em mecanismos de comunicação relativamente simples. Uma vez que a ordem de consumo

das mensagens pode ser ditada pela líder, a ordenação consistente de entrega de mensagens não é mais necessária. Portanto, um serviço de comunicação atômico não mais é requerido, podendo ser substituído por uma alternativa mais simples e eficiente, tal como um serviço de difusão restrita confiável [Barrett et al. 90]. Este tipo serviço é tal que se uma mensagem chega a um destino livre de falha, ela chegará a todos os outros destinos livres de falha dentro de um tempo limitado. O mecanismo normal para se implementar este requisito é o de *reconhecimento e reenvio de mensagens.*

O uso do serviço de difusão restrita confiável traz vários benefícios para o desempenho do sistema:

- o número total de mensagens enviadas é reduzido em comparação com um sistema baseado no serviço de difusão restrita atômica;
- o atraso efetivo de propagação de mensagens é reduzido, visto que as mensagens podem se tornar disponíveis para a réplica líder imediatamente após a sua chegada pela rede de comunicação;
- o serviço de difusão restrita confiável é mais simples do que o serviço atômico e, portanto, sob circunstâncias normais (e para muitas condições de falha), o sistema de comunicação é inerentemente mais eficiente.

3.8.3. Processamento do Erro

Do mesmo modo que a técnica de replicação passiva, a replicação semi-ativa trabalha sob suposição de semântica de falha silenciosa e, sendo assim, a detecção da falha de uma réplica pode ser reduzida à detecção do silêncio da réplica.

Já a ação de recuperação é mais simples do que na replicação passiva visto que, por seu próprio princípio, a técnica de replicação semi-ativa garante que o estado interno das réplicas seguidoras permanece a todo tempo consistente com aquele da réplica líder. Quando uma falha na réplica líder é detectada, uma réplica seguidora é selecionada para assumir o papel de líder, atualizando seu estado pelo processamento das mensagens presentes na fila de entrada, se houver. A seleção de uma nova líder pode ser executada por meio de uma eleição dinâmica ou pode ser baseada numa ordenação preestabelecida entre as réplicas seguidoras, de forma bastante semelhante àquela adotada pela replicação passiva.

3.9. Sumário

Neste capítulo, vimos como tolerância a faltas pode ser conseguida através da replicação de componentes de *software* em várias unidades de processamento do sistema distribuído. As duas principais semânticas de falha foram discutidas: silenciosa e arbitrária. Os conceitos de domínio de replicação, determinismo de réplicas e determinismo do grupo de réplicas foram introduzidos. Três técnicas de replicação foram detalhadamente estudadas, considerando-se seus requisitos, sobretudo quanto à suposição da semântica de falha, às propriedades do sistema de comunicação e à exigência ou não de determinismo para os componentes de *software*.

Na replicação ativa, todas as réplicas processam concorrentemente todas as mensagens de entrada e, na ausência de faltas, todas elas produzem as mesmas mensagens de saída e na mesma ordem. Tal modelo pode ser empregado para tolerância a faltas sob suposições tanto de falha silenciosa quanto de falha arbitrária. A fim de tolerar falhas arbitrárias, as saídas de todas as réplicas são votadas e a decisão majoritária é usada. No caso de semântica de falha silenciosa, esta técnica pode ser usada sem votação uma vez que qualquer mensagem produzida pode ser assumida como correta. A exigência para o sistema de comunicação é que ele garanta entrega atômica de mensagens.

Na replicação passiva, apenas uma das réplicas (a *réplica primária*) processa as mensagens de entrada e provê mensagens de saída - na ausência de faltas, os estados internos das demais réplicas (as *réplicas suplentes*) são regularmente atualizados por meio de salvaguardas da réplica primária. Se a réplica primária falha, uma suplente é ativada e começa a executar a partir de seu ponto de salvaguarda mais recente. A suposição de semântica de falha silenciosa garante que as mensagens propagadas pela líder são sempre corretas. Como na replicação ativa, o sistema de comunicação deve garantir entrega confiável e ordenada, a menos que a estratégia de salvaguarda sistemática seja adotada.

A abordagem de replicação semi-ativa pode ser vista como um híbrido das duas técnicas anteriores: todas as réplicas processam todas as mensagens de entrada, mas apenas uma delas (a *réplica líder*) provê mensagens de saída. As decisões que afetam o determinismo são tomadas pela líder e comunicadas às demais réplicas (as *seguidoras*).

Do mesmo modo que a replicação passiva, a suposição de falha silenciosa garante que as mensagens geradas pela líder são corretas, dispensando qualquer validação. O requisito de ordenação para o sistema de entrega de mensagens pode ser relaxado, pois a réplica líder pode ditar a ordem de consumo das mensagens para as suas seguidoras; dessa forma, um serviço de comunicação confiável é suficiente.

A Tabela 3.1 resume os principais parâmetros a serem considerados no momento da escolha entre uma das técnicas apresentadas acima [Powell 92].

Técnica de replicação	Sobrecarga do processamento de erro	Não-determinismo da réplica	Comportamento em falha arbitrária
Ativa	Mais baixo	Proibido	Tolerado
Passiva	Mais alto	Permitido	Proibido
Semi-ativa	Baixo	Resolvido	Proibido

Tabela 3.1: Principais Características das Técnicas de Replicação

Como veremos no Capítulo 7, o Seljuk-Amoeba oferece suporte para os três modelos de replicação aqui discutidos, ficando a cargo do usuário do ambiente definir qual das técnicas atende melhor às suas expectativas.

Capítulo 4:

Comunicação em Grupo

4.1. Introdução

Aplicações distribuídas são geralmente estruturadas como um grupo de processos que cooperam para prover um serviço. Para que esta cooperação seja possível, cada membro do grupo deve ser capaz de trocar mensagens com o restante do grupo e, muitas vezes, com membros de outros grupos. É útil, portanto, que se tenha disponível alguma forma de comunicação que permita a aplicação enviar uma única mensagem para n destinos de uma só vez.

A maioria dos sistemas operacionais, entretanto, provêem apenas comunicação ponto-a-ponto, que obriga um processo pertencente a um grupo com n membros enviar n -1 mensagens ponto-a-ponto sempre que ele deseja se comunicar com o resto do grupo. Em grande parte dos sistemas, isto custaria no mínimo $2(n\text{-}1)$ pacotes (um pacote para a mensagem e outro para o seu reconhecimento). Se a mensagem é fragmentada em vários pacotes antes de ser enviada, o custo se tornaria ainda maior. Estes fatos revelam que a comunicação ponto-a-ponto, quando utilizada por aplicações distribuídas, é lenta e ineficiente [Kaashoek-Tanenbaum 94].

Além do mais, projetar aplicações distribuídas usando apenas comunicação ponto-a-ponto é uma tarefa difícil. Considere, por exemplo, o caso de dois servidores agindo sobre um banco de dados replicado - se ambos os servidores recebem pedidos para atualizar um mesmo dado, estas atualizações devem ser realizadas numa ordem única em ambas as réplicas; caso contrário, as cópias do dado podem se tornar inconsistentes. Como o sistema de comunicação ponto-a-ponto não garante entrega ordenada de mensagens, os servidores devem ser construídos de modo que eles próprios tratem este tipo de situação, o que torna mais árduo o trabalho do projetista da aplicação.

No contexto deste trabalho, o conceito de grupos de processos, como introduzido anteriormente, é empregado para aumentar a confiança no funcionamento do sistema,

através da replicação do processamento. Neste caso, o grupo é formado pelas diversas réplicas do componente de *software* que é desejável ser tolerante a faltas.

As técnicas de replicação apresentadas no capítulo anterior dependem basicamente de dois serviços [Chérèque et al. 92]:

(i) Gerência da afiliação do grupo: como um componente de software é instanciado como um grupo de réplicas e como a afiliação do grupo é atualizada como conseqüência de falhas e reparos?

(ii) Coordenação inter-réplicas: como a atividade do grupo de réplicas é coordenada a fim de processar erros e dar a ilusão para outros componentes do sistema de um componente de software único, livre de falha?

Prover tais serviços requer alguma demanda do sistema de suporte à comunicação, o que depende da técnica de replicação particular. A abordagem da replicação ativa, por exemplo, requer que o protocolo básico de comunicação atenda às propriedades de *acordo* e *ordenação* quando da disseminação de mensagens para múltiplos destinos. Tal demanda pode ser atendida pelo uso de primitivas adequadas de comunicação em grupo.

As diversas propriedades dos protocolos para comunicação em grupo, bem como outros aspectos relacionados, serão discutidos neste capítulo, que inicia apresentando algumas noções sobre grupos de processos e comunicação em grupo. Finalizaremos o capítulo discutindo algumas questões de desenho que diferenciam os sistemas operacionais existentes quanto ao suporte à comunicação em grupo e caracterizando alguns dos principais protocolos para comunicação em grupo propostos na literatura.

4.2. Noções sobre Grupos de Processos

A idéia de estruturar um sistema distribuído como grupos de processos não é nova. Os sistemas V [Cheriton-Zwaenepoel 85], CIRCUS [Cooper 85] e ISIS [Birman-Joseph 87] usaram o conceito de grupos de processos para este propósito. Nestes sistemas, um grupo de processos é definido como um conjunto de processos agrupados para cooperativamente fornecerem um serviço. Há vários motivos que justificam a existência de grupos de processos: distribuição da carga de trabalho, compartilhamento de recursos, necessidade de paralelismo, aumento da confiabilidade e disponibilidade do serviço, entre outros.

Neste trabalho, estamos particularmente interessados em empregar o conceito de grupos de processos na implementação de tolerância a faltas, objetivando, portanto, aumentar a confiança no funcionamento do sistema.

Grupos de Processos Tolerantes a Faltas

Já vimos que, para garantir que um serviço permaneça disponível aos seus clientes apesar de falhas do servidor, o serviço pode ser implementado como um *grupo* de servidores redundantes, fisicamente independentes, de modo que, se alguns deles falham, os servidores remanescentes podem continuar a prover o serviço adequadamente.

Seguindo os conceitos introduzidos em [Cristian 91], nós dizemos que um grupo *mascara* a falha de um membro *m* se o grupo como um todo responde aos seus usuários conforme o especificado apesar da falha de *m*. Um grupo de servidores que está hábil a mascarar *t* falhas simultâneas de quaisquer dos seus membros é chamado de *tolerante-a-t-faltas* (ou alternativamente, *t-resiliente*). Neste caso, as falhas de membros individuais são inteiramente escondidas dos usuários por meio dos mecanismos de gerência de grupo. Complementando, a frase "o grupo *g* tem semântica de falha *F*" é usada como um resumo para "as falhas prováveis de serem observadas pelos usuários de *g* são da classe *F*". Classes possíveis de falha são: silenciosa ou parada (*crash*), omissão, temporização e Bizantina [Cristian et al. 86].

Uma regra geral da computação tolerante a faltas é: "Quanto mais forte (mais restritiva) é a semântica de falha dos membros do grupo e da comunicação, mais simples e mais eficientes podem ser os mecanismos de gerência de grupo. Em contraposição, quanto mais fraca (menos restritiva) a semântica de falha dos membros e da comunicação, mais complexos e caros se tornam os mecanismos de gerência de grupo" [Cristian 91].

Classificação de Grupos de Processos

De acordo com o seu comportamento externo, grupos de processos distribuídos podem ser classificados em duas categorias principais: determinísticos e não-determinísticos. Um grupo é considerado determinístico se, na ausência de faltas, dado o mesmo estado inicial e o mesmo conjunto ordenado de pedidos, cada um dos seus membros produz as mesmas respostas e na mesma ordem. Por outro lado, um grupo é dito ser não-determinístico se cada membro pode responder diferentemente a um pedido

ou simplesmente não responder [Liang et al. 90]. Entre as estratégias de processamento de erro vistas no capítulo anterior, apenas a replicação ativa é baseada em grupos determinísticos; as outras duas - a replicação passiva e a semi-ativa - implementam mecanismos próprios para tratar o não-determinismo.

4.3. Noções sobre Comunicação em Grupo

Em redes de computadores, a *difusão irrestrita*, ou simplesmente *difusão*, é um mecanismo de transmissão "um-para-todos", que entrega uma mensagem enviada por um transmissor único a *todos* os possíveis destinos do sistema, permitindo aos seus receptores processarem a mensagem concorrentemente. Uma variação da difusão, onde as mensagens são entregues a um *subconjunto* dos possíveis destinos do sistema, é chamada de *difusão restrita* (transmissão "um-para-muitos").

Em contraste à difusão restrita, uma abstração de comunicação em rede, *comunicação em grupo* é uma abstração a nível de sistema operacional que oferece conveniência e clareza ao programador [Liang et al. 90]. Porém, por questões de generalidade e compatibilidade com a literatura corrente, daqui por diante utilizaremos o termo *difusão restrita* como um sinônimo para *comunicação em grupo*, como é adotado por grande parte da bibliografia.

Sob este prisma, um grupo é composto por objetos que compartilham uma semântica de aplicação comum e um mesmo identificador de grupo e/ou endereço de difusão restrita. Cada grupo é visto como uma entidade lógica única, sem expor sua estrutura interna e interações ao seu ambiente externo. Atualmente, poucos sistemas operacionais oferecem suporte para esta abstração, dentre os quais destaca-se o Amoeba.

O uso da comunicação em grupo traz melhorias tanto para a eficiência do sistema quanto para a conveniência do programador porque [Liang et al. 90]:

(i) ela entrega uma única mensagem a múltiplos receptores, podendo tirar vantagem da capacidade de difusão irrestrita ou restrita da maioria das redes existentes no mercado, reduzindo assim a sobrecarga de processamento do transmissor e o tráfego da rede;

(ii) ela provê uma abstração de comunicação de alto nível que simplifica a interação dos programas de usuários (clientes) com um grupo de servidores; e

(iii) ela esconde das aplicações as coordenações internas de um grupo (por exemplo, mudanças na sua afiliação).

Com tudo isto, a comunicação em grupo provê o suporte de comunicação exigido pelos dois serviços básicos em que se apóiam as técnicas de processamento de erro baseadas em replicação: a coordenação inter-réplica e a gerência da afiliação do grupo, como veremos nas subseções a seguir. Qualquer técnica de replicação pode tirar vantagem das facilidades da comunicação em grupo, usando-as como um meio de simplificar a implementação dos protocolos de detecção, recuperação e compensação do erro.

O sistema de comunicação da arquitetura Delta-4 [Powell 92], por exemplo, batizado de *Multicast Communication System* - MCS, é hábil para manipular a comunicação entre os componentes replicados de uma maneira transparente para o programador da aplicação. O protocolo de processamento de erro, chamado de *Inter Replica Protocol* - IRp, trabalha em um nível mais alto, confiando nos serviços providos pelas camadas mais baixas do sistema de comunicação, especialmente no serviço de difusão restrita atômica e de notificação de falhas das estações.

4.3.1. Suporte para Replicação

As técnicas de replicação apresentadas no capítulo anterior possuem diferentes requisitos quanto à interação entre as réplicas. Estes requisitos são satisfeitos por propriedades específicas do serviço de comunicação, que serão melhor definidas na Seção 4.5. No momento, daremos apenas um visão geral destas propriedades, relacionando-as com as diferentes técnicas de replicação.

A replicação ativa possui o requisito do determinismo, que exige que uma mensagem enviada para o grupo seja entregue a todas as réplicas e na mesma ordem - propriedades chamadas de *unanimidade* e *ordenação total*, respectivamente. Um serviço que provê estas propriedades é chamado de *atômico*.

Já a replicação passiva e a semi-ativa confiam em um participante privilegiado, que é o representante do grupo de réplicas. A presença de tal representante permite que os

requisitos de unanimidade e ordenação total sejam relaxados, pois ele pode ficar responsável por verificar o recebimento das mensagens pelas demais réplicas e por ditar a ordem de entrega das mensagens. Entretanto, pelo menos unanimidade pode ser preservada com vantagem, dado que [Powell 92]:

(i) no caso da replicação semi-ativa, como as réplicas seguidoras têm que executar os mesmos comandos da réplica líder, é interessante que todas elas recebam as mesmas mensagens recebidas por este (e aproximadamente ao mesmo tempo) diretamente do originador das mensagens; neste caso, a ordem de processamento das mensagens precisa ainda ser ditada pela líder;

(ii) no caso da replicação passiva, uma vez que a salvaguarda emitida pela réplica primária para as suas *suplentes* deve afetar igualmente a todas elas, é econômico que a primária envie uma única mensagem de salvaguarda para todas as *suplentes* de uma só vez; como as mensagens são enviadas por um único transmissor, os próprios receptores podem se encarregar de ordená-las segundo o seu número de seqüência; e

(iii) em todas as técnicas de replicação, mudanças no conjunto de réplicas, decorrentes de falhas e reparos das mesmas, devem ser percebidas consistentemente por todo o grupo.

Um serviço de comunicação que provê apenas a propriedade de unanimidade (e, no máximo, ordenação de mensagens de transmissores individuais conforme a ordem de envio, i.e., FIFO) é chamado de *confiável.*

4.3.2. Suporte para Grupos

Há muitas situações em que é necessário que a afiliação do grupo seja conhecida por todos os seus membros, como por exemplo [Powell 92]:

(i) para verificar se os membros atuais reúnem as funcionalidades necessárias para a execução de uma determinada aplicação distribuída;

(ii) para ativar procedimentos de recuperação; e

(iii) para restabelecer o nível de redundância do sistema.

O serviço mais geral de gerência de grupo é prover uma *visão consistente do grupo* para todos os seus participantes. Uma visão de grupo é uma fotografia da afiliação do grupo em um instante particular do tempo. A visão do grupo muda à medida que membros falham e se recuperam, ou são afiliados e desafiliados, em paralelo com outras atividades do grupo. Para se conseguir consistência, é essencial que os membros do grupo observem as mudanças em sua afiliação de uma maneira uniforme, ou seja, cada mudança na afiliação do grupo é indicada, numa ordem total, para todos os participantes daquele grupo. Esta exigência pode ser atendida por um protocolo de comunicação em grupo atômico.

No sistema ISIS [Birman-Joseph 87], por exemplo, em caso de falha de algum membro do grupo, a consistência da visão do grupo é mantida através da emissão de um comunicado, feita pelo próprio sistema em prol do membro que apresentou a falha. Este anúncio chega em cada membro na mesma ordem com respeito às outras mensagens do grupo, de modo que todos os membros vêem a mesma seqüência de transições virtualmente ao mesmo tempo. Isto garante que nenhuma mensagem chega de um membro após sua falha ter sido anunciada. Se o membro se recupera, seu estado deve ser atualizado com uma visão consistente do estado interno do grupo. Além disso, todos os participantes percebem a existência do novo membro antes de receber uma mensagem enviada por ele.

4.4. Requisitos para Transparência de Grupos de Réplicas

A transparência de grupo é uma propriedade desejável que permite um cliente tratar o serviço fornecido por um grupo de servidores como se ele fosse executado por um único servidor. Uma única chamada é feita e um único resultado, se há um, é esperado do grupo de servidores. Dentre os diversos requisitos para transparência de grupo [Liang et al. 90], podemos identificar os apresentados abaixo como sendo os mais comumente exigidos pelas técnicas de replicação discutidas no capítulo anterior.

(1) ***Transparência de comunicação***. Consiste de dois aspectos fundamentais: unanimidade (confiabilidade) e ordenação.

Entrega de mensagem confiável. A propriedade de unanimidade esconde uma falha parcial na comunicação de grupo, convertendo-a em uma falha total: se algum membro do grupo não pode receber a mensagem, nenhum outro membro a receberá.

Ordenação consistente. Se a ordem de entrega das mensagens para os membros do grupo pode afetar os resultados produzidos pelo grupo, tal entrega necessita ser sincronizada: todo membro de um grupo deve processar a mesma seqüência de pedidos.

(2) ***Transparência na manipulação de pedidos e respostas***. Para cada pedido de um *cliente* para um *grupo servidor*, o cliente deveria fazer uma chamada um-para-muitos e então manipular respostas muitos-para-um recebidas dos servidores, que podem ou não ser idênticas. A transparência na manipulação de pedidos e respostas garante que uma aplicação cliente não necessita estar ciente nem do envio de múltiplos pedidos nem do recebimento de múltiplas respostas. Ela envia um único pedido para o grupo e vê uma única resposta sem ter que se preocupar como esta resposta é derivada a partir de muitas outras (p. ex., por meio de votação).

(3) ***Transparência de nomes***. As aplicações normalmente preferem usar um nome lógico para endereçar um serviço fornecido por um grupo em vez de ter de conhecer cada membro individual. O requisito de transparência de nomes envolve, portanto, associar membros do grupo a um nome único de forma dinâmica e transparente. Nomear grupos consiste de duas partes: (i) mapear um nome lógico de serviço em um grupo de servidores; e (ii) permitir a afiliação do grupo mudar dinamicamente. A fim de manter uma visão coerente do grupo, as mudanças na sua afiliação devem ser detectadas de uma maneira consistente. Faz-se, portanto, conveniente seriar mudanças na visão do grupo consistentemente com respeito a outras atividades do grupo.

(4) ***Transparência da falha***. Na maioria das vezes, é desejável que a falha de um membro do grupo de servidores seja escondida dos seus clientes. Neste caso, outros membros do grupo podem assumir o papel do membro que falhou ou, então, a falha pode ser apresentada como uma falha do grupo completo (p. ex., o grupo como um todo deixa de prover o serviço) - a estratégia escolhida depende da função do grupo.

Vejamos um exemplo: quando um grupo determinístico é usado para aumentar a confiabilidade de dados, como em sistemas de arquivos ou banco de dados duplicados, todas as réplicas devem se manter em um mesmo estado. Neste caso, a estratégia de controle de consistência deve transformar uma falha parcial em total: se o estado de uma

réplica não pode ser atualizado, o estado de nenhuma outra réplica do grupo é atualizado. Entretanto, quando um grupo é usado para aumentar a confiabilidade do serviço, como em processamento replicado, manter o serviço para os clientes é importante. Em caso de falha de algum membro do grupo, os membros remanescentes devem continuar a cumprir suas obrigações a fim de minimizar o prejuízo.

(5) ***Requisito de tempo real***. Em sistemas de tempo real, mensagens devem ser entregues dentro de um limite de tempo pré-definido; ao contrário, a mensagem é considerada obsoleta, caracterizando uma falha por temporização. Na comunicação interprocessos um-para-um, um servidor pode simplesmente ignorar uma mensagem que apresente falha por temporização ou pode responder ao cliente com uma falha de operação. Já na comunicação em grupo, embora unanimidade e ordenação sejam garantidas, alguns servidores podem receber a mensagem dentro do tempo especificado, enquanto outros enxergam uma falha por temporização. Quando isto ocorre, servidores devem se coordenar para agirem consistentemente em cada mensagem que apresente falha por temporização em qualquer dos servidores. O tempo de distribuição da mensagem por difusão restrita deve ser limitado de forma que as ações do grupo possam ser escalonadas para ocorrerem atômica e simultaneamente em todos os membros do grupo

4.5. Propriedades da Comunicação em Grupo

Quando um protocolo de comunicação em grupo é usado para disseminar mensagens para diferentes unidades de processamento do sistema distribuído, três propriedades devem ser consideradas: *confiabilidade*, *ordenação consistente* e *preservação de causalidade* [Jalote 94, pp. 141-142].

A propriedade de confiabilidade requer que a mensagem enviada seja recebida por todos os destinos operacionais. A ordenação consistente requer que diferentes mensagens enviadas por diferentes transmissores sejam entregues a todos os destinos na mesma ordem. Finalmente, preservação de causalidade requer que a ordem em que as mensagens são entregues aos seus destinos seja consistente com a causalidade entre os eventos de envio destas mensagens.

Em caso de faltas no sistema, a confiabilidade só pode ser obtida se houver um *acordo* entre os receptores livres de falha. Já as propriedades de ordenação consistente e preservação de causalidade dependem da forma como os eventos são ordenados no sistema distribuído. Nas subseções seguintes, discutiremos as possíveis formas de *acordo* e *ordenação.*

4.5.1. Acordo

A obtenção de acordo distribuído na presença de faltas é assunto de um grande número de publicações. Paradigmas úteis para esta questão foram identificados, entre os quais destaca-se o bem conhecido *Acordo Bizantino* [Lamport et al. 82], cuja intenção é fazer um participante do grupo (o transmissor) disseminar um valor para todos os outros participantes (os receptores) de tal maneira que:

IC1. Todos os receptores livres de falha concordam no mesmo valor.

IC2. Se o transmissor é livre de falha, então todos os receptores consideram o valor por ele enviado como sendo o valor que todos concordam. Caso contrário, um valor *default* é usado.

As condições acima estabelecidas são conhecidas como *Condições da Consistência Interativa* e são satisfeitas por qualquer protocolo que vise solucionar o clássico *Problema dos Generais Bizantinos* - o problema de alcançar acordo em um sistema onde componentes podem falhar de uma maneira arbitrária. Daí porque *falhas arbitrárias* são comumente referenciadas na literatura como *falhas Bizantinas*. Exemplos deste protocolo são dados em [Lamport et al. 82, Dolev-Strong 83].

Para garantir que os participantes se comportem de maneira consistente, formas de acordo, tais como aquelas encontradas em *protocolos de difusão confiável ou atômico* [Birman-Joseph 87; Chang-Maxemchuck 84; Cristian et al. 85; Jalote 94, pp. 142-170] especificam, alternativamente, se uma mensagem pode ou não ser entregue; em caso afirmativo, tal mensagem é entregue a todos os receptores pretendidos. A palavra 'pretendidos' é usada para representar que a especificação pode não incluir todos os componentes do sistema, mas apenas um *grupo* deles, ou para referenciar formas relaxadas de acordo, tais como maioria, no-mínimo-*N* etc.

A forma mais forte de acordo é a *unanimidade* [Powell 92]:

Unanimidade: qualquer mensagem entregue a um receptor, é entregue a todos os receptores livres de falha.

Em outras palavras, isto quer dizer que uma mensagem ou é entregue a todos os receptores ou a nenhum deles. Protocolos de comunicação em grupo que atendem a este requisito são denominados de *protocolos de difusão restrita confiável*.

A propriedade de unanimidade, também referenciada na literatura como *atomicidade* [Cristian et al. 85] ou *confiabilidade* [Jalote 94, pp. 141-142], apresenta um custo desnecessário em algumas situações. Por exemplo, consultas a um grupo de réplicas necessitam alcançar apenas uma das réplicas, ou um *quorum* delas, não importando exatamente qual. Dependendo da condição de validade, isto é, se a mensagem deve sempre alcançar *N* participantes, ou se ela pode não alcançar todos eles, uma das seguintes semânticas de acordo pode ser obtida [Powell 92]:

No-mínimo-*N*: qualquer mensagem entregue a um receptor é entregue a no mínimo *N* receptores livres de falha.

Best-effort-N: qualquer mensagem entregue a um receptor é entregue a no mínimo *N* receptores livres de falha, na ausência de faltas.

4.5.2. Ordenação

Os componentes de um sistema distribuído devem observar como o sistema se desenvolve. Cada participante observa a evolução do sistema como uma seqüência de eventos, que pode não ser a mesma em todos os participantes devido à relatividade de suas posições. Além do mais, como eventos são gerados de forma independente nos diferentes componentes do sistema e como um sistema distribuído não possui relógio global único, o problema de definir os tempos relativos em que diferentes eventos ocorrem é difícil de solucionar.

Uma forma de ordenar eventos em um sistema distribuído utilizando *relógios lógicos* em vez de relógios físicos foi proposta em [Lamport 78], cuja idéia geral será aqui apresentada. Considere dois eventos *a* e *b* ocorrendo em um mesmo processo *p*. Nós dizemos que *a precede b*, se o evento *a* ocorre antes do evento *b* na seqüência de eventos realizadas por *p*.

Agora, vamos estender a relação *precede* (ou *ocorre antes*, como é preferido por alguns autores [Jalote 94, pp. 64]) para eventos que ocorrem em diferentes processos de um sistema distribuído. Em um sistema cuja comunicação se dá por meio de passagem de mensagens, como é o caso dos sistemas distribuídos, o recebimento de uma mensagem não pode ocorrer obviamente antes do seu envio. Assim, podemos dizer que "o envio de uma mensagem *precede* o recebimento daquela mesma mensagem". Esta ordenação de eventos baseada no envio/recebimento de mensagens é o mecanismo básico para a ordenação de eventos de diferentes processos.

A definiça formal da relação *precede* apresentada em [Lamport 78] é a seguinte:

Definição. A relação ➔ em um conjunto de eventos de um sistema distribuído é a menor relação satisfazendo as três condições seguintes: (1) Se a e b são eventos ocorridos em um mesmo processo e a ocorre antes de b, então a ➔ b. (2) Se a é o envio de uma mensagem por um processo e b é o recebimento desta mesma mensagem por um outro processo, então a ➔ b. (3) Se a ➔ b e b ➔ c, então a ➔ c. Dois eventos distintos são ditos *concorrentes* se nem a ➔ b nem b ➔ a.

Já que há eventos que não são ordenados pela relação *precede* - os eventos concorrentes, esta relação define apenas uma *ordem parcial* de eventos. A relação causa-efeito é a ordem parcial natural de eventos em um sistema. Portanto, uma outra maneira de ver a relação ➔ é dizer que se a ➔ b implica que o evento a pode causalmente afetar o evento b. Se os eventos são concorrentes, então nenhum deles pode afetar causalmente o outro [Jalote 94, pp. 64]. No nosso contexto, um evento significa basicamente o envio ou recebimento de mensagens. Uma *ordem causal* na comunicação pode então ser definida como [Powell 92]:

Ordem Causal: se quaisquer duas mensagens entregues a qualquer receptor livre de falha não são concorrentes, elas são entregues a todo receptor em sua ordem de precedência. Se as mensagens são concorrentes, sua ordem de entrega é indefinida.

Um protocolo para comunicação em grupo que respeita a ordem causal dos eventos - e, portanto, apresenta a propriedade de preservação da causalidade - é denominado de *protocolo de difusão restrita causal.*

Há muitos sistemas, porém, que, independente da maneira como os seus componentes causalmente se relacionam, necessitam definir uma *ordem total* dos

eventos. Como exemplo, podemos citar os sistemas baseados em replicação ativa, que exigem que todas as réplicas de um componente de *software* processem as mesmas mensagens na mesma ordem. Uma maneira de conseguir isto é garantir que elas recebam todas as mensagens na mesma ordem. Uma ordem total é definida como [Powell 92]:

Ordem Total: quaisquer duas mensagens entregues a receptores livres de falha são entregues na mesma ordem àqueles receptores.

A propriedade de ordem total só pode ser aplicada àquelas mensagens entregues a *todos* os receptores livres de falha do sistema. Portanto, para garantir a entrega ordenada de *todas* as mensagens enviadas, antes mesmo de atender ao requisito de ordenação total, é necessário atender ao requisito de unanimidade. A combinação das propriedades de ordenação total e unanimidade (ou, com outra nomenclatura, ordenação consistente e confiabilidade [Jalote 94, pp. 141-142]) produz o chamado *protocolo de difusão restrita atômica*.

O requisito de ordenação total pode ser satisfeito atribuindo-se identificadores únicos às mensagens e fazendo com que elas sejam processadas por seus receptores de acordo com a relação de ordem total determinada por estes identificadores. Em [Lamport 78] é proposta uma forma de se colocar ordem total nos eventos de um sistema distribuído usando relógios lógicos.

Um relógio lógico não tem relação direta com a noção do tempo físico, real. O relógio lógico C_i para um processo P_i é uma função que atribui um valor $C_i(a)$ a um evento a do processo P_i [Jalote 94, pp. 65]. Um sistema de relógios lógicos é considerado *correto* se os valores atribuídos aos eventos são consistentes com a relação *precede* apresentada acima, ou seja, se ele satisfaz a seguinte condição [Lamport 78]:

Condição de Relógio. Para quaisquer eventos a e b, se $a \rightarrow b$ então $C(a) < C(b)$.

Uma maneira bastante fácil de implementar um sistema de relógios lógicos é também apresentada em [Lamport 78]. Suponha que, quando uma mensagem m é enviada por um processo P, o *timestamp* do evento de envio é incluído na mensagem m e pode ser recuperado pelo receptor. Se o *timestamp* do receptor é menor do que aquele indicado na mensagem, o receptor atualiza seu *timestamp* de acordo com o recebido. Um método simples para se obter ordenação total dos eventos é ordená-los de acordo com os *timestamps* atribuídos pelo sistema de relógio lógico. Visto que os eventos gerados

por processos diferentes podem apresentar o mesmo *timestamp*, uma outra forma de ordenação deve ser usada para fazer o "desempate", como, por exemplo, a ordem alfabética dos nomes dos processos.

Baseado no conceito de relógios lógicos, a relação de ordenação total $\Rightarrow$ pode ser definida [Lamport 78]:

Definição. Para dois eventos a e b dos processos P_i e P_j respectivamente, $a \Rightarrow b$ se e apenas se ou (i) $C_i(a) < C_j(b)$, ou (ii) $C(a)=C(b)$ e P_i vem antes de P_j na ordem (p. ex., alfabética) dos processos.

Como vimos, nos modelos de replicação passiva e semi-ativa, há uma réplica privilegiada que realiza as operações de ordenação e instrui as demais réplicas. Neste caso, um protocolo mais simples pode ser usado, provendo apenas unanimidade, e nenhuma ordenação ou, no máximo, ordenação FIFO (*First In First Out*) [Powell 92]:

Ordem FIFO: se quaisquer duas mensagens, entregues a quaisquer receptores livres de falha, foram enviadas pelo mesmo participante, elas são entregues na ordem de envio. Se as mensagens não foram enviadas pelo mesmo transmissor, sua ordem de entrega é indefinida.

4.6. Questões de Desenho em Comunicação em Grupo

Atualmente, poucos sistemas operacionais oferecem suporte à comunicação em grupo para o nível de aplicação. Com relação a este aspecto, há seis questões básicas que diferenciam os sistemas existentes, conforme ilustra resumidamente a Tabela 4.1, extraída de [Kaashoek-Tanenbaum 94].

Para entender melhor estes critérios de desenho, vejamos, de forma breve, a escolha feita por dois conhecidos sistemas que oferecem suporte para comunicação em grupo.

Sistema ISIS [Birman-Joseph 87]

- Mensagens são endereçadas usando um identificador de grupo ou uma lista de endereços.
- Em caso de falhas, *todos* ou *nenhum* dos membros sobreviventes recebem a mensagem.
- Ordenação total pode ser conseguida.

- O número de resposta que se espera receber é especificado pelo usuário no momento do envio da mensagem.

<u>Sistema Amoeba</u> [Kaashoek-Tanenbaum 94]

- O endereçamento é feito usando um identificador de grupo.
- Entrega confiável na presença de falhas de comunicação e, opcionalmente, de processadores é garantida.
- Entrega de mensagens obedece ordem total dentro de cada grupo.
- Entrega garantida a *todos* os destinos e, opcionalmente, a *todos-ou-nenhum* (quando tolera falhas de processadores)
- Nenhuma primitiva de grupo é disponível para envio de resposta.
- Grupos fechados e dinâmicos.

Questão	**Descrição**
Endereçamento	Método de endereçamento para o grupo (p. ex., lista de membros, endereço de grupo)
Confiabilidade	Comunicação confiável ou não, na presença de falhas
Ordenação	Ordem de entrega de mensagens (p. ex., ordem FIFO, ordem causal, ordem total)
Semântica de entrega	Número de processos que devem receber a mensagem corretamente (*k* processos, maioria de processos ou todos eles)
Semântica de resposta	Número de respostas que o transmissor espera receber (nenhuma, uma, muitas ou todas as respostas)
Estrutura do grupo	Semântica do grupo (p. ex., fechado ou aberto, estático ou dinâmico).

Tabela 4.1: Principais Questões de Desenho para Comunicação em Grupo

Diversos protocolos para comunicação em um-para-muitos têm sido propostos ao longo dos anos [Chang-Maxemchuck 84, Cristian et al. 85, Powell 92]. As principais características dos protocolos adotados pelos dois sistemas supracitados são resumidas a seguir.

[Birman-Joseph 87] descreve o desenho de uma facilidade de comunicação, adotada pelo sistema distribuído ISIS, que provê suporte para *grupos de processos tolerantes a faltas*, na forma de uma família de protocolos para difusão restrita de mensagens. Estes protocolos conseguem altos níveis de concorrência, enquanto respeitam as restrições da ordem de entrega específicas da aplicação, e têm custo e

desempenho variáveis, dependendo do grau de ordenação desejado. Tal facilidade de comunicação também garante que processos pertencentes a um grupo de processos tolerante a faltas observem os eventos afetando o grupo como um todo (p. ex., falhas, recuperações e migrações de processos) numa ordem consistente.

[Kaashoek-Tanenbaum 94] descreve um conjunto de primitivas para *comunicação em grupo* que provêem comunicação um-para-muitos eficiente e atômica na presença de falhas de canais de comunicação e, opcionalmente, de processadores. Dependendo do suporte de *hardware* para comunicação oferecido pela rede, o protocolo proposto pode explorar a capacidade de difusão restrita (preferencialmente) ou difusão irrestrita da rede, ou ainda, na ausência de ambos, usar simplesmente mensagens ponto-a-ponto. As primitivas apresentadas são integradas ao sistema operacional distribuído Amoeba e serão estudadas mais detalhadamente no Capítulo 7 desta dissertação.

4.7. Sumário

A forma mais comum de se obter confiança no funcionamento em aplicações distribuídas é organizá-las como um grupo de processos que cooperam para prover o serviço adequado. Cada processo do grupo é uma réplica da aplicação e, em caso de falha de algumas das réplicas, o restante do grupo é capaz de garantir a continuidade de serviço. A replicação de processamento foi objeto de estudo do capítulo anterior.

Neste capítulo, vimos que as técnicas de replicação dependem de algum mecanismo de comunicação um-para-muitos que garanta a entrega de mensagens para todos os membros do grupo. Além do mais, há muitas situações em que a afiliação do grupo precisa ser conhecida por todos os seus membros e, nestas situações, todos os membros precisam ter uma visão consistente do grupo. Para tanto, qualquer mudança na afiliação do grupo deve ser comunicada a todos os seus membros e, assim, comunicação multiponto é também necessária.

Estas exigências podem ser atendidas por protocolos de comunicação em grupo, que exibem diferentes propriedades para atender aos diferentes requisitos tanto da replicação propriamente dita quanto da gerência de grupos. Além disso, comunicação em grupo provê a transparência necessária para os clientes de aplicação, que não precisam tomar conhecimento da existência do grupo.

As principais propriedades de um protocolo para comunicação em grupo são: *confiabilidade* (a mensagem enviada é entregue a todos os membros livres de falha), *ordenação consistente* (diferentes mensagens enviadas por diferentes transmissores são entregues a todos os membros na mesma ordem) e *preservação da causalidade* (a ordem de entrega das mensagens é consistente com a causalidade dos eventos de envio das mesmas). Estas propriedades podem ser obtidas por meio de diversas formas de *acordo* (unanimidade, no-mínimo-*N* etc.) e *ordenação* (ordem causal, ordem total, ordem FIFO). Diferentes protocolos para comunicação em grupo podem exibir diferentes propriedades.

Algumas questões que diferenciam os sistemas distribuídos quanto ao suporte à comunicação em grupo são: modo de endereçamento do grupo, grau de confiabilidade na presença de falhas, semântica de entrega, semântica de resposta e estrutura do grupo. Dois sistemas (ISIS e Amoeba) foram caracterizados quanto a estas questões e às propriedades da comunicação em grupo por eles oferecida. O Seljuk-Amoeba aproveita o serviço de difusão atômica oferecido pelo Amoeba para prover comunicação confiável e ordenada ao grupo de réplicas que implementam a aplicação robusta.

Capítulo 5:

Diagnóstico da Falta

5.1. Introdução

Como colocado anteriormente, a meta de um sistema tolerante a faltas é mascarar a falha de alguns dos seus componentes para o usuário do sistema, de modo que este sequer tome conhecimento da ocorrência de tal falha. Para tanto, é necessário que a falha de um componente seja detectada e diagnosticada por outros componentes do sistema, o que é conseguido através da realização de testes por cada componente sobre outros componentes. Isto implica que os componentes do sistema precisam realizar algumas atividades, além de suas funções normais, que viabilizem a detecção de falhas em outros componentes.

Uma vez que a falha de um componente do sistema representa uma falta no sistema como um todo, o diagnóstico de faltas no sistema pode ser encarado como a detecção da falha de seus componentes. Podemos, então, dizer que "o objetivo básico do diagnóstico de faltas é identificar todos os componentes apresentando falha no sistema", o que claramente nem sempre é possível. Por exemplo, quando todas os componentes do sistema estão falhos, nenhum componente pode ser utilizado para realizar o diagnóstico e assim atingir a meta idealizada. Um limite no número de componentes falhos no sistema em cada instante do tempo precisa, portanto, ser considerado. A fim de sermos mais realísticos, podemos dizer que "a meta do diagnóstico de faltas em sistemas distribuídos é garantir que se alguns componentes falham (ou se recuperam), então os outros componentes no sistema descobrem esta falha (recuperação) em um tempo finito" [Jalote 94, pp. 116 e 120].

Qualquer algoritmo para diagnóstico de faltas deve idealmente satisfazer as seguintes propriedades [Shin-Ramanathan 87]:

P1. **Corretude** - Todo componente que é diagnosticado pelo algoritmo como sendo falho é de fato falho.

P2. Completude - Todo componente falho no sistema é identificado pelo algoritmo.

Além disso, um serviço de diagnóstico ideal deve permitir que todos os componentes livres de falha do sistema concordem sobre quais componentes são falhos; ou seja, cada componente livre de falha deve chegar a um mesmo diagnóstico do sistema. Quando os componentes do sistema apresentam semântica de falha silenciosa, este objetivo pode ser alcançado de forma razoavelmente simples. Por outro lado, se eles se comportam de forma arbitrária quando em falha, não se conhece nenhum protocolo que possa garantir um diagnóstico completo em todas as situações; neste caso, o máximo que se pode conseguir é um diagnóstico parcial do sistema.

Em geral, não é viável que cada componente realize testes em todos os demais componentes do sistema. Por isso, na maioria dos algoritmos de diagnóstico, cada componente realiza testes em apenas um subconjunto pré-determinado de componentes e recebe destes informações sobre aqueles componentes que ele não testou diretamente.

Na próxima seção, discutiremos aspectos gerais relacionados ao diagnóstico da falta em sistemas distribuídos e introduziremos o conceito de *sistemas t-diagnosticáveis*. Nas seções seguintes, apresentaremos alguns algoritmos de diagnóstico que fazem diferentes suposições sobre o número e a semântica de falhas dos componentes do sistema e que apresentam diferentes custos quanto ao número de testes realizados e ao número de mensagens trocadas durante a execução do algoritmo.

5.2. Diagnóstico de Falta Distribuído

Técnicas para diagnóstico de faltas em sistemas não-distribuídos geralmente confiam em um supervisor centralizado que controla os testes, coleta os resultados e decide sobre o estado do sistema como um todo. Em sistemas verdadeiramente distribuídos, compostos por grande número de componentes geograficamente separados, não faz muito sentido utilizar um elemento central para coordenar as atividades de manipulação de falhas por alguns motivos lógicos, dentre os quais:

(i) a carga de trabalho envolvida nestas atividades é muito alta para ser suportada por um único elemento; e,

(ii) tal elemento central seria um gargalo na confiabilidade de todo o sistema visto que sua falha poderia permitir que componentes falhos continuassem

operando ou, ainda, poderia erroneamente excluir componentes livres de falha.

Por estes e outros motivos, esta abordagem de observador centralizado, utilizada amplamente nos sistemas centralizados, não é adequada para diagnóstico de faltas em sistemas distribuídos. Qualquer mecanismo para detecção de falhas e diagnóstico de faltas neste tipo de sistema provavelmente deveria estar distribuído através da rede. Porém, uma vez que a distribuição da responsabilidade de diagnóstico envolve fluxo de informações de diagnóstico através da rede e que os próprios componentes falhos podem participar deste fluxo, talvez alterando, destruindo ou gerando informações incorretas, o processo do diagnóstico pode se tornar bastante complexo [Hosseini et al. 84].

Alguns autores [Kuhl-Reddy 80, Kuhl-Reddy 81] propuseram uma abordagem para diagnosticar faltas em um sistema distribuído, chamada *tolerância a faltas distribuída*, na qual cada componente do sistema produz o diagnóstico baseado na informação resultante de testes realizados por ele próprio sobre seus vizinhos juntamente com resultados, repassados para tal componente, de testes realizados por outros componentes do sistema. Visto que componentes falhos podem estar envolvidos na execução das atividades de teste e na disseminação de mensagens de diagnóstico, a informação repassada para o componente pode não ser inteiramente correta ou confiável. Por isso, uma vez que uma unidade livre de falha detecta a falha de uma outra unidade do sistema (i.e., diagnostica uma falta), ela deve desprezar as informações derivadas daquela unidade falha. Se todos os componentes livres de falha agem desta maneira, todos os componentes falhos são isolados lógica e efetivamente do sistema.

Diagnóstico a nível de sistemas foi introduzido por Preparata, Metze e Chien em [Preparata et al. 67], que definiram o primeiro modelo de sistemas diagnosticáveis, conhecido como Modelo PMC. A sigla PMC é derivada dos próprios nomes dos criadores do modelo, que tem servido de base para muitos algoritmos para diagnóstico distribuído. Naquele mesmo trabalho, foi também definido o conceito de sistemas t-diagnosticáveis:

Definição. Seja a *síndrome* de um sistema S o conjunto dos resultados de todos os testes realizados pelos componentes de *S*. O sistema *S* é dito ser *t-diagnosticável* se, dada uma síndrome, todos os componentes apresentando falha podem ser identificados, desde que o número de componentes que podem falhar não exceda o limite *t*.

[Preparata et al. 67] apresenta provas de que, se um sistema é t-diagnosticável, então cada componente do sistema deve ser testado por no mínimo t outros componentes para que um diagnóstico correto possa ser alcançado.

5.3. Diagnóstico de Falta para Sistemas Distribuídos com Falha e Reparo Dinâmicos

Em [Hosseini et al. 84] foi proposto o primeiro algoritmo totalmente distribuído para autodiagnóstico de sistemas. Tal algoritmo, denominado NEW_SELF, permite que *todas* as unidades de processamento livres de falha diagnostiquem independente e corretamente o estado de todas as demais unidades de processamento e canais de comunicação do sistema, com base nos resultados dos seus testes e nos resultados recebidos de seus vizinhos (i.e., daquelas unidades que estão conectadas diretamente). Para tanto, assume-se que as unidades possuem a habilidade de testar outros componentes (unidades de processamento e canais de comunicação) quanto a presença de falhas e de repassar os resultados de diagnóstico para seus vizinhos e, através destes, alcançar sucessivamente outras unidades no sistema. O esquema permite ainda o retorno de componentes anteriormente falhos e que tenham sido recuperados, bem como a introdução no sistema de componentes inteiramente novos.

O algoritmo proposto funciona corretamente desde que: (1) o número de unidades de processamento falhas no sistema não exceda um dado limite t, tal que $N \geq 2t + 1$, onde N é o número total de unidades de processamento no sistema; (2) haja um conjunto fixo de testes; e (3) cada unidade seja responsável por testar um subconjunto definido dos seus vizinhos.

5.3.1. Modelo do Sistema

No modelo empregado pelo Algoritmo NEW_SELF, o sistema distribuído é composto por um conjunto de *nodos* (i.e., unidades de processamento) e canais de comunicação entre os nodos. As seguintes suposições são feitas: (1) nodos podem apresentar comportamento arbitrário, mas não malicioso, quando em falha; (2) um nodo não pode falhar e depois se recuperar sem que sua falha seja detectada por outros nodos; (3) um nodo é ca

paz de testar alguns dos seus vizinhos quanto à operação correta (ou à presença de falhas); (4) nodos livres de falha realizam os testes confiavelmente e sempre chegam a uma conclusão correta a respeito do estado dos nodos que ele testa; e (5) os nodos com falha podem realizar os testes incorretamente e podem apresentar conclusões inválidas. Esta última suposição implica que o diagnóstico gerado por nodos diagnosticados como falhos deve ser considerado não-confiável.

Mensagens de diagnóstico, que contêm informação relativa aos resultados dos testes, podem fluir entre os nodos e alcançar nodos distantes através de um ou mais nodos intermediários. A suposição sobre a semântica de falha dos nodos é que eles podem falhar de forma arbitrária, porém não maliciosa. Isto implica que uma mensagem, inclusive de diagnóstico, passando através de um nodo com falha pode ser alterada ou destruída e que um nodo com falha pode emitir mensagens errôneas, mas ele não pode agir maliciosamente, enviando diferentes informações de diagnóstico para diferentes destinos.

O modelo de falta também permite a ocorrência de falhas na rede de comunicação, assumindo que as práticas de comunicação comumente usadas (p. ex., codificação de mensagens com códigos de detecção de erro) são suficientes para permitir a detecção de falhas afetando os canais de comunicação. Ou seja, é assumido que qualquer falha de um canal internodos se manifesta de uma maneira detectável pelos nodos fazendo uso daquele canal.

5.3.2. Algoritmo NEW_SELF

A principal dificuldade em se conseguir diagnóstico distribuído em ambientes cujos componentes falhos podem se comportar arbitrariamente é que as mensagens de diagnóstico podem ser alteradas, destruídas ou erroneamente emitidas por tais componentes. Assim, um nodo não pode acreditar que as mensagens de diagnóstico que chegam até ele sejam completas ou corretas. O Algoritmo NEW_SELF trata este problema por restringir o fluxo de informação de uma maneira tal que garanta sua confiabilidade. Especificamente, um nodo P_i aceitaria uma mensagem de diagnóstico de um nodo vizinho P_j apenas se P_i é um testador de P_j e está certo de que P_j é livre de falha. Isto pode ser conseguido por um mecanismo simples, cujo funcionamento é apresentado logo adiante. Antes disto, transcreveremos informalmente parte da notação

empregada pelo algoritmo com o objetivo de facilitar o entendimento do mecanismo [Hosseini et al. 84]:

TESTED_BY(P_v) = Conjunto de nodos no sistema que são testados pelo nodo P_v.

TESTERS_OF(P_v) = Conjunto de nodos no sistema que são testadores do nodo P_v.

O mecanismo supracitado funciona da seguinte forma: quando um nodo $P_j \in$ TESTED_BY(P_i) envia uma mensagem de diagnóstico para P_i, P_i armazena temporariamente esta mensagem em um *buffer* caso ele tenha diagnosticado P_j como sendo livre de falha no último teste realizado. Da próxima vez que P_i testa P_j, se P_i conclui que P_j continua livre de falha, ele marca todas as mensagens recebidas de P_j no intervalo entre os dois testes como mensagens válidas a serem usadas para produzir o diagnóstico do sistema e a serem repassadas para outros nodos. Isto requer, porém, que uma falha temporária de P_j, ocorrendo no intervalo entre os dois testes, possa ser detectada. Uma forma de conseguir isto é construir nodos autotestáveis que, quando consultados durante um teste, possam indicar a ocorrência de falha naquele intervalo, ou então que, ao requerer readmissão na rede após uma falha, notifique todos os seus testadores da ocorrência da falha.

A possibilidade de readmissão de componentes anteriormente falhos ou de admissão de novos componentes ao sistema adiciona considerável complexidade ao processo de diagnóstico, devido à necessidade de se controlar a ordem de certas mensagens de diagnóstico fluindo pela rede. Por exemplo, suponha que um nodo *Pi* falha e se recupera, mas volta a falhar novamente. As mensagens de diagnóstico indicando estas ocorrências podem chegar a um nodo distante P_j em várias ordens. Se a notícia da última falha alcança P_j antes da indicação da recuperação da falha anterior, P_j não deve interpretar esta mensagem de recuperação como indicando a recuperação da última falha, mas sim da primeira. Para lidar com este problema são usados esquemas para ordenação de mensagens baseados em relógios locais ou *timestamps*, semelhantes àqueles discutidos no capítulo anterior.

Há várias ocasiões em que um nodo P_i testa um vizinho $P_j \in$ TESTED_BY(P_i):

(i) em intervalos de tempo periódicos de acordo com algum escalonamento determinado localmente;

(ii) em resposta a alguma situação suspeita relacionada a P_j (por exemplo, não ter recebido notícias de P_j durante um longo tempo); ou ainda,

(iii) em resposta a um pedido para entrada de P_j ao sistema, onde P_j pode ser um novo nodo, um nodo substituto ou um nodo recuperado de uma falha anterior.

Após descobrir o estado de P_j, P_i difunde uma mensagem indicando tal estado para todos os nodos em TESTERS_OF(P_i) via um protocolo de difusão restrita confiável. Se P_i conclui que P_j é livre de falha e P_i encontra em seu *buffer* algum registro emitido por P_k indicando a falha de P_j, P_i remove tal registro e anuncia a seus testadores que P_j se recuperou daquela falha detectada por P_k. Ao receber e validar (por meio do mecanismo descrito anteriormente) qualquer mensagem indicando que P_j é livre de falha, um nodo P_m remove do seu *buffer* qualquer mensagem contrária (emitida em um tempo anterior à mensagem mais recente) e repassa a informação recebida para todos os seus testadores, garantindo assim sucessivamente a disseminação através de toda a rede.

Critérios especiais, baseados nos *timestamps* das mensagens, são adotados para evitar que os *buffers* mantendo informações sobre os testes realizados por outros nodos cresçam de um maneira incontrolada, bem como para decidir se uma mensagem recebida é ainda válida e se deve ou não ser disseminada para os testadores do nodo; isto garante um tempo de vida finito para as mensagens. Por exemplo, quando uma mensagem indicando a falha de P_j chega a P_m, este verifica se há em seu *buffer* algum registro de recuperação de P_j com *timestamp* maior do que o da mensagem de falha. Se não há, *Pm* adiciona a mensagem a seu *buffer* e a distribui para todos os seus testadores. Se há, isto indica que o nodo já se recuperou e aquela mensagem de falha não é mais válida, devendo ser simplesmente ignorada (como foi dito acima, uma mensagem indicando a recuperação de P_j pode chegar a um nodo antes da mensagem comunicando à falha anterior a esta recuperação).

Quando um nodo, novo ou recuperado, deseja se juntar à rede, ele emite um pedido de entrada para seus testadores e inicializa todas as suas tabelas para vazio e o seu relógio local para zero. Porém, seu relógio precisa estar aproximadamente sincronizado com os relógios dos demais nodos para que suas mensagens sejam consideradas válidas e não sejam descartadas. Um mecanismo simples de se conseguir sincronização entre todos os relógios da rede é fazer com que, cada vez que um nodo

recebe uma mensagem com *timestamp* maior do que o valor do seu próprio relógio, ele atualize o seu relógio para o valor contido na mensagem.

O Algoritmo NEW_SELF mantém, em cada nodo P_i, dois *buffers* para mensagens de diagnóstico: o ACCUSERS$_i$, contendo informações sobre nodos detectados como em estado de falha, e o ENTRY$_i$, que informa os nodos que se apresentaram livres de falha durante os testes. O *buffer* ENTRY$_i$ mantém mensagens da forma $[P_j(t_j), P_m(t_m), P_k]$, indicando que um nodo P_j testou P_k no tempo t_j e encontrou P_k livre de falha. Se $m \neq j$, isto significa que P_m havia testado P_k no tempo t_m, anterior a t_j, e havia diagnosticado P_k como falho; assim, uma mensagem com $m \neq j$ indica que P_j descobriu que P_k se recuperou da falha diagnosticada por P_m.

Devido à ocorrência de falhas e reparos de nodos e ao atraso dos canais de comunicação, é impossível manter os conteúdos destes *buffers* constantemente atualizados e, conseqüentemente, as informações neles contidas não podem ser diretamente usadas no momento de P_i realizar o diagnóstico final do sistema. Faz-se, portanto, necessária a realização de uma computação simples no *buffer* ENTRY$_i$. Primeiro, P_i deve se certificar da condição dos nodos que ele testa diretamente. Em seguida, para todo nodo $P_j \in$ TESTED_BY(P_i) que P_i encontrou ser livre de falha, se há qualquer mensagem do tipo $[P_j(t_j), P_j(t_j), P_k]$ ou $[P_j(t_j), P_m(t_m), P_k]$ no buffer ENTRY$_i$, então P_k é efetivamente considerado livre de falha. De forma similar, para qualquer mensagem $[P_k(t_k), P_k(t_k), P_s]$ ou $[P_k(t_k), P_r(t_r), P_s]$, P_s é considerado livre de falha e assim sucessivamente.

O procedimento também permite o diagnóstico de falhas em canais de comunicação de duas formas. Primeiro, a falha de um teste realizado por um nodo P_i sobre um nodo P_j pode ocorrer devido a uma falha do nodo, do canal de comunicação ou de ambos. Então, uma mensagem indicando uma falha deve ser interpretada pelo nodo receptor como um indicativo de qualquer uma destas três situações. Segundo, se um nodo P_i não consegue comunicação normal (não relacionada a testes) com um vizinho P_j, novamente devido a qualquer um dos motivos acima, P_i pode emitir uma mensagem indicando este fato para a rede. Se P_i é um testador de P_j, a mensagem enviada é da forma $[nil, P_i(t_i), P_j]$ e informa simplesmente que o teste falhou; caso contrário, a mensagem informa que o canal de comunicação falhou (embora nem sempre seja verdade) e possui o formato $[link, P_i(t_i), P_j]$. Um nodo P_m recebendo uma mensagem

[nil, $P_i(t_i)$, P_j] inicialmente a trata como um indicativo de falha de P_j e a armazena no *buffer* ACCUSERS$_m$. Se P_j é livre de falha e a falha, na verdade, é do canal de comunicação entre P_i e P_j, então P_m eventualmente recebe uma mensagem do tipo [$P_k(t_k)$, $P_i(t_i)$, P_j], que seria gerada quando a mensagem [nil, $P_i(t_i)$, P_j] alcançasse $P_k \in$ TESTERS_OF(P_j), como explicado anteriormente. Ao receber aquela mensagem, P_m remove a mensagem de falha do *buffer* ACCUSERS$_m$ e a adiciona ao *buffer* FAULTY_LINKS$_m$ indicando que o canal entre P_i e P_j apresentou uma falha. No evento de reparo do canal, esta mensagem é removida do *buffer* de P_m logo que este toma conhecimento do fato.

Como evidenciado pela descrição acima, os resultados de testes são disseminados entre os nodos livres de falha na direção reversa em que os testes são realizados. Em suma, o seguinte esquema de teste e validação de registros é utilizado por cada nodo i sobre cada um dos seus vizinhos j:

1. i testa j como livre de falha;
2. i recebe resultados de testes de j;
3. i testa j como livre de falha;
4. i assume a informação de diagnóstico recebida no passo 2 como sendo válida

O algoritmo completo é apresentado em [Hosseini et al. 80], bem como provas de que, usando o processo acima, todos os nodos livres de falha convergem para um diagnóstico correto de todas as faltas no sistema desde que não mais do que t componentes (nodos ou canais de comunicação) falhem simultaneamente.

5.3.3. Análise do Algoritmo NEW_SELF

Para alcançar um diagnóstico correto usando o Algoritmo NEW_SELF, cada nodo deve receber registros de testes relativos a todos os outros nodos do sistema. Esta condição é satisfeita se cada nodo é testado por no mínimo t + 1 outros nodos. O número de mensagens requeridas para transferir os resultados de testes é $N^2(t+1)^2$, onde N é o número de nodos do sistema.

O Algoritmo NEW_SELF foi estendido em [Bianchini et al. 90], originando o Algoritmo EVENT_SELF, que leva em consideração as limitações de recursos dos sistemas distribuídos atuais. Neste algoritmo, os resultados de testes só são repassados

de um nodo para outro se eles diferem dos resultados anteriores já armazenados no nodo. Desta maneira, apenas registros que representam um novo evento de falha ou recuperação (mudança de estado) no sistema seriam repassados. No estado de funcionamento normal do sistema, onde todos os nodos permanecem livres de falha, os testes seriam realizados, mas nenhum resultado seria disseminado através da rede. Este procedimento reduz consideravelmente o número de mensagens requeridas pelo algoritmo para se chegar ao diagnóstico. Tal número é reduzido para ΔfNt^2, onde Δf é a mudança no número de faltas.

5.4. Diagnóstico Adaptativo a Nível de Sistema Distribuído

Para um sistema com N nodos e capacidade de diagnóstico t, o número de mensagens requeridas para transferir os resultados de testes no Algoritmo NEW_SELF é $N^2(t+1)^2$, o que pode ser muito significante mesmo para sistemas distribuídos pequenos. Por exemplo, uma rede de $N = 8$ nodos e capacidade de diagnóstico $t = 2$ requer a troca de 576 mensagens em cada ciclo de testes.

Além disso, tanto o NEW_SELF quanto sua extensão, o EVENT_SELF, apresentam duas desvantagens significantes [Bianchini-Buskens 92]. A primeira delas diz respeito à capacidade de diagnóstico limitada: qualquer algoritmo não-adaptativo só é capaz de diagnosticar um número limitado de faltas. Os algoritmos em questão funcionam corretamente apenas se o número de nodos falhos no sistema não excede t, tal que $t < N/2$.

A segunda desvantagem relaciona-se com a redundância tanto em termos de testes quanto de disseminação dos seus resultados. Para sistemas t-diagnosticáveis, cada nodo deve ser testado por no mínimo $t + 1$ outros nodos. Na realidade, porém, para se garantir diagnóstico correto, é suficiente que cada nodo seja testado apenas por um nodo livre de falha. Sendo assim, todos os demais t testes realizados pelos algoritmos SELF são redundantes. Da mesma forma, também são redundantes os resultados gerados por estes testes e disseminados através da rede por vários caminhos diferentes.

Mais recentemente, um algoritmo denominado *DSD Adaptativo* (Diagnóstico de Sistemas Distribuídos Adaptativo) foi proposto e implementado para sistemas

distribuídos grandes, visando superar as desvantagens supracitadas [Bianchini-Buskens 91, Bianchini-Buskens 92], como veremos a seguir.

5.4.1. Modelo do Sistema

O Algoritmo DSD Adaptativo assume uma rede de nodos distribuídos, onde a cada nodo é atribuído um *estado de falta*: *livre de falha* ou *com falha* (lembremos mais uma vez que a *falha* de um nodo representa uma *falta* no sistema). Falhas dos canais de comunicação não são consideradas. Nodos realizam testes sobre outros nodos. A *síndrome* da rede é o conjunto de todos os resultados de testes. A *situação de falta* da rede é o conjunto de todos os estados de falta dos nodos. *Diagnóstico* é a determinação da situação de falta de uma rede dada sua síndrome.

Testes são realizados por cada nodo adaptativamente e são determinados pela situação de falta da rede. Cada nodo, além dos resultados dos seus próprios testes, usa os resultados de testes realizados pelos outros nodos do sistema. Os resultados de testes realizados por nodos livres de falha são assumidos corretos; já os testes realizados por nodos falhos não são confiáveis, podendo apresentar resultados arbitrários. Por isto, um nodo só aceita e usa informação de teste de outro nodo se ele tem certeza de que tal nodo é livre de falha.

5.4.2. Algoritmo DSD Adaptativo

O elemento principal do algoritmo é o vetor TESTED_UP_i, que contém N elementos, indexados pelo número do nodo, onde N é o número de nodos do sistema. Cada elemento de TESTED_UP_i contém um número de nodo. A entrada $\text{TESTED_UP}_i[k] = j$ significa que o nodo i recebeu a informação de diagnóstico de um nodo livre de falha especificando que o nodo k atestou j como sendo livre de falha. Se k não é livre de falha, a entrada correspondente é indefinida e não interessa para a realização do diagnóstico.

Em cada nodo, o Algoritmo DSD Adaptativo primeiro identifica um nodo livre de falha e então atualiza sua informação de diagnóstico local (TESTED_UP) a partir da informação recebida daquele nodo. Isto é conseguido da seguinte forma: os nodos são ordenados em uma lista circular (0, 1, 2, ..., n -1, 0). Cada nodo i testa seqüencialmente os seus sucessores na lista (i + 1, i + 2 etc.) até que ele encontre um nodo livre de falha.

A informação de diagnóstico recebida deste nodo é então assumida ser válida e é usada por i para atualizar TESTED_UPi. O algoritmo para o nodo i é mostrado na Figura 5.1, extraída de [Bianchini-Buskens 92].

```
1. t = i
2. repeat
3.        t = (t + 1) mod n
4.        pede a t para enviar TESTED_UPt para i
5. until (i testa t e conclui que t é "livre de falha")
6. TESTED_UPi[i] = t
7. for j = 0 to n-1 do
8.        if ( j ≠ i )
9.                TESTED_UPi[j] = TESTED_UPt[j]
```

Figura 5.1: O Algoritmo DSD Adaptativo

O algoritmo é executado em cada nodo i ($0 \leq i \leq n$-1) do sistema em intervalos de tempo pré-definidos. Os passos de 1 a 5 identificam o primeiro nodo livre de falha depois de i na seqüência da lista ordenada. Um nodo t é avaliado "livre de falha" no passo 5 se t permaneceu livre de falha desde a última vez que ele foi testado por i, incluindo o período requerido para transferência de TESTED_UP$_t$ para i. Isto garante que a informação de diagnóstico incluída em TESTED_UP$_t$ é correta. Quando um nodo i encontra um nodo t livre de falha, ele salva esta informação em TESTED_UP$_i[i]$ (passo 6) e atualiza sua informação de diagnóstico local com a informação de diagnóstico recebida de t (passos de 7 a 9).

No próximo ciclo de testes, a informação de que t é livre de falha, juntamente com a informação de diagnóstico repassada de t para i, é tomada pelo primeiro predecessor de i livre de falha na seqüência de nodos da lista circular (digamos j), que, por sua vez, acerta o valor de TESTED_UP$_j[j]$ para i e o de TESTED_UP$_j[i]$ para t. No próximo ciclo, esta informação alcançará o primeiro predecessor livre de falha de j e assim por

diante. Desta forma, a informação de diagnóstico se propaga por todo o sistema na direção reversa dos testes. O teste usado na implementação do Algoritmo DSD Adaptativo é descrito em [Bianchini-Buskens 91]. Um nodo é atestado como falho se ele não consegue executar corretamente as ações definidas pelo teste. Nenhuma outra suposição é feita sobre o comportamento de falha dos nodos.

O vetor TESTED_UP mantém informação sobre o estado do sistema de uma maneira indireta, devendo portanto ser processado sempre que um nodo *i* deseja obter o diagnóstico do sistema inteiro. Este processamento é feito pelo Algoritmo de Diagnóstico mostrado na Figura 5.2, extraída de [Bianchini-Buskens 92].

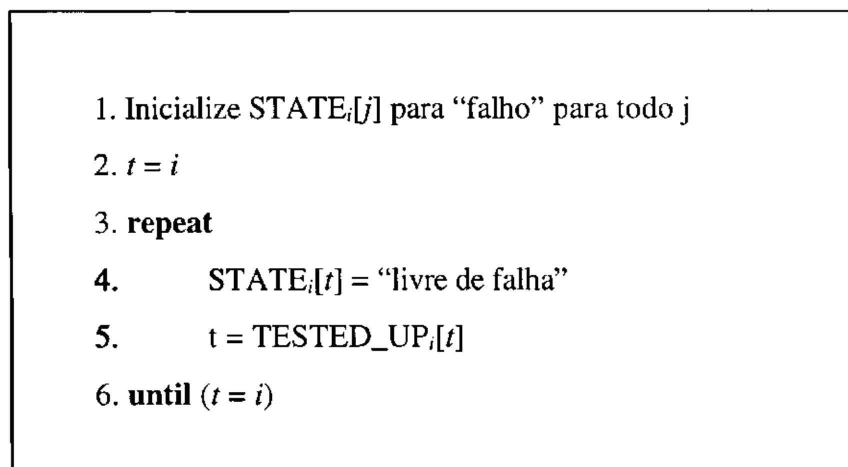


Figura 5.2: O Algoritmo de Diagnóstico

Para um nodo *i*, o algoritmo usa a informação em $TESTED_UP_i$ para determinar o estado do sistema, que é armazenado no vetor $STATE_i$. O *k*-ésimo elemento de $STATE_i$ representa o estado do nodo *k* conforme determinado pelo nodo *i*. Antes de mais nada, o estado de todos os nodos é inicializado para "falho". No passo 4 da primeira interação do *repeat*, o próprio estado de *i* é estabelecido ser "livre de falha". Durante a segunda interação, o algoritmo define como "livre de falha" o estado do nodo que i diretamente testou (digamos *j*) e determinou ser livre de falha. Na próxima interação, o estado do nodo que *j* testou e concluiu ser livre de falha é estabelecido. Isto continua até que se alcance o nodo que determinou *i* como livre de falha, quando então o laço se completa e o algoritmo termina.

5.4.3. Análise do Algoritmo Adaptativo

Uma análise mais apurada do algoritmo demonstra que, depois de no máximo *N* ciclos de execução, os resultados de testes do primeiro ciclo alcançam todos os nodos livres de falha do sistema, onde *N* é o número total de nodos no sistema. Uma suposição implícita do algoritmo é que nenhuma mudança (falha ou reparo) ocorre durante os *N* ciclos. Em outras palavras, o algoritmo adaptativo é tal que depois do estado do sistema mudar, o sistema se estabiliza depois de no máximo *N* ciclos, após o que cada nodo pode diagnosticar o estado correto do sistema. Isto é, há um *período de convergência* seguindo uma mudança no estado do sistema, durante o qual um diagnóstico correto de todos os nodos não é garantido. Isto é aceitável se falhas e recuperações não ocorrem com muita freqüência [Jalote 94, pp. 125].

Embora o Algoritmo DSD Adaptativo seja ótimo em termos de número de testes (cada nodo é testado por um único nodo livre de falha), ele requer mais do que o número mínimo necessário de mensagens de diagnóstico, visto que cada nodo gera e repassa um registro de teste para cada resultado que ele obtém. Para nodos cujos estados permanecem inalterados ao longo de vários ciclos, registros duplicados seriam gerados e propagados desnecessariamente. Uma forma de evitar este custo extra é garantir que apenas informações de diagnóstico novas sejam disseminadas, o que pode ser conseguido empregando-se um esquema de *timestamp* tal como proposto em [Bianchini et al. 90].

Provas formais da corretude desta abordagem e registros de experiências são dadas em [Bianchini-Buskens 91, Bianchini-Buskens 92]. Para o âmbito do nosso trabalho basta dizer que, usando os dois algoritmos apresentados na subseção anterior, todos os nodos livres de falha capturariam o estado correto de todos os outros nodos do sistema distribuído e chegariam a um mesmo diagnóstico do sistema.

5.5. Diagnóstico de Nodos com Semântica de Falha Bizantina

Em termos de diagnóstico, um nodo em um sistema distribuído é dito apresentar uma falha Bizantina ou comportar-se maliciosamente se ele tenta esconder sua falha enviando diferentes informações sobre a mesma entidade para diferentes partes do sistema [Shin-Ramanathan 87]. Na literatura atual, há registro de vários algoritmos que podem ser usados para *mascarar* falhas Bizantinas [Lamport et al. 82, Dolev-Strong 83],

porém poucos tratam do problema de *identificar* corretamente os nodos apresentando este tipo de falha. A primeira solução para este problema foi dada em [Shin-Ramanathan 87], onde foi proposto um algoritmo *off-line* que pode seguramente identificar qualquer nodo[8] malicioso que exiba este comportamento mais do que M vezes, onde M é o número máximo de faltas a serem toleradas no sistema. Entretanto, na maioria das situações, não é necessário que um nodo exiba seu comportamento malicioso $M + 1$ para que ele seja diagnosticado como tal.

Visto que todos os nodos maliciosos eventualmente exibirão seu comportamento errôneo enviando diferentes informações para diferentes partes do sistema, é natural localizar tais nodos através da observação de todas as mensagens trocadas entre os nodos durante um período de tempo suficientemente longo. Cada nodo observa o comportamento dos outros nodos por meio das mensagens que ele recebeu durante a missão e registra todo comportamento que lhe parecer inconsistente. O usuário pode, então, analisar os registros obtidos de cada nodo e localizar os nodos falhos no final de cada missão, ou ainda (e preferencialmente) os próprios nodos podem trocar alguma informação adicional e identificar os nodos maliciosos durante a missão. Este processo, entretanto, torna-se complicado pelo fato de um nodo malicioso poder mentir até mesmo ao registrar informação sobre outros nodos, tornando impossível confiar completamente em qualquer registro.

O algoritmo proposto em [Shin-Ramanathan 87], batizado de Algoritmo D, coleta os resultados dos testes - a *síndrome* do sistema - observando as mensagens trocadas pelo nodos, identifica os nodos suspeitos, distribui esta informação para vários nodos e analisa esta informação com base em todas as síndromes anteriores.

Neste algoritmo, é solicitado aos nodos para trocarem entre si um valor pré-determinado e executarem repetidamente um algoritmo de consistência interativa autenticado - AICA [Lamport et al. 82, Dolev-Strong 83] a fim de alcançarem um acordo no valor privado de cada nodo (vide Seção 4.6.1).

No final de cada execução do algoritmo de acordo, cada nodo examina as mensagens por ele recebidas na execução corrente e identifica um conjunto de nodos suspeitos. Este conjunto é levado ao conhecimento dos outros nodos do sistema, que

[8] No texto original, o nome *processador* foi usado em lugar de *nodo*; aqui, preferimos este último a fim de manter consistência com os algoritmos apresentados anteriormente.

usam esta informação juntamente com a informação coletada no passado para identificar os nodos falhos.

Nas subseções seguintes apresentaremos o modelo básico do sistema e o Algoritmo D, conforme definidos em [Shin-Ramanathan 87].

5.5.1. Modelo Básico

Do mesmo modo que os algoritmos de consistência interativa baseados em autenticação, o Algoritmo D faz as seguintes suposições sobre o sistema de troca de mensagens a fim de prevenir que um nodo malicioso possa interferir na comunicação entre dois nodos livre de falha:

A1. Toda mensagem enviada por um nodo livre de falha é entregue corretamente.

A2. O receptor da mensagem sabe quem a enviou.

A3. A ausência de uma mensagem pode ser detectada.

A4. Todo nodo adiciona sua própria assinatura a toda mensagem que ele envia. Esta assinatura digital contém a mensagem codificada de tal forma que nenhum nodo possa forjar a assinatura de qualquer outro nodo.

Implementações práticas para estas suposições são dadas em [Lamport et al. 82, Dolev-Strong 83]. Um algoritmo clássico para obtenção de assinaturas digitais é apresentado em [Rivest et al. 78].

Visto que cada execução de AICA consiste de muitos ciclos de troca de mensagens, as mensagens trocadas podem ser usadas para diagnosticar os nodos falhos. Se todos os nodos no sistema são livres de falha, então todas as mensagens trocadas entre eles numa dada execução do algoritmo de consistência interativa têm o mesmo valor. Entretanto, se alguns dos nodos são maliciosos, valores diferentes também poderão ser trocados. Esta divergência de valores é a única fonte de informação disponível para detectar e localizar os nodos maliciosos. Cada nodo compara o valor que ele recebeu durante a execução do algoritmo de acordo com o valor privado do transmissor e registra todos os valores que diferem do valor computado. Baseado nestes dados, o algoritmo de diagnóstico pode identificar os nodos maliciosos do sistema. Tal algoritmo é apresentado na próxima subseção, seguindo a descrição dada em [Shin-Ramanathan 87].

5.5.2. Algoritmo de Diagnóstico Parcial de Faltas Bizantinas

Observando as mensagens trocadas por todos os nodos na execução do algoritmo de consistência interativa, é possível identificar um conjunto de nodos, CS_i, que acusaram i de ser malicioso no final da execução corrente. Seja o conjunto CS_i obtido no final da (n + 1)-ésima execução do AICA denotado por CS_i^{n+1}. Se $|CS_i^{n+1}| > m$, onde m é número máximo de faltas a serem toleradas, então i é malicioso já que há pelo menos um nodo livre de falha que o está acusando de ser malicioso. Se $|CS_i^{n+1}| \leq m$, então é possível que um conjunto de nodos maliciosos tenham conspirado contra o nodo livre de falha i e tenham acusado i de ser malicioso. Assim, não é possível afirmar imediatamente se o nodo acusado é ou não malicioso. Entretanto, pode-se usar alguma informação anterior sobre alguns nodos para esclarecer esta dúvida e chegar a uma conclusão definitiva. Esta informação é representada pelo conjunto de nodos CF_i que são necessariamente maliciosos se i é livre de falha no final da n-ésima execução de AICA, ou seja, aqueles nodos que acusaram i de ser malicioso.

Inicialmente, o conjunto CF_i^n é vazio para todo i. Nodos são adicionados ao conjunto sempre que i ou é acusado ou acusa algum nodo de ser malicioso. Dessa forma, a qualquer instante de tempo, se $|CF_i^n| > m$, para algum i e n, então i é definitivamente malicioso, pois há no conjunto algum nodo j que é livre de falha e que acusou i de ser malicioso; ou ainda, o nodo j foi acusado por i de ser malicioso e, como j é livre de falha, então i é malicioso. Em algumas situações, pode ainda ser possível diagnosticar i como sendo malicioso mesmo se $|CF_i^n| \leq m$. O algoritmo para diagnosticar os nodos maliciosos em tais situações é descrito abaixo.

Seja $M_i^{n+1,l}$ o conjunto de todas as mensagens recebidas por i no l-ésimo ciclo da (n+1)-ésima execução de AICA. Cada mensagem compreende um valor e uma seqüência de assinaturas dos nodos que já viram a mensagem. Devido as suposições A1-A4 apresentadas anteriormente, nodos livres de falha são capazes de detectar mensagens impróprias, ou seja, mensagens adulteradas ou com assinaturas forjadas, ausência de mensagens etc.

Seja $IM_i^{n+1,l}$ o conjunto de todas as mensagens impróprias recebidas por i no l-ésimo ciclo da (n+1)-ésima execução de AICA. Seja cada mensagem denotada por *val:sigs*, onde *val* é o valor da mensagem e *sigs* é a seqüência de assinaturas presentes

naquela mensagem. Sejam V_{mess} e S_{mess}, respectivamente, o valor de uma mensagem particular, $mess \in M_i^{n+1,l}$, e a assinatura do nodo que enviou esta mensagem para i.

Seja v_i^{n+1} o valor privado do transmissor como computado por um nodo livre de falha i no final da (n+1)-ésima execução de AICA. Se v_i^{n+1} = *default*, o transmissor é definitivamente malicioso. Uma vez que é difícil, neste caso, identificar qualquer outro nodo falho, o algoritmo descrito a seguir só é executado se $v_i^{n+1} \neq$ *default*.

No primeiro passo do algoritmo, cada nodo i examina as mensagens por ele recebidas durante a execução corrente de AICA e identifica todas as mensagens impróprias ou com valor diferente daquele computado para o transmissor, associando ainda a cada uma delas a identidade do nodo do qual ele recebeu aquela mensagem. O nodo i coleta estas informações em um conjunto T_i, que é codificado e enviado para outros nodos usando uma outra execução de AICA. Ao receber T_i de i, o nodo j forma um conjunto T contendo a união de todos os T_i's, para cada nodo i do sistema. Visto que cada nodo usa um AICA para disseminar seu T_i, todos os nodos livres de falha terão um conjunto T idêntico neste estágio.

Ao mesmo tempo em que T possui informação correta sobre nodos maliciosos, ele também contém informação incorreta. Por exemplo, suponha que um nodo malicioso q envia uma mensagem incorreta para um nodo livre de falha p. O nodo p então registraria este fato em T_p. Porém, o nodo q, como é malicioso, poderia registrar em T_q uma informação incorreta sobre p, acusando-o de ser malicioso. Assim, o conjunto T teria registros antagônicos (p acusando q e vice-versa) e, logicamente, um deles é incorreto. Nesta situação, p é dito ser *co-suspeito* de q (i.e., p acusa q de ser malicioso) e vice-versa. Através de uma varredura em T é possível identificar todos os co-suspeitos (i.e., de cada nodo i que possui algum co-suspeito), gerando os conjuntos CS_i's.

Esta informação é então usada, juntamente com a informação similar obtida em execuções anteriores de AICA, para diagnosticar os nodos maliciosos: os CF's são atualizados, adicionando-se a cada conjunto CF_i existente, o CS_i mais recente bem como a identificação dos nodos acusados por i de serem maliciosos (i.e., os nodos dos quais i é co-suspeito). O conjunto de nodos realmente maliciosos é finalmente identificado e é composto por todo nodo i tal que $|CF_i^{n+1}| > m$. Depois disto, uma fase de recuperação é executada, onde os nodos falhos são substituídos por nodos livres de falha e os CF's são atualizados para refletirem o novo estado do sistema.

de trabalho (isto é, as réplicas executando no componente falho) deve ser redistribuída entre os recursos remanescentes, o que conduz à degradação do desempenho do sistema como um todo. Uma outra alternativa é simplesmente abandonar o processamento das réplicas que estavam sendo executadas no componente que falhou. Neste caso, o grau de redundância é reduzido, afetando a capacidade de tolerância a faltas subseqüentes. Esta alternativa pode, porém, ser selecionada sem maiores implicações quando: (1) há outras réplicas daquela mesma aplicação executando em componentes livres de falha; e (2) o tempo de missão da aplicação está perto de se esgotar, havendo pouca probabilidade de ocorrerem todas as faltas que o sistema ainda é capaz de tolerar.

6.2. Conceito de Unidades de Substituição

Em [Cristian 91], uma *unidade de hardware substituível* (*replaceable hardware* unit) é definida como uma unidade física de falha, substituição e crescimento - isto é, uma unidade que falha independentemente das outras unidades, que pode ser removida do sistema sem afetar as demais unidades e que pode ser adicionada ao sistema para aumentar seu desempenho, capacidade ou disponibilidade.

A meta primordial por trás do conceito de unidades de *hardware* substituíveis é habilitá-las a serem fisicamente removidas do sistema (ou por causa de uma falha ou para manutenção preventiva) e reinseridas no sistema sem interromper a atividade das aplicações rodando em níveis de abstração mais altos. Isto, porém, é normalmente muito caro ou mesmo impossível de se conseguir. A próxima meta seria, então, garantir que o serviço fornecido pelos servidores de *hardware* em cada unidade de substituição tenha um semântica de falha controlada, tal como silenciosa ou omissão, de modo que níveis mais altos de abstração de *software* possam prover mascaramento de falha do *hardware* com sobrecarga bem mais baixa do que a necessária para tratar uma semântica de falha mais fraca [Cristian 91]. O conceito de nodos, conforme definido na Seção 1.1.2, satisfaz esta meta.

Servidores de *software*, por sua vez, são as unidades básicas de falha, substituição e crescimento de *software* e têm metas semelhantes às unidades de *hardware* substituíveis. Uma unidade de *software* substituível nada mais é do que uma réplica de um componente de *software*.

Unidades de substituição, também denominadas *unidades suplentes*, podem ser passivas ou ativas. Uma suplente ativa (*hot spare*) realiza concorrentemente as mesmas operações realizadas pela unidade que ela poderá vir a substituir, não necessitando de inicialização quando ocorre a troca de unidades no sistema. Já uma suplente passiva (*cold spare*) é uma unidade que ou não está ligada ou está sendo usada para outras tarefas, requerendo inicialização quando ela é ativada para substituir uma unidade que falhou. Projetistas de sistemas devem pesar o custo de se manter suplentes ociosas contra o custo do tempo de inicialização quando forem decidir entre usar suplentes ativas ou passivas [Nelson 90], como discutido para as técnicas de replicação ativa e passiva..

Neste trabalho, a reconfiguração do sistema é considerada em termos de unidades de *software*, no sentido de que, quando uma falha ocorre, seja ela no *hardware* ou no *software*, novas réplicas dos componentes de *software* são adicionadas ao sistema. Isto leva em consideração que, em um nível mais alto de abstração, a substituição de um componente de *hardware* nada mais é do que a realocação das réplicas dos componentes de *software* que estavam sendo ali executadas em outras unidades de processamento do sistema (ou, em certos casos, nele próprio), que podem ou não já estarem sendo usadas para outros processamentos.

6.3. Realocação de Componentes de *Software*

Quando uma unidade de processamento falha, algumas ou todas as réplicas de componentes de *software* que estavam executando naquela unidade passam a funcionar incorretamente ou simplesmente param de funcionar. A fim de restaurar o nível de redundância requerida pelos protocolos de processamento de erro para continuarem funcionando corretamente diante de falhas posteriores, tais réplicas precisam ser criadas novamente. A localização em que a nova réplica é criada pode ser [Powell 92]:

a) a mesma unidade de processamento em que a réplica original estava localizada antes da falha; ou

b) qualquer outra localização dentro do domínio de replicação do componente.

A alternativa "a" pode ser selecionada quando:

(i) a falta é considerada transiente e assim a simples reinicialização do estado da réplica é suficiente para trazê-la de volta ao funcionamento normal; ou

(ii) a manutenção corretiva da unidade que apresentou a falha foi executada.

Se a realocação das réplicas não é possível, então alguns componentes de *software* ou terão de ser abandonados em favor de outros mais críticos ou, no mínimo, a operação tolerante a faltas será degradada para uma operação de falha controlada a fim de garantir segurança dos dados e/ou integridade das aplicações distribuídas. Na ausência de recursos, a reconfiguração do sistema terá de ser adiada até que alguma unidade de processamento se recupere (após a manutenção).

Quando a falha ocorre na própria réplica, caracterizando uma falta de concepção do *software*, a nova réplica deve ser derivada de um projeto diferente da anterior e pode ser alocada na mesma unidade de processamento. Na impossibilidade ou inviabilidade de diversidade de projeto para as réplicas do componente de *software*, uma alternativa para tolerar certas faltas de concepção é alocar a nova réplica em uma plataforma diferente, visto que leves diferenças no ambiente local das réplicas podem levar tais faltas a se manifestarem independentemente nas diferentes réplicas [Powell 92].

6.4. Reconfiguração com Replicação Ativa

Como foi colocado no Capítulo 3, um dos requisitos da técnica de replicação ativa é que as réplicas sejam determinísticas. Tal determinismo pode ser conseguido através da adoção de um modelo restritivo de computação baseado em *máquinas de estado.* Ou seja, os componentes de *software* a serem replicados ativamente devem ser estruturados como *máquinas de estados*, que consistem de variáveis de estado, que guardam o seu estado, e comandos, que transformam este estado e são implementados por um programa determinístico. Réplicas de máquinas de estado são mais comumente conhecidas como *réplicas ativas.*

No mesmo capítulo, vimos também que um sistema consistindo de um grupo de réplicas de uma máquina de estado é dito ser *t-resiliente* (ou *tolerante-a-t-faltas*) se ele satisfaz sua especificação desde que não mais do que t destas réplicas apresentem falha durante um dado intervalo de tempo.

Entretanto, um grupo de réplicas ativas é capaz de tolerar mais do que t faltas se é possível: (1) remover do grupo réplicas falhas (ou executando em unidades de processamento falhas); e (2) adicionar ao grupo novas réplicas (executando em unidades

de processamento livres de falha ou reparadas). Ou seja, se a reconfiguração do sistema é realizada após faltas terem sido diagnosticadas.

6.4.1. Condição de Combinação e Tolerância Ilimitada

Seja $P(\tau)$ o número total de réplicas de alguma máquina de estado de interesse no tempo τ e seja $F(\tau)$ o número de réplicas falhas no tempo τ. Para garantir que o grupo produza a saída correta, a seguinte condição deve ser satisfeita [Schneider 90]:

Condição de Combinação: $P(\tau) - F(\tau) > Enuf$ para todo $\tau \geq 0$, onde

Enuf = $P(\tau)/2$ se falhas Bizantinas são possíveis, ou

Enuf = 0 se apenas falhas silenciosas são possíveis.

No caso de falhas Bizantinas, a condição obriga que mais da metade das réplicas estejam livres de falha, de forma que o grupo de réplicas produza um valor majoritário correto. Para falhas silenciosas, como todas as réplicas geram valores corretos e idênticos, basta que uma delas esteja operante para que uma saída correta seja produzida pelo grupo.

Falhas das réplicas podem levar a Condição de Combinação a ser violada à medida que $F(\tau)$ aumenta e, conseqüentemente, $P(\tau) - F(\tau)$ diminui. Quando falhas Bizantinas são possíveis, a remoção da réplica falha diminui *Enuf* sem diminuir $P(\tau) - F(\tau)$; isto pode impedir que a Condição de Combinação seja violada.

Consideremos o seguinte exemplo: suponha um grupo composto por cinco réplicas que podem exibir falhas Bizantinas. Temos inicialmente que $P(\tau) = 5$ e *Enuf* = 2,5. Pela restrição $P(\tau) > 2F(\tau)$, em condições normais, no máximo duas faltas Bizantinas podem ser toleradas. Porém, se a remoção das réplicas falhas é feita, é possível tolerar um total de três faltas: após as duas faltas permitidas teríamos $P(\tau) = 3$ e *Enuf* = 1,5; ocorrendo mais uma falta, ficaríamos com $P(\tau) - F(\tau) = 3 - 1 = 2$ que é maior do que o atual valor de *Enuf*. Portanto, a Condição de Combinação continua sendo satisfeita mesmo após três faltas.

Quando apenas falhas silenciosas são de interesse, adicionar novas réplicas ao grupo é a única maneira de impedir a violação da condição acima, visto que aumentar $P(\tau)$ é o único meio de garantir que $P(\tau) - F(\tau) > 0$ se mantenha.

Com base nas colocações acima podemos afirmar que é possível manter a Condição de Combinação eternamente satisfeita, e assim tolerar um número total ilimitado de faltas durante a vida do sistema, se as seguintes condições podem ser garantidas [Schneider 90]:

F1: Se falhas Bizantinas são possíveis, então as réplicas da máquina de estado apresentando falhas são identificadas e removidas do grupo antes que a Condição de Combinação seja violada por falhas subseqüentes.

F2: Novas réplicas da máquina de estado são adicionadas ao grupo antes que a Condição de Combinação seja violada por falhas subseqüentes.

Além do mais, remover réplicas falhas de um grupo de máquinas de estado pode ainda melhorar o desempenho do sistema. Isto porque o número de mensagens que devem ser enviadas para alcançar acordo sobre o conteúdo de um pedido é geralmente proporcional ao número de réplicas que participam deste acordo, bem como o tempo de execução de alguns protocolos de acordo é proporcional ao número de réplicas falhas no grupo. Remover estas réplicas claramente reduz tanto o número de mensagens trocadas quanto o tempo gasto na execução de tais protocolos.

6.4.2. Gerência da Configuração

Conforme definido em [Schneider 90], a configuração de um sistema estruturado como uma máquina de estado e dos seus clientes pode ser descrita usando-se três conjuntos: os clientes *C*, as réplicas da máquina de estado *S* e os dispositivos de saída *O*. Como estamos particularmente interessados em tolerar faltas da máquina de estado propriamente dita, trataremos apenas da gerência de configuração a ela relacionada.

Dois problemas devem ser solucionados para suportar mudanças na configuração do sistema. Primeiro, o conjunto *S* deve estar disponível sempre que for requerido. Segundo, sempre que uma máquina de estado é adicionada à configuração, o seu estado deve ser atualizado para refletir o estado corrente do sistema. Este último problema é discutido na próxima seção.

A configuração de um sistema pode ser gerenciada usando a máquina de estado daquele sistema [Schneider 90]. O conjunto *S* é armazenado em variáveis de estado e modificado através de comandos. Cada configuração é *válida* para uma coleção de

pedidos - aqueles pedidos *r* tal que *uid(r)* está na faixa definida por dois pedidos sucessivos de mudança de configuração. Assim, sempre que uma réplica da máquina de estado executa uma ação relacionada ao processamento de *r*, ela usa a configuração que é válida para *r*. Isto significa que um pedido de mudança de configuração deve escalonar a nova configuração para um ponto futuro longe o suficiente para que todas as réplicas da máquina de estado tomem conhecimento da nova configuração antes dela entrar em vigor efetivamente e, desta forma, possam se manter consistentes.

Pedidos para mudar a configuração do sistema são feitos por um mecanismo de detecção de falha/recuperação. É conveniente pensar neste mecanismo como uma coleção de clientes, um para cada elemento de *S*. Cada um destes *configuradores* é responsável por detectar a falha ou recuperação do único objeto que ele gerencia e, então, fazer um pedido para o elemento de S alterar a configuração.

Um configurador livre de falha satisfaz as duas propriedades seguintes [Schneider 90]:

C1. Um elemento só é removido da configuração se ele apresenta alguma falha.

C2. Um elemento só é adicionado à configuração se ele é livre de falha.

Mudar a configuração do sistema só aumenta a tolerância a faltas se, além de C1 e C2, F1 e F2 também se mantiverem. Para F1 e F2 se manterem, um configurador deve: (1) detectar faltas e solicitar a remoção de elementos; e (2) detectar reparos e solicitar a adição de elementos.

A fim de analisarmos a tolerância a faltas de um sistema que usa configuradores, a falha de um configurador pode ser considerada equivalente à falha do elemento que ele gerencia. Isto porque, com respeito a Condição de Combinação, a remoção de um elemento livre de falha ou a adição de um elemento falho tem o mesmo impacto de um elemento ter falhado. Assim, num sistema t-resiliente, a soma do número de configuradores falhos mais o número de componentes falhos cujos configuradores são livres de falha deve ser limitada a *t*.

6.4.3. Integração de um Objeto Recuperado

Um objeto adicionado à configuração deve não apenas ser livre de falha, como também deve ter o seu estado correto e atualizado de modo que suas ações sejam consistentes com aquelas do resto do sistema.

Seja $sm[r_i]$ o estado em que uma máquina de estado sm livre de falha estaria depois de processar os pedidos de r_0 até r_i. Uma máquina de estado m se juntando à configuração imediatamente depois de um pedido r_{join} deve estar no estado $e[r_{join}]$ antes de poder participar do sistema em execução.

O protocolo para integrar uma réplica da máquina de estado sm_{new} ao sistema é um pouco complexo, visto que não é suficiente que uma réplica sm_i simplesmente envie os valores de todas as suas variáveis de estado e cópias de quaisquer pedidos pendentes para sm_{new}. Isto porque alguns pedidos de clientes podem ter sido recebidos por sm_i depois do envio de $e[r_{join}]$, mas supostamente entregues a sm_{new} antes do seu reparo. Tal pedido nunca seria refletido na informação de estado repassada por sm_i para sm_{new} nem recebido por sm_{new} diretamente. Assim, sm_i deve, durante algum tempo, retransmitir para sm_{new} pedidos recebidos dos clientes. Visto que os pedidos de um dado cliente são recebidos por sm_{new} na ordem enviada e ordenados de forma ascendente pelo identificador do pedido, uma vez que sm_{new} recebe um pedido p diretamente de um cliente c, não é mais necessário que pedidos de c com identificadores maiores que o identificador de p sejam repassados de sm_i para sm_{new}. Se sm_{new} informa a sm_i o identificador do primeiro pedido recebido diretamente de cada cliente c, então sm_i pode saber quando deve parar de retransmitir para sm_{new} pedidos de c.

Quando os componentes do sistema exibem apenas falhas silenciosas, uma única réplica da máquina de estado é suficiente para integrar o novo elemento no sistema, já que a informação de estado obtida de qualquer réplica é correta. Já para componentes com semântica de falha Bizantina, uma única réplica não é suficiente porque a informação fornecida pela réplica pode não ser correta. Para tolerar t faltas, sob suposição de falha Bizantina, num sistema com $2t + 1$ réplicas, $t + 1$ cópias idênticas da informação de estado e $t + 1$ cópias idênticas de mensagens retransmitidas precisam ser obtidas [Schneider 90].

6.5. Reconfiguração com Replicação Passiva e Semi-Ativa

As estratégias de replicação passiva e semi-ativa trabalham sob suposições de semântica de falha silenciosa, o que implica que para tolerar t faltas, apenas $t + 1$ réplicas são necessárias, garantindo que pelo menos uma delas permanecerá operante. Uma vez que todas as t faltas ocorram, a próxima falta não mais seria tolerada.

Entretanto, durante o tempo de missão do sistema, mais de t faltas podem ser toleradas se for possível adicionar novas réplicas ao grupo que implementa o componente de *software* antes que a condição $N > t$ seja violada, onde N é o número total de réplicas do grupo em cada instante. Aliás, se isto sempre for possível e for realizado, o grau de tolerância a faltas pode ser mantido constante durante toda a missão do sistema. Isto funciona de forma semelhante ao discutido para replicação ativa com suposição de semântica de falha silenciosa (vide Seção 6.4.1).

Embora seja suficiente que apenas uma das réplicas (primária/líder) esteja funcionando a cada instante para que o serviço apropriado seja fornecido, é necessário que os estados de todas as réplicas sejam monitorados para evitar que a condição acima colocada seja violada, o que poderia ocorrer quando a réplica primária/líder falhasse e, ao tentar eleger uma réplica substituta, percebesse que todas as outras réplicas também falharam e suas falhas passaram desapercebidas.

Associado a cada réplica, portanto, deve haver um processo *configurador*, que é responsável por detectar a falha da réplica que ele gerencia, removê-la da configuração atual e solicitar a adição de uma nova réplica ao grupo. As propriedades para um configurador livre de falha apresentadas para replicação ativa (vide Seção 6.4.2) também são válidas para as replicações passiva e semi-ativa.

A integração de uma nova réplica se dá também de forma bastante semelhante à replicação ativa para semântica de falha silenciosa: a réplica primária/líder envia uma cópia do seu estado corrente para a nova réplica. No caso da replicação semi-ativa, uma cópia de todas as mensagens na fila de entrada também deve ser enviada, bem como cópias das novas mensagens que chegam para garantir que a nova réplica as receberá; quando a nova réplica percebe que já está recebendo corretamente as mensagens diretamente dos clientes, ela comunica o fato à primária/líder para que ela não mais as retransmita (vide Seção 6.4.3). Este procedimento também deve ser feito para as

implementações de replicação passiva onde as suplentes mantêm diário das mensagens de entrada, como ocorre, por exemplo, quando a estratégia de salvaguarda periódica é adotada.

6.6. Sumário

Em um sistema tolerante-a-t-faltas, é possível tolerar efetivamente mais do que *t* faltas se a reconfiguração do sistema for realizada antes do sistema falhar. A atividade de reconfiguração, portanto, tem por objetivo manter a capacidade de tolerância a faltas do sistema durante todo o tempo de sua missão.

Um sistema pode ser reconfigurado de duas formas: (1) substituindo o componente falho por um suplente; ou (2) reconfigurando a estrutura do sistema ou a distribuição de trabalho. A primeira forma é normalmente usada na tolerância a faltas em componentes de *software*, enquanto a última é geralmente aplicável a componentes de *hardware*.

Uma unidade de substituição é uma unidade básica de falha, substituição e crescimento. Unidades de substituição existem tanto no *hardware* quanto no *software*. Elas podem ainda ser ativas ou passivas.

Quando falhas Bizantinas são consideradas, o nível de tolerância a faltas pode ser mantido através da remoção do componente falho, bem como da adição de componentes livres de falha. Já com falhas silenciosas, a adição de novos componentes é a única forma de manter o grau de resiliência do sistema durante todo o tempo de sua missão.

Um aspecto importante da atividade de reconfiguração do sistema é que ela seja executada *on-line* e sem nenhuma intervenção manual.

O Seljuk-Amoeba oferece um serviço de reconfiguração automática para grupos que implementam servidores sem estado, independente da técnica de replicação adotada. No caso de servidores com estado, a reconfiguração automática só é possível para grupos que seguem o modelo de réplicas passivas, a menos que, nos demais modelos, a própria aplicação implemente algum mecanismo de recuperação de estado.

Capítulo 7:

Tolerância a Faltas no Seljuk-Amoeba

7.1. Introdução

Com o aumento dos requisitos de confiança no funcionamento e, conseqüentemente, da complexidade dos sistemas baseados em computador, o esforço desprendido na construção destes sistemas tem aumentado consideravelmente. Uma possível maneira de minimizar este esforço é prover um ambiente operacional que ofereça suporte para a construção e a execução de aplicações distribuídas robustas, livrando o programador do trabalho de implementar, ele próprio, os mecanismos para tolerância a faltas sobre os quais tais aplicações são construídas e executadas.

Neste capítulo, descreveremos uma proposta para a implementação dos serviços para tolerância a faltas no ambiente operacional Seljuk-Amoeba [Brasileiro 97]. Tal implementação possui a grande vantagem de não requerer qualquer recurso especial de *hardware*, visto que os serviços aqui descritos, como toda a plataforma Seljuk, serão implementados inteiramente em *software*. Além disso, os serviços propostos apresentam-se bastante flexíveis à medida que a aplicação pode, em tempo de ativação, selecionar o grau de tolerância a faltas desejado e a semântica de falha assumida para a infra-estrutura de processamento e comunicação. Adicionalmente, apenas aquelas aplicações que requisitem tais serviços pagarão, sob a forma de queda de desempenho, pelo seu uso.

Iniciaremos o capítulo apresentando o modelo arquitetural do Seljuk, com ênfase nos seus objetivos e na sua estrutura. Em seguida, descreveremos resumidamente o sistema operacional distribuído Amoeba, ambiente em que a plataforma será desenvolvida, destacando o seu suporte à comunicação interprocessos. Por fim, a proposta de implementação dos mecanismos para tolerância a faltas identificados nos capítulos anteriores será apresentada e discutida.

7.2. A Arquitetura Seljuk

7.2.1. Objetivos

A arquitetura Seljuk tem como objetivo principal prover uma plataforma de desenvolvimento que facilite a construção de aplicações distribuídas robustas através do fornecimento de duas classes de serviços:

(i) serviços que permitem restringir a semântica de falha dos componentes que formam a infra-estrutura de processamento sobre a qual a aplicação irá executar; e

(ii) serviços que implementam mecanismos para tolerância a faltas.

A primeira classe é responsável pelo oferecimento de serviços de processamento com diferentes semânticas de falha, que possam satisfazer a suposição estabelecida pela aplicação sobre o modo de falha dos componentes do sistema distribuído. A definição da semântica de falha assumida é feita pela aplicação no instante de sua ativação. A plataforma fica então responsável por prover os mecanismos necessários para garantir tal semântica, considerando as limitações do *hardware* disponível (processadores e canais de comunicação) e a semântica de falha real deste *hardware* (que também é definida pela aplicação). Obviamente, quando a semântica de falha real dos processadores é pelo menos tão restritiva quanto a semântica requerida pela aplicação, não há necessidade de se usar os serviços da plataforma. Tais serviços são discutidos em mais detalhes em [Brasileiro 97] e [Gallindo-Brasileiro 97].

A segunda classe, por sua vez, inclui serviços de mais alto nível que provêem suporte básico para a construção de aplicações distribuídas robustas, quais sejam: replicação do processamento, comunicação em grupo, diagnóstico de faltas e reconfiguração do sistema. A implementação destes serviços utiliza os serviços da classe anterior para obter uma semântica de falha mais restritiva, diminuindo assim a sua complexidade. Esta última classe de serviços será assunto de discussão deste capítulo.

7.2.2. Estrutura

A arquitetura Seljuk segue um modelo organizado em camadas, conforme ilustra a Figura 7.1, extraída de [Brasileiro 97].

A camada mais baixa da arquitetura, denominada "Processadores e Canais de Comunicação", abriga os componentes de *hardware* que fornecem os serviços básicos de processamento e comunicação. A camada seguinte é constituída pelo "Sistema Operacional Distribuído", responsável por controlar e gerenciar os diversos componentes do sistema, considerando principalmente a distribuição de tarefas pelos vários processadores e as necessidades de comunicação decorrentes disto. É nesta camada onde estão implementados os serviços que garantem a semântica de falha requerida pela aplicação (classe i) e alguns dos serviços necessários à implementação dos mecanismos para tolerância a faltas (classe ii).

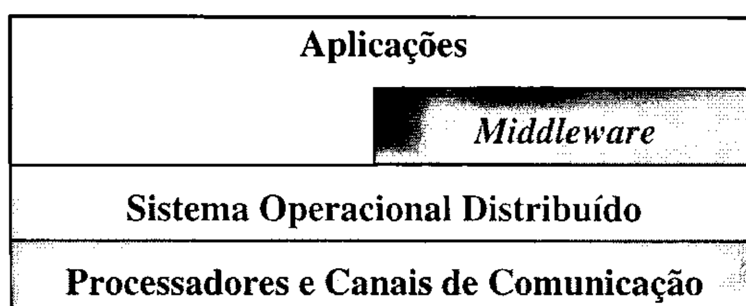


Figura 7.1: Estrutura do Ambiente Operacional Seljuk

A camada intermediária, denominada "*Middleware*", implementa os demais serviços para tolerância a faltas, tirando proveito dos serviços fornecidos pelas camadas inferiores. Finalmente, a camada superior agrupa os componentes que implementam o serviço oferecido pela aplicação propriamente dita.

Observa-se, portanto, que, além dos serviços normalmente oferecidos por um ambiente operacional tradicional, o Seljuk oferece serviços de processamento e comunicação confiáveis e, apoiados nestes, serviços especializados para tolerância a faltas. Como será visto mais adiante, estes serviços são oferecidos pela arquitetura de forma flexível e com ônus apenas para aquelas aplicações que fazem uso deles.

7.2.3. Ambiente de Desenvolvimento

O ambiente operacional Seljuk está sendo especificado e desenvolvido sobre sistemas operacionais baseados na tecnologia de micronúcleo, cujo princípio básico é minimizar o tamanho do sistema operacional que executa em modo supervisor com o objetivo de aumentar a sua flexibilidade.

Um micronúcleo provê basicamente quatro serviços mínimos: um mecanismo para comunicação interprocessos, alguma gerência de memória, uma quantidade limitada de gerência/escalonamento de processos e manipulação de entrada/saída de baixo nível. Todos os demais serviços do sistema operacional (p. ex., sistema de arquivos, sistema de diretório etc.) são geralmente implementados como servidores que executam em modo usuário. Para obter um serviço, o usuário envia uma mensagem para o servidor apropriado, que realiza a tarefa e retorna o resultado [Tanenbaum 92, pp. 388-389]. Este modelo apresenta várias vantagens:

(i) alta modularidade: há uma interface bem definida para cada serviço e todo serviço é igualmente acessível para todo cliente independente de sua localização;

(ii) facilidade para implementar, instalar e depurar novos serviços, já que adicionar ou modificar um serviço não requer parar o sistema e reiniciar um novo núcleo;

(iii) os serviços podem ser mais facilmente modificados/adaptados para atender às exigências das diferentes aplicações;

(iv) usuários não satisfeitos com os serviços oficiais oferecidos pelo sistema operacional podem escrever seus próprios servidores.

Um dos micronúcleos de maior destaque no meio acadêmico e científico, por sua disponibilidade tanto a nível de código fonte quanto de documentação, é o Amoeba [Mullender et al. 90], que, por este motivo, foi escolhido como sistema hospedeiro da primeira implementação do Seljuk, batizada de *Seljuk-Amoeba*.

7.3. O Sistema Operacional Distribuído Amoeba

O Amoeba é um sistema operacional distribuído - ele faz um conjunto de processadores e equipamentos de estrada/saída agirem como um computador único. O projeto do Amoeba originou-se na Universidade Vrije, Amsterdã, com o objetivo principal de construir um sistema operacional distribuído transparente. Para o usuário comum, usar o Amoeba é como usar um sistema de tempo compartilhado tradicional como o UNIX: ele se conecta ao sistema (*login*), edita e compila programas, move arquivos etc. Na verdade, a diferença é que cada uma destas ações faz uso de múltiplas

máquinas espalhadas ao longo da rede, mas o usuário não toma ciência disto [Tanenbaum 92, pp. 589]. Quando um usuário se conecta, ele o faz no sistema como um todo, não em uma máquina específica. Todos os recursos pertencem ao sistema como um todo, não sendo dedicados a usuários específicos. O modelo de *software* do Amoeba é baseado na tecnologia de micronúcleos, como veremos mais adiante.

7.3.1. Arquitetura do Amoeba

Um sistema Amoeba é formado basicamente por três componentes:

(i) *pools* de processadores;

(ii) terminais; e

(iii) servidores.

Toda a força computacional do sistema está localizada em um ou mais *pools* de processadores, cada qual consistindo de um número substancial de processadores, que podem inclusive ser de diferentes arquiteturas. Cada processador, por sua vez, possui sua própria memória local e conexão à rede. Um *pool* não necessariamente é um computador com uma única placa contendo todos os processadores - quando isto não é disponível, um conjunto de microcomputadores ou estações de trabalho pode ser projetado para compor um *pool*. Além do mais, a localização física dos componentes de um *pool* é irrelevante; os processadores do *pool* podem até mesmo estar localizados em países diferentes.

Os processadores de um *pool* não são dedicados a qualquer usuário do sistema. A cada comando do usuário, o sistema dinamicamente aloca um ou mais processadores para executar aquele comando. Os processadores escolhidos são aqueles que apresentam menor carga de processamento no momento em que o comando foi submetido. Portanto, cada comando emitido pelo usuário pode ser (e provavelmente será) executado em um processador diferente do sistema.

O segundo elemento da arquitetura é o terminal, através do qual o usuário tem acesso ao sistema. Um terminal Amoeba típico é um terminal X, com uma grande tela e um *mouse*. Alternativamente, um computador pessoal ou uma estação de trabalho podem também ser usados como terminal.

O outro componente do Amoeba consiste de servidores especiais, tais como servidor de arquivos ou de diretórios, que por razões de hardware e de software necessitam rodar em um processador exclusivo. Em muitos casos, um servidor pode ser executado no processador de um *pool*, sendo ativado apenas quando invocado, porém, por razões de desempenho, é melhor deixá-lo ativo todo o tempo.

7.3.2. O Modelo de *Software* do Amoeba

Em se tratando de *software*, o Amoeba consiste de dois componentes: um *micronúcleo*, que é executado em todo processador do sistema, e uma *coleção de servidores*, que provêem a maior parte das funcionalidades de um sistema operacional tradicional. O mesmo núcleo é executado nos processadores, nos terminais (se estes são computadores) e nos servidores especializados.

O micronúcleo tem quatro funções básicas:

(i) Gerenciar processos e suas linhas de execução (*threads*). O Amoeba provê o conceito de processos, bem como múltiplas linhas de execução dentro de um único espaço de endereçamento, porém com registradores, contadores de programa e pilha próprios.

(ii) Prover suporte para gerência de memória de baixo nível. As diversas linhas de execução de um processo podem alocar e desalocar blocos de memória, denominados *segmentos*, que podem ser lidos, escritos e mapeados dentro e fora do espaço de endereçamento do processo ao qual a linha de execução pertence.

(iii) Gerenciar entrada/saída de baixo nível. Associado a cada dispositivo de entrada/saída conectado à máquina, há um *driver* de dispositivo no núcleo responsável por gerenciar todas as operações de entrada/saída.

(iv) Manipular comunicação interprocessos. Duas formas de comunicação são fornecidas pelo Amoeba: *comunicação ponto-a-ponto* e *comunicação em grupo*. O primeiro tipo de comunicação é baseado no modelo cliente-servidor, onde um cliente envia uma mensagem para um servidor e se bloqueia até que este envie uma resposta de volta. A outra forma de comunicação permite uma mensagem ser enviada de um único transmissor para múltiplos destinos de

uma só vez. Os protocolos oferecidos pelo Amoeba provêem comunicação em grupo atômica para os processos do usuário mesmo na presença de falhas de comunicação e, opcionalmente, falhas de processamento.

Todas as demais funções que não são realizadas pelo núcleo ficam sob a responsabilidade de processos servidores. A idéia deste desenho é minimizar o tamanho do núcleo e aumentar a flexibilidade do sistema. Implementar serviços padrões, como o sistema de arquivos, fora do núcleo permite que eles sejam mais facilmente modificados e que múltiplas versões possam ser executadas simultaneamente no sistema para atender às necessidades de diferentes comunidades de usuários [Tanenbaum 92, pp. 592-594].

Cada servidor define uma interface procedural que os clientes podem chamar. Estas rotinas de biblioteca são procedimentos *stubs*. Para usar um servidor, um cliente normalmente faz uma chamada ao *stub*, que empacota os parâmetros, envia a mensagem para o servidor e se bloqueia esperando sua resposta. Este mecanismo esconde do usuário todos os detalhes de implementação, permitindo que mudanças sejam introduzidas nos servidores sem que os seus clientes tomem conhecimento delas. O Amoeba oferece um compilador para aqueles usuários que desejem escrever *stubs* para seus próprios servidores. Embora as primitivas para comunicação cliente-servidor a nível de núcleo sejam na verdade relacionadas com passagem de mensagens, o uso de *stubs* faz este mecanismo parecer uma RPC (*Remote Procedure Call*) [Nelson 81] aos olhos do programador.

Um dos servidores de maior interesse no Amoeba é o *Run Server*, responsável por decidir em qual processador um comando do usuário deve ser executado. Quando um usuário digita um comando, o próprio interpretador de comandos do Amoeba (*shell*) verifica se aquele comando está disponível para várias arquiteturas. O *shell* faz uma chamada ao *Run Server* informando para quais arquiteturas o comando está disponível e solicitando a indicação de um processador para executar o comando. O *Run Server* faz então uma seleção preliminar, resultando apenas nos processadores das arquiteturas indicadas. Em seguida, dentre os processadores selecionados, o *Run Server* elimina aqueles que não possuem memória suficiente para executar o programa. Finalmente, uma estimativa é feita sobre o poder computacional que pode ser dedicado ao programa por cada processador resultante das seleções anteriores. Aquele processador que apresentar melhor resultado é indicado para o *shell*, que faz então uma chamada para o servidor de

processos daquele processador, solicitando a criação do processo que executará o comando do usuário.

7.3.3. Comunicação Interprocessos

O sistema de comunicação do Amoeba é implementado no núcleo em duas camadas. A camada inferior implementa o *Fast Local Internet Protocol* (FLIP), que é um protocolo que opera em modo datagrama, não orientado a conexão. Comunicação confiável é fornecida pelos dois serviços da camada superior - comunicação em grupo e RPC, que utilizam o serviço não-confiável fornecido pelo FLIP para enviar mensagens confiavelmente. O FLIP é um protocolo da camada de rede especialmente projetado para atender às necessidades da computação distribuída.

A cada processo é associado um número aleatório de 64 bits - *endereço FLIP*, que é a base para comunicação de baixo nível do Amoeba. Um endereço FLIP identifica unicamente um processo (ou um grupo deles), não uma máquina, o que torna a comunicação no Amoeba insensível a mudanças na topologia e no endereçamento da rede [Tanenbaum 92, pp. 618].

Toda comunicação ponto-a-ponto no Amoeba consiste de um cliente enviando uma mensagem para um servidor e o servidor enviando uma mensagem de resposta para o cliente, porém, como dito anteriormente, o uso de *stubs* torna este mecanismo de troca de mensagens transparente para o programador, que enxerga a comunicação como um mecanismo de RPC e assim a trataremos daqui por diante.

Para um cliente fazer uma RPC com um servidor, é necessário porém que ele conheça o endereço do servidor. Cada linha de execução de um processo pode escolher um número aleatório de 48 bits, denominado *porta*, a ser usado como endereço para as mensagens enviadas para aquela linha de execução. Uma porta nada mais é do que um tipo de endereço lógico semelhante a um endereço IP, só que ele não está "amarrado" a qualquer localização física particular. Uma porta, na verdade, identifica um *serviço* e não um *servidor*. Na atual implementação do Amoeba, um serviço pode ser fornecido por múltiplos servidores independentes com o objetivo de aumentar o desempenho e a confiabilidade. Servidores replicados normalmente escutam a mesma porta, porém podem possuir endereços FLIP diferentes. Quando um cliente solicita um serviço

fornecido por vários servidores, a camada RPC seleciona um dentre os vários servidores (endereços FLIP) para o qual o pedido é enviado.

Além das primitivas para comunicação ponto-a-ponto, o Amoeba também oferece primitivas para comunicação em grupo que garantem entrega atômica de mensagens a todos os membros livres de falha do grupo mesmo na presença de falhas dos canais de comunicação e, opcionalmente, dos processadores. Tais primitivas são implementadas sobre um protocolo de difusão confiável que garante ordenação total de mensagens mesmo na ocorrência de problemas na transmissão das mensagens.

Desta forma, as primitivas de grupo provêem uma abstração que permite os programadores construírem aplicações consistindo de um ou mais processos executando em diferentes máquinas. Todos os membros do grupo vêem todos os eventos relacionados ao seu grupo na mesma ordem, incluindo aqueles eventos de junção/remoção de um membro e de recuperação de falhas no grupo.

A estrutura de grupos do Amoeba é fechada, significando que um processo que deseje enviar mensagem para um grupo deve fazer parte deste grupo. Este desenho foi escolhido para prover o maior grau de transparência possível para o cliente da aplicação. A idéia é que da mesma forma que os clientes normalmente usam RPC para se comunicar com servidores individuais, eles também usariam RPC para se comunicar com grupos de servidores. Desta forma, para um cliente ter acesso ao serviço provido por um grupo, ele deve fazer uma RPC com um dos membros do grupo, que, se necessário, usa comunicação em grupo para informar aos outros membros do grupo, de forma consistente, o serviço solicitado pelo cliente. O Amoeba oferece por exemplo uma primitiva (*ForwardRequest*) que permite um membro repassar o pedido recebido para um outro membro do grupo; isto é útil quando o servidor que recebeu originalmente um pedido via RPC não está hábil a atendê-lo, podendo repassá-lo então para um outro servidor de forma totalmente transparente para o cliente.

A implementação da comunicação em grupo no Amoeba será assunto de estudo na Seção 7.5.1.

7.4. Pressupostos

A implementação dos mecanismos para tolerância a faltas no ambiente Seljuk-Amoeba assume que os componentes do sistema que formam a infra-estrutura de processamento das aplicações possuem semântica de falha silenciosa. Para garantir tal semântica, estes serviços invocam o serviço de processamento confiável oferecido pelo próprio Seljuk-Amoeba. Este serviço é fornecido por um servidor denominado *FT Run Server*, cujos detalhes são apresentados em [Gallindo-Brasileiro 97]. Mais adiante, a funcionalidade e o funcionamento deste servidor se tornarão mais claros.

Esta restrição, além de facilitar a implementação de todos os serviços, permite o reaproveitamento de grande parte do suporte à comunicação em grupo oferecido pelo Amoeba, que provê um protocolo de difusão que garante entrega atômica de mensagens na presença de falhas *silenciosas* em alguns processadores do sistema. A utilização de nodos[9] com falha silenciosa permite ainda o suporte às técnicas de replicação passiva e semi-ativa, que fazem esta suposição sobre a semântica de falha dos componentes do sistema, bem como aumenta o desempenho dos protocolos de processamento de erro utilizados pela replicação ativa, sobretudo com respeito à ordenação de mensagens exigida por esta técnica. Protocolos para ordenação de mensagens sob suposições de falha silenciosa trabalham bem mais rápidos do que sob suposições de falhas Bizantinas.

Os servidores de *software* que implementam os serviços para tolerância a faltas, por sua vez, são considerados livres de falha. A realização deste pressuposto pode ser atingida através da execução dos servidores em nodos com semântica de falha mascarada, também oferecidos pelo *FT Run Server*.

Caso não se deseje usar os serviços do *FT Run Server*, uma alternativa é utilizar o chamado *Boot Server* do próprio Amoeba, que, de tempos em tempos, inspeciona o funcionamento dos servidores indicados em seu arquivo de configuração. Se o servidor inspecionado falha em responder após um certo número de tentativas, o *Boot Server* declara que o servidor está "morto" e solicita a alocação de um novo processador do *pool* para executar uma nova cópia do servidor. Isto implica, porém, que deve haver alguma forma de recuperar o estado corrente do servidor antes de sua falha. Um

[9] Nodos são as unidades de processamento fornecidas pelo *FT Run Server* que apresentam a semântica de falha requerida pela aplicação.

mecanismo de armazenagem estável [Jalote 94, pp. 99-107] pode ser usado para este fim.

Obviamente, servidores sem estado não necessitam guardar e, portanto, recuperar qualquer estado, o que simplifica a criação de um nova cópia. Utilizar o *FT Run Server*, em vez do *Boot Server*, entretanto, garante que não haverá interrupção no serviço que se pretende manter tolerante a faltas.

Como a implementação dos serviços de replicação, diagnóstico e reconfiguração se apóiam no serviço de comunicação, começaremos a proposta tratando deste serviço.

7.5. Serviço de Comunicação em Grupo

A seguir veremos como a comunicação em grupo é atualmente implementada no Amoeba e depois apresentaremos nossas proposta de implementação no Seljuk-Amoeba.

7.5.1. Comunicação em Grupo no Amoeba

Vimos anteriormente que a comunicação em grupo do Amoeba garante entrega confiável e ordenada de mensagens a todos os membros livres de falha do grupo, mesmo na presença de falhas dos canais de comunicação e, opcionalmente, dos processadores. As primitivas para gerência de grupo e comunicação em grupo integradas ao Amoeba, bem como os seus algoritmos e medidas de desempenho, são descritas detalhadamente em [Kaashoek-Tanenbaum 94] e apresentadas resumidamente na Tabela 7.1.

Como a estrutura de grupo adotada pelo Amoeba é fechada (vide Seção 7.3.3), tais primitivas, exceto *JoinGroup()*, só podem ser chamadas por processos que são membros do grupo.

No contexto do nosso trabalho, um grupo é formado por réplicas de um mesmo processo (digamos, um servidor) que cooperam para fornecer um serviço único adequado. No Amoeba, a cada grupo é associado um identificador único, denominado *porta*. Todos os membros do grupo que implementam aquele servidor conhecem a porta que ele escuta. O conceito de portas provê a transparência desejada para os clientes do serviço, que não precisam tomar conhecimento que o servidor que está atendendo seu pedido é replicado.

Primitiva	Descrição
CreateGroup	Cria um novo grupo com as características definidas pelo chamador.
JoinGroup	Torna o chamador um membro do grupo.
LeaveGroup	Remove o chamador do grupo.
SendToGroup	Envia atomicamente uma mensagem para todos os membros do grupo.
ReceiveFromGroup	Bloqueia o chamador até a chegada de uma mensagem do grupo.
ResetGroup	Inicia a recuperação depois da falha de um membro do grupo.
GetInfoGroup	Retorna informações sobre o estado do grupo, tais como: número de membros e identificador do chamador no grupo.
ForwardRequest	Repassa um pedido endereçado ao grupo para um outro membro do grupo.

Tabela 7.1: Primitivas para Comunicação em Grupo do Amoeba

Usando a comunicação em grupo nativa do Amoeba, baseada em grupos fechados, a interação dos clientes com o servidor replicado se daria da seguinte forma: o cliente que deseja se comunicar com o servidor faz uma RPC, indicando a porta do servidor. Embora todas as réplicas escutem a mesma porta, cada uma delas tem seu próprio endereço FLIP privado. A camada RPC difunde então uma mensagem para descobrir o(s) endereço(s) FLIP correspondente(s) àquela porta, o qual é respondido por todos os núcleos que executam réplicas daquele servidor. Uma vez que cada réplica possui um endereço FLIP diferente, a cada resposta recebida, é criada uma entrada na tabela de roteamento mantida pela camada RPC. Quando um pedido deve ser enviado para uma dada porta, a camada RPC pesquisa sua tabela e escolhe um dos endereços FLIP, para onde o pedido é encaminhado. A camada FLIP, por sua vez, mantém uma tabela que roteia endereços FLIP em endereços de rede (físicos), que é o tipo de endereço que pode ser efetivamente usado na transmissão via rede de comunicação. A cada mensagem recebida, incluindo a resposta à mensagem difundida pela camada RPC, a camada FLIP atualiza sua tabela mapeando o endereço FLIP da mensagem no endereço de rede do seu transmissor (este processo será melhor detalhado mais adiante nesta seção).

Nesta configuração, apenas uma réplica recebe o pedido, ficando encarregada de distribuí-lo para as demais réplicas do seu grupo. Neste caso, a disseminação é feita

usando um endereço FLIP que identifica todos os processos do grupo. Na tabela de roteamento mantida pelo FLIP, este endereço é mapeado para os endereços de rede de todas as máquinas (digamos n) que hospedam réplicas do grupo. Se a rede física oferece a facilidade de difusão restrita, a camada FLIP pede à camada dependente da rede física para criar um endereço de difusão restrita para as n localizações. Caso contrário, difusão irrestrita é usada, se a rede possui esta capacidade, ou a difusão restrita é simulada com n mensagens ponto-a-ponto.

No Amoeba, se múltiplos processos tentam criar um grupo com a mesma porta, nenhum erro é retornado. Quando um pedido é endereçado para aquela porta um dos endereços FLIP associados é escolhido aleatoriamente, para o qual o pedido é encaminhado. O processo que recebe este pedido, retransmite-o usando o endereço FLIP do seu grupo. Desta forma, apenas os membros do grupo ao qual o processo que recebeu o pedido pertence recebem tal pedido.

Embora a escolha de usar grupos fechados tenha levado em conta a transparência oferecida para as aplicações clientes, ela vai de encontro ao nosso objetivo de tornar a replicação o mais transparente possível para o programador da aplicação servidora à medida que requer que a aplicação implementando o servidor seja codificada usando as primitivas de comunicação em grupo listadas acima. Cabe então ao ambiente operacional Seljuk-Amoeba prover facilidades que atendam aos requisitos de comunicação em grupo e transparência tanto a nível de cliente quanto a nível de servidor, simultaneamente.

7.5.2. Comunicação em Grupo no Seljuk-Amoeba

A transparência a nível de cliente é garantida como no Amoeba: toda comunicação cliente-servidor, seja ela um-para-um ou um-para-muitos, é feita, aparentemente, da mesma forma - via RPC. A diferença da nossa proposta é que, se o servidor é implementado por um grupo de réplicas, o próprio ambiente se encarrega de distribuir o pedido do cliente para todo o grupo, eliminando a necessidade de intervenção de algum dos seus membros (aquele que, na configuração atual do Amoeba, intermedia a comunicação do cliente com os demais membros do grupo). Como no Amoeba, a resposta do servidor para o pedido do cliente é também enviada para este usando a interface RPC.

A função da camada de comunicação em grupo será ampliada para realizar novas atividades de gerência de grupo, como por exemplo a verificação do tipo de replicação adotado por determinado grupo. Todo pedido de um cliente para um servidor replicado passa pela camada de comunicação em grupo na máquina cliente e na máquina servidora, e toda resposta de um servidor replicado para um cliente passa pela camada de comunicação em grupo da máquina servidora. Por questões de generalidade e compatibilidade, a interface para comunicação em grupo atualmente oferecida pelo Amoeba será também mantida no Seljuk-Amoeba.

A implementação da comunicação em grupo no Seljuk-Amoeba aproveita o protocolo de difusão atômica que forma a base da implementação no Amoeba [Kaashoek-Tanenbaum 94]: quando uma mensagem é enviada para o grupo, a mensagem é corretamente entregue a todos os membros do grupo, mesmo que a rede física possa perder alguns pacotes ou que algumas máquinas possam parar de funcionar.

Quando um grupo de servidores é criado, a ele é atribuído uma porta e um endereço FLIP; um cliente endereça pedidos para um grupo de servidores pelo número da porta que o grupo escuta. Como no Amoeba, os pacotes passados pela camada de comunicação em grupo, bem como pela camada RPC, para a camada FLIP são endereçados por endereços FLIP e os pacotes passados pela camada FLIP para a rede de comunicação são endereçados por endereços de rede. As camadas de comunicação em grupo e RPC mantêm tabelas que associam cada porta ao(s) endereço(s) FLIP correspondente(s); a camada FLIP, por sua vez, mantém uma tabela de roteamento que associa um endereço FLIP a um ou mais endereços físicos. Esta associação de 1-para-*n* endereços ocorre, por exemplo, quando o endereço FLIP em questão identifica um grupo de servidores replicados, onde cada servidor é executado numa diferente máquina do sistema e, portanto, possui um endereço de rede diferente dos demais servidores.

As tabelas mantidas pela camada RPC no Amoeba devem ser modificadas para indicar se a porta em questão é atualmente atendida por um único servidor (ou vários servidores independentes) ou por um grupo de servidores replicados. Um campo *type* é acrescentado às entradas da tabela para este fim. A camada de comunicação em grupo, por sua vez, mantém informações sobre todos os grupos existentes no sistema, incluindo a porta, o endereço FLIP e o tipo de replicação utilizados pelo grupo. Vejamos como se dá a comunicação em cada um dos lados da comunicação cliente-servidor.

a) Lado Cliente

Quando um cliente faz uma chamada RPC para o servidor (normalmente, por meio de uma rotina *stub* que chama a primitiva RPC *trans*), a camada RPC verifica em sua tabela se há alguma entrada para a porta indicada na chamada; em caso negativo, uma mensagem é difundida de forma irrestrita (a nível de FLIP) perguntando o endereço FLIP de quem escuta atualmente aquela porta. Dependendo do número de respostas obtidas, três diferentes ações podem ser tomadas:

(i) Se apenas um processo responde, uma única entrada é criada na tabela com *type* = "servidor" e o pedido é repassado para a camada FLIP com o endereço FLIP recebido.

(ii) Se vários processos respondem com endereços FLIPs diferentes, o que ocorre quando vários servidores independentes escutam a mesma porta, uma entrada para cada resposta é criada na tabela, também com *type* = "servidor"; a camada RPC escolhe uma das entradas e endereça o pedido para a camada FLIP de acordo.

(iii) Se mais de um processo responde com o mesmo endereço FLIP, uma única entrada na tabela é criada com *type* = "grupo de servidores" e o pedido do cliente é repassado para a camada de comunicação em grupo. Esta se encarrega de executar o protocolo de difusão atômica que garante a entrega do pedido a todos os membros do grupo que escutam a porta indicada, repassando para a camada FLIP o pedido identificado pelo endereço FLIP do grupo.

Originalmente no Amoeba, os diferentes processos de um grupo respondem à mensagem difundida pela camada RPC com os seus endereços FLIP individuais. No Seljuk-Amoeba, tal resposta passará a conter o endereço FLIP do grupo. Portanto, várias respostas com o mesmo endereço FLIP poderão chegar à camada RPC (caso iii).

Como explicado para o Amoeba, em todos os casos, quando os pacotes que contêm a resposta para a mensagem difundida pela camada RPC (como qualquer outro pacote) passam pela camada FLIP, esta cria uma entrada em sua tabela de roteamento para cada endereço FLIP recebido, mapeando o(s) endereço(s) de rede correspondente(s). No caso (i), uma única entrada é criada e um único endereço de rede

é associado; no caso (ii) várias entradas são criadas e a cada uma delas um único endereço de rede é associado; e no caso (iii) uma única entrada é criada, mas esta mapeia vários endereços de rede. Neste último caso, se os vários endereços pertencem a mesma rede local e a rede de comunicação oferece a facilidade de difusão restrita, a camada FLIP solicita a criação de um endereço de difusão restrita que atenda a todos os endereços recebidos.

Quando os pedidos são passados da camada RPC ou da camada de comunicação em grupo para a camada FLIP, esta última pesquisa em sua tabela de roteamento a existência do endereço FLIP indicado no pedido. Se o endereço não está correntemente na tabela (endereços que passam muito tempo sem ser referenciados são retirados da tabela para liberar espaço), a camada FLIP age da mesma forma que a camada RPC, difundindo de forma irrestrita uma mensagem para tentar descobrir o(s) endereço(s) de rede correspondente(s) àquele endereço FLIP. Quando a(s) resposta(s) chega(m), a tabela é atualizada de acordo. Duas situações podem ocorrer: (i) se uma única resposta chega, o endereço FLIP é mapeado para um único endereço de rede; e (ii) se várias respostas chegam, o endereço FLIP é mapeado em vários endereços de rede e um endereço de difusão restrita pode ser solicitado à rede de comunicação, como explicado anteriormente.

b) Lado Servidor

Pedidos dos clientes são recebidos pelo servidor através da interface RPC (primitiva *getrequest*), do mesmo modo como funciona atualmente no Amoeba, só que agora todos os servidores do grupo receberão os pedidos[10] diretamente dos clientes, eliminando a necessidade de um servidor ficar responsável por recebê-los e repassá-los para todos os outros servidores do seu grupo. Um dos parâmetros da primitiva *getrequest* é um cabeçalho, que contém o número da porta que o servidor individual escuta.

Números de portas podem ser escolhidos pelo programador da aplicação ou podem ser gerados pelo núcleo do sistema operacional. Se o programador prefere utilizar um porta pré-definida, todos os servidores do grupo escutarão a mesma porta e, quando o grupo é criado, a ele é atribuído o mesmo valor da porta (como veremos na

[10] No caso da replicação passiva, apenas um servidor (aquele representado pela réplica primária) faz a chamada *getrequest* e, portanto, apenas ele recebe efetivamente o pedido.

próxima seção). O programador, porém, pode pedir ao núcleo para gerar um número de porta aleatório para o servidor, através da primitiva *uniqport*. Neste caso, como cada servidor do grupo fará sua própria chamada a *uniqport*, há uma grande probabilidade de cada membro do grupo receber um número diferente de porta. Quando o grupo é criado, uma porta única é a ele atribuída (provavelmente diferente daquelas usadas pelos seus membros individuais) e, portanto, um mapeamento deve ser feito entre a porta individual de cada servidor e a porta do grupo. A camada RPC mantém uma tabela que mapeia a porta individual na porta do grupo. Servidores escutam mensagens na sua porta individual, mas clientes endereçam seus pedidos à porta do grupo. Assim, quando um pedido chega à camada RPC, ela deve pesquisar esta tabela para traduzir a porta do grupo na porta que o servidor local está efetivamente escutando, para então poder entregar o pedido. Se não há entrada na tabela para aquela porta de grupo, assume-se que o mesmo endereço está sendo usado como porta coletiva do grupo e como porta individual de todos os servidores do grupo; neste caso, o pedido é entregue na porta indicada no próprio pedido.

Quando uma resposta é enviada pelo servidor (primitiva *putreply*), a camada RPC primeiro verifica se aquele servidor é ou não replicado (esta informação é adicionada nas tabelas quando o grupo é criado). Em caso negativo, a resposta é repassada diretamente para a camada FLIP que se encarrega de enviá-la para o cliente. Caso contrário, a camada RPC traduz, se necessário, a porta individual do servidor na porta do grupo ao qual ele pertence e, então, repassa a resposta para a camada de comunicação em grupo por meio da nova primitiva *SendToRepGroup()*, que fica responsável por tomar as ações necessárias para enviar uma única mensagem correta do grupo para o cliente. A forma como isto é feito depende da técnica de replicação utilizada.

(i) Replicação ativa: como semântica de falha silenciosa é garantida pelo serviço de processamento confiável do Seljuk-Amoeba, todas as mensagens geradas são corretas, e, portanto, qualquer uma pode ser usada como resposta para o pedido do cliente. Porém, como discutido na Seção 3.6.2, apenas uma resposta deve ser efetivamente entregue ao cliente do serviço. A forma mais simples de garantir entrega de uma única mensagem ao cliente é permitir que todas as réplicas enviem suas respostas para o cliente e deixar a tarefa de selecionar a mensagem para o núcleo do receptor. Como todas as mensagens

enviadas possuem o mesmo identificador e o mesmo endereço de origem (porta), a camada RPC do cliente pode facilmente descartar as mensagens duplicadas. Esta solução minimiza o atraso na provisão de respostas (pois a mensagem seria recebida da réplica mais rápida) e garante o máximo de continuidade do serviço às custas de geração de carga extra na rede de comunicação. Uma alternativa, que reduz este tráfego extra, porém aumentando o tempo de provisão de respostas, é filtrar as mensagens de saída no transmissor. Neste caso, algum mecanismo deve ser usado para decidir qual entre as múltiplas cópias da mensagem deve ser enviada. Um critério simples para se tomar esta decisão é o seguinte: o núcleo que hospeda a réplica do servidor com menor identificador dentro do grupo envia a mensagem para o cliente; como todo núcleo mantém informação atualizada sobre a afiliação do grupo, é fácil saber qual deles deve enviar a mensagem. Porém, um problema pode ocorrer quando, quase simultaneamente à produção da resposta, a réplica de menor identificador no grupo ou o nodo que a hospeda falha (silenciosamente) e a resposta deixa então de ser enviada para o cliente (até que a falha seja percebida pelos outros membros do grupo e um deles passe a enviar as mensagens), o que comprometeria a continuidade do serviço que a técnica de replicação ativa visa prover. Para evitar isto, após a resposta para um pedido ter sido entregue com sucesso ao cliente, o núcleo que a enviou, informa aos demais núcleos (que hospedam membros do grupo) o sucesso da operação, por meio de uma pequena mensagem de notificação. Quando uma resposta para um pedido é repassada pela camada RPC para a camada de comunicação em grupo das outras réplicas do grupo, em vez de enviar a mensagem, esta aguarda o recebimento da mensagem de notificação do membro de menor identificador; se esta mensagem não chega dentro de um tempo preestabelecido, o próximo membro de menor identificador deve enviar a mensagem para o cliente e a notificação para o grupo; este procedimento é repetido até que pelo menos uma mensagem de notificação chegue em todos os membros do grupo[11]. O Seljuk-Amoeba provê primitivas que permitem a

[11] Mensagens de notificação podem se perder durante a transmissão e vários núcleos podem enviar a mesma mensagem, mas estas podem ser facilmente descartadas pelo receptor. É melhor gerar algumas mensagens repetidas do que deixar de garantir a continuidade do serviço.

escolha de uma das duas abordagens no momento da solicitação do serviço de replicação, conforme veremos na próxima seção.

(ii) Replicação Passiva: Como apenas um membro do grupo de servidores (a réplica primária) produz uma resposta e, portanto, faz a chamada RPC *putreply()*, a única camada de comunicação em grupo que é invocada pela camada RPC envia imediatamente a resposta para o cliente.

(iii) Replicação Semi-Ativa: A fim de tornar a replicação semi-ativa o mais transparente possível para a aplicação, em vez de apenas um servidor do grupo (representado pela réplica líder) enviar a resposta para o cliente, isto é, fazer o *putreply()*, todos o fazem e as respostas são filtradas pela camada de comunicação em grupo, que deixa passar apenas a mensagem de uma delas (neste caso, a líder do grupo), da mesma forma como explicado para replicação ativa. Se a réplica líder falha, nenhuma resposta é enviada entre o tempo da falha e o tempo da sua detecção; somente após uma outra réplica assumir o papel de primária, é que o provimento do serviço será continuado. Porém, como dissemos na Seção 3.8.1, se é desejável obter o máximo de continuidade de serviço e o mínimo de atraso na provisão de mensagens, uma alternativa é permitir que todas as réplicas enviem suas mensagens, as quais seriam filtradas pela camada RPC receptora, da mesma forma explicada para a replicação ativa. A utilização de uma ou de outra abordagem é também selecionada na solicitação do serviço de replicação do Seljuk-Amoeba.

O esquema do sistema de comunicação do Seljuk-Amoeba pode ser representado da forma mostrada na Figura 7.2.

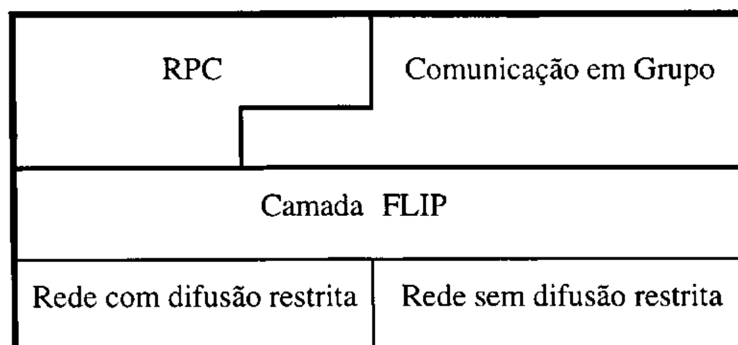


Figura 7.2: Camadas do Sistema de Comunicação do Seljuk-Amoeba

Do mesmo modo que as primitivas nativas do Amoeba, as primitivas a serem acrescentadas ao Seljuk-Amoeba, sumariamente apresentadas na Tabela 7.2, são bastante flexíveis, permitindo que se possa balancear desempenho contra requisitos de tolerância a faltas.

Primitiva	**Descrição**
CreateRepGroup	Cria um novo grupo com as características definidas pelo chamador. O chamador não passa a fazer parte do grupo.
JoinRepGroup	Adiciona novos membros ao grupo. Usada para propósito de reconfiguração do grupo.
SendToRepGroup	Envia atomicamente uma mensagem para todos os membros do grupo. O chamador não é membro do grupo.
ResetRepGroup	Inicia o processo de reorganização do grupo após a falha de algum membro ter sido detectada.

Tabela 7.2: Primitivas para Comunicação em Grupo do Seljuk-Amoeba

O funcionamento e a implementação destas novas primitivas são bastante semelhantes àquelas nativas do Amoeba, mas ao contrário delas, as novas primitivas são ativadas por um processo que não faz parte do grupo e que normalmente não é uma aplicação do usuário, mas sim um processo componente do próprio ambiente operacional Seljuk-Amoeba. Estas primitivas serão melhor compreendidas no decorrer do texto a seguir.

7.6. Serviço de Replicação

O serviço de replicação de processamento no Seljuk-Amoeba é oferecido por um servidor de replicação - o *Replicator* - implementado na camada *Middleware* da arquitetura Seljuk (vide Capítulo 1). A fim de criar um componente de *software* replicado, daqui por diante referenciado como processo ou servidor replicado, o *Replicator* é invocado por meio de uma primitiva do tipo:

Replicate*(program, port, diversity, resilience, failure-semantics, autoreconfiguration)*

Replicate[12] é, na verdade, uma rotina *stub* responsável por iniciar uma RPC com o *Replicator* (através da primitiva RPC *trans*), solicitando a criação de um processo replicado para executar o código indicado em *program*. O parâmetro *port* é opcional e é usado quando a aplicação deseja escutar uma porta pré-definida. O parâmetro *diversity* indica se diversidade de projeto deve ser considerada (com o intuito de obter tolerância a faltas de concepção do *software*); o grau de resiliência (isto é, o número de faltas a serem toleradas) é determinado por *resilience* e a semântica de falha real considerada para os componentes do sistema, incluindo a própria aplicação, é indicada em *failure-semantics*. Finalmente, o parâmetro *autoreconfiguration* informa se o ambiente deve ou não realizar a reconfiguração automática do processo replicado quando alguma réplica falha.

O grau de resiliência define o nível de tolerância a faltas do serviço oferecido pelo servidor replicado e, juntamente com a semântica de falha assumida, determina o número de réplicas necessárias para obter o nível desejado. Como o *Replicator* usa a abstração de nodos com semântica de falha silenciosa oferecida pelo serviço de processamento confiável do próprio Seljuk-Amoeba, se o grau de resiliência é definido por k, $k + 1$ réplicas devem ser criadas.

Ao receber o pedido emitido por meio da primitiva *Replicate*, o *Replicator* procura o arquivo *program* no diretório /bin (diretório onde os binários são armazenados no Amoeba e, por compatibilidade, no Seljuk-Amoeba) ou, se *program* inclui um caminho completo, procura no diretório indicado neste parâmetro. Se o programa indicado em *program* estiver disponível para várias arquiteturas, o resultado da busca é um diretório, e não um arquivo propriamente dito. Para cada arquitetura, por sua vez, podem existir várias versões do mesmo programa. Portanto, dentro do diretório do programa haverá: (i) arquivos binários para as arquiteturas que só possuem uma versão do programa; e (ii) um arquivo padrão e um subdiretório para cada arquitetura que possui mais de uma versão do programa; nestes subdiretórios, estão as várias outras versões do programa para aquela arquitetura em particular. A Figura 7.3, extraída de [Gallindo-Brasileiro 97], mostra um exemplo da organização atual do diretório padrão onde se encontram os arquivos executáveis no Seljuk-Amoeba (/bin).

---

[12] *Replicate* é aqui usado como um nome genérico para as diversas primitivas oferecidas pelo *Replicator*, que serão introduzidas quando falarmos da implementação do serviço replicado para cada técnica de replicação em particular, um pouco mais adiante. No momento oportuno, os parâmetros dependentes da primitiva em particular serão acrescentados aos aqui apresentados, que são comuns a todas as primitivas.

Neste exemplo, o diretório /bin contém dois comandos: *dir* e *sort*. O comando *dir* está disponível apenas para a arquitetura VAX. Já o comando *sort* encontra-se disponível para a arquitetura 80386 em várias versões: o arquivo pd.i80386 representa a versão padrão do *sort* para tal arquitetura e dentro do diretório i80386 estão mais três implementações do comando para esta mesma arquitetura. Esta diversidade de implementações para um mesmo programa permite a obtenção de tolerância a faltas de concepção tanto no *hardware* quanto no *software*, à medida que possibilita que as diferentes réplicas do programa sejam executadas em arquiteturas diferentes e a partir de versões diferentes do programa.

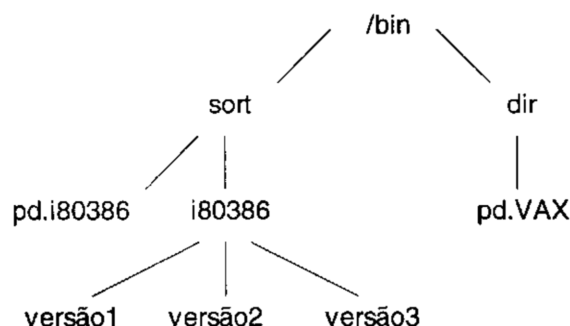


Figura 7.3: Sistema de Arquivos do Seljuk-Amoeba

Terminada a busca, o *Replicator* deve requisitar ao servidor de execução do ambiente operacional (*FT Run Server*) a alocação de unidades de processamento para executar o programa desejado, passando para este a lista dos descritores de processos para todas as versões disponíveis do programa (*process-descriptors*). Se a semântica de falha definida em *failure-semantics* é menos restritiva do que a semântica de falha silenciosa, o *Replicator* deve requisitar ao *FT Run Server* a criação de nodos com semântica de falha silenciosa. O *Replicator* faz tantas chamadas ao *FT Run Server* quantas forem necessárias para criar um número de nodos suficiente para executar todas as réplicas necessárias, ou seja, *resilience* + 1 chamadas são feitas. Cada um destes nodos, por sua vez, é formado por um par de processadores com semântica de falha *failure-semantics*, de modo que, supondo-se que os dois processadores (ou as réplicas executando neles) não falham simultaneamente, se um deles falha, esta falha é detectada e o nodo simplesmente pára[13]. Desta forma, 2(*resilience* + 1) processadores são necessários para prover o grau de resiliência desejado.

[13] A partir de um par de processadores com semântica de falha qualquer é possível construir um nodo com semântica de falha silenciosa, desde que apenas um destes processadores falhem a cada instante.

Observe que, ao custo de um único processador extra, podemos utilizar o serviço de processamento confiável oferecido pelo *FT Run Server* e, assim, simplificar os protocolos para tolerância a faltas de alto nível, pois, se fôssemos tratar diretamente falhas arbitrárias neste nível, *2resilience* + 1 processadores seriam necessários e os protocolos seriam bem mais complexos. Além do mais, a maioria dos protocolos só trabalham com falhas silenciosas, como é o caso das técnicas de replicação passiva e semi-ativa, e não podem ser utilizados se esta restrição não puder ser atendida.

A chamada remota feita pelo *Replicator* para o *FT Run Server* possui os seguintes parâmetros:

- uma lista dos descritores de processos para todas as versões disponíveis do programa: ***process-descriptors***;
- a semântica de falha real dos processadores do sistema: ***failure-semantics***;
- a semântica de falha requerida para o nodo a ser criado: **falha silenciosa**;
- o fator de replicação do nodo, isto é, quantos processadores independentes formarão o nodo: **zero, se *failure-s*emantics = falha silenciosa; dois, caso contrário;**
- uma indicação se diversidade de projetos deve ser ou não usada: ***diversity***;
- uma lista das versões já utilizadas nas chamadas anteriores; e
- uma lista dos processadores que não se deseja utilizar na execução daquela aplicação: **aqueles já usados nas chamadas anteriores** (se houver).

Se diversidade de *software* deve ser empregada, o *FT Run Server* vai procurar diversificar ao máximo a escolha das arquiteturas de processadores e das versões do programa a serem usadas, para tanto a lista das versões já utilizadas anteriormente deve ser conhecida. Com isto, pode-se conseguir tolerância a faltas de concepção tanto de *hardware* quanto de *software*, já que cada réplica poderá ser caracterizada por um par (arquitetura do processador, versão do programa) diferente das demais.

O último parâmetro repassado para o *FT Run Server* - a lista de todos os processadores já utilizados na composição dos nodos criados pelas chamadas anteriores - evita que mais de uma réplica do mesmo processo sejam executadas em um mesmo processador, o que levaria a diminuição do grau de resiliência daquele processo, contrariando assim o nosso propósito.

A chamada ao *FT Run Server* bloqueia o *Replicator* e, quando realizada com sucesso, retorna para este um descritor do nodo criado, que contém informações como, um identificador para o nodo, uma lista dos processadores que compõem o nodo (o primeiro processador da lista é o coordenador do nodo) e as versões escolhidas para executar em cada processador do nodo. Do mesmo modo que o *Run Server* do Amoeba, o *FT Run Server* constrói nodos a partir de processadores que estejam com menor carga de processamento no momento. Além disso, o *FT Run Server* leva em consideração o parâmetro *diversity*, alocando processadores que permitam o emprego de diversidade de projeto, se requerido. Dessa forma, há um balanceamento entre melhor desempenho e maior tolerância a faltas que deve ser considerado no momento de decidir se diversidade de projetos deve ou não ser empregada, pois nem sempre os processadores com menor carga no momento da alocação são de arquiteturas diferentes ou versões diversas para o programa estão disponíveis para ele.

Quando a aplicação já considera uma semântica de falha silenciosa para a infra-estrutura de processamento, a criação de nodos não é necessária. Neste caso, o *FT Run Server* é responsável apenas por selecionar para a aplicação *resilience* + 1 processadores, indicando, mais uma vez, aqueles que disponham de melhor condição para executar as réplicas do processo e considerando diversidade de projetos, se solicitado.

O fato de uma réplica está sendo executada em um processador único ou em um nodo, que é um conjunto de processadores, é indiferente para o nosso nível de abstração, já que o ambiente Seljuk-Amoeba provê o serviço de processamento confiável, implementado pelo conceito de nodos, de forma transparente para os usuários deste serviço [Gallindo-Brasileiro 97]. Sob o ponto de vista do *Replicator*, o comportamento de um nodo é equivalente ao comportamento de um processador que possua a mesma semântica de falha implementada pelo nodo. Por questões de generalidade, daqui por diante, adotaremos o termo nodo para referenciar tanto um processador quanto um conjunto deles.

Se todos os nodos forem alocados com sucesso[14], o *Replicator* pode prosseguir; caso contrário, um erro é retornado para o chamador. No primeiro caso, é necessário agora que se forme um grupo com todas as réplicas do processo para que escutem uma mesma porta e possam interagir a fim de proverem o serviço adequado. Portas válidas só

[14] Um nodo pode não ser alocado devido a restrições na disponibilidade de recursos.

podem possuir valores positivos. Se o parâmetro *port* da chamada *Replicate* traz um valor válido, o grupo a ser criado deve usar este valor como a porta do grupo; por outro lado, se o parâmetro traz um valor negativo, assume-se que cada réplica da aplicação definirá a sua própria porta em tempo de execução (através da primitiva do núcleo *uniqport*) e o *Replicator* pede também ao núcleo a criação de uma porta, que será associada ao grupo. O mapeamento (porta do grupo, porta da réplica) é feito pela camada RPC como explicado na seção anterior.

Além disso, o *Replicator* solicita ao núcleo a criação de um endereço FLIP a ser usado pelo grupo na troca de mensagens a nível da camada FLIP. Este endereço deverá ser registrado em todos os nodos que hospedam algum membro do grupo. Neste ponto, o grupo pode ser efetivamente criado, o que é feito pelo *Replicator* por meio da nova primitiva de gerência de grupo oferecida pelo Seljuk-Amoeba:

CreateRepGroup*(port, address, resilience, rep-type, node-list)*

Esta chamada cria um grupo de servidores que deverão receber todas as mensagens enviadas para a porta *port*, a qual deverá ser mapeada para o endereço FLIP *address*. Ou seja, quando a camada RPC de um cliente difundir uma mensagem perguntando quem escuta a porta *port* (vide Seção 7.5), todos os membros do grupo devem responder com *address*. O número de faltas que o grupo deve ser hábil a tolerar é indicado por *resilience* e o tipo de replicação a ser usado é definido por *rep-type*. O valor de *rep*-type depende da primitiva chamada no momento da solicitação do serviço. O Replicator possui uma tabela que relaciona cada primitiva com o tipo de replicação por ela adotado. Por fim, o parâmetro *node-list* indica os nodos onde as diversas réplicas devem ser alocadas. *CreateRepGroup* retorna um descritor de grupo, a ser utilizado para identificar o grupo em chamadas posteriores.

Um *objeto de configuração* é então criado para guardar informações sobre o grupo e seus membros, tais como: a lista dos nodos que hospedam membros do grupo, o grau de resiliência do grupo, a porta que o grupo escuta e o endereço FLIP do grupo. Tal objeto guarda ainda informações sobre cada membro do grupo, como por exemplo: o identificador do membro dentro do grupo (índice) e um endereço FLIP individual para o membro. Todas estas informações devem ser mantidas em cada um dos nodos do grupo para que os protocolos de gerência e comunicação em grupo possam fazer uso delas sempre que necessário. Um endereço FLIP para cada membro é gerado pelo núcleo

durante a execução de *CreateRepGroup* e é utilizado pelo protocolo de difusão atômica, no qual toda a comunicação em grupo se apóia, para, por exemplo, reenviar para um único membro uma mensagem perdida durante a transmissão; não faz sentido reenviar para todo o grupo a mensagem que apenas um membro perdeu. Feito isto, o *Replicator* solicita ao serviço de processamento de cada um dos nodos a criação de um processo para executar a versão escolhida de *program*. Um dos parâmetros desta chamada é o descritor do grupo, por meio do qual todas as informações sobre o grupo e os seus membros podem ser acessadas. Por exemplo, quando o processo criado fizer uma chamada *getrequest* na sua porta individual, o núcleo, a nível da camada RPC, extrai do objeto de configuração do grupo identificado pelo descritor recebido a porta do grupo e faz o respectivo mapeamento. Assim, sempre que um pedido chegar para a porta do grupo, ele será entregue ao processo correspondente, embora este esteja escutando sua porta local (vide Seção 7.5).

O serviço de processamento é oferecido pelo *servidor de processos*, no caso de um único processador, ou pelo *servidor de nodos* do processador coordenador do nodo [Gallindo-Brasileiro 97], no caso de um nodo propriamente dito. Ambos os servidores fazem parte do núcleo que executa em cada processador do Amoeba. A Figura 7.4 mostra o cenário da criação de um processo replicado no Seljuk-Amoeba, quando nodos são usados.

Tendo concluído o trabalho de criação dos processos, o *Replicator* retorna para o chamador um código indicando o sucesso da operação. Note que a aplicação em si não toma conhecimento da existência do grupo criado, ou seja, em seu código não há qualquer referência ao grupo.

Neste ponto, as réplicas começam efetivamente a executar. O funcionamento do processamento replicado depende da técnica de replicação usada, como veremos a seguir.

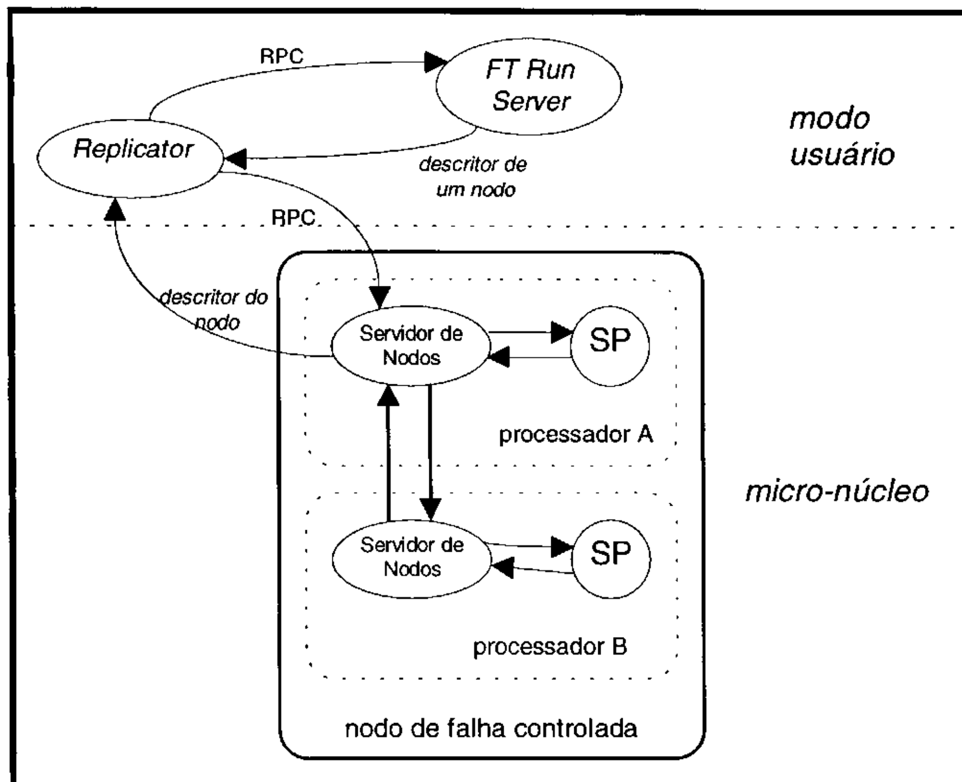


Figura 7.4: Replicação de Processos no Seljuk-Amoeba

7.6.1. Serviço de Replicação Ativa

Na replicação ativa, cada uma das réplicas processa todas as mensagens destinadas ao processo replicado e provê mensagens de saída. Ou seja, todas as réplicas executam num modo ativo. Desta forma, mensagens de saída duplicadas são inevitavelmente geradas. Porém, como foi discutido na Seção 7.5, estas mensagens são descartadas pelo serviço de comunicação da máquina receptora ou, alternativamente, são filtradas pela camada de comunicação em grupo da máquina transmissora. A primeira opção provê melhor grau de continuidade de serviço às custas de tráfego extra na rede, enquanto a segunda diminui o tráfego na rede, mas também diminui o grau de continuidade do serviço.

A primitiva oferecida pelo *Replicator* para solicitação de processamento replicado ativo possui um parâmetro extra (*continuity*) que indica se o grau de continuidade do serviço desejado deve ser maximizado ou não. No primeiro caso, a camada de comunicação em grupo de todas as réplicas envia todas as mensagens de saída produzidas; no último caso, a camada de comunicação em grupo de apenas uma das

réplicas envia, *a priori*, respostas para o cliente (vide Seção 7.5). A primitiva genérica *Replicate*, introduzida no início desta seção, tomaria a seguinte forma:

ActiveReplicate*(program, port, diversity, resilience, failure-semantics, autoreconfiguration, continuity)*

Exceto o parâmetro *continuity*, cujo objetivo foi explicado acima, todos os demais parâmetros têm o mesmo significado explicado para a primitiva genérica *Replicate*.

No Seljuk-Amoeba, qualquer aplicação distribuída que utilize apenas troca de mensagens na comunicação e que tenha um comportamento determinístico pode ser replicada de forma ativa de maneira completamente transparente, isto porque o ambiente provê suporte para execução replicada mesmo para aquelas aplicações que não foram projetadas para serem executadas de forma replicada (desde que elas atendam às restrições mencionadas acima). Nenhuma alteração no código fonte da aplicação precisa ser realizada. Na verdade, ela sequer precisa ser recompilada.

7.6.2. Serviço de Replicação Passiva

Como vimos, na replicação passiva, apenas uma réplica (a *primária*) processa as mensagens de entrada e provê as mensagens de saída. As demais réplicas (as *suplentes*) são passivas visto que, na ausência de faltas, elas não executam qualquer processamento, exceto a atualização de seus estados internos. Uma aplicação a ser replicada passivamente deve, portanto, ser implementada de forma que ela possa operar num *modo ativo*, executando efetivamente o processamento, ou num *modo passivo*, onde ela simplesmente recebe informações sobre a atualização de estado. Durante sua execução, a réplica do processo pode reverter do modo passivo para o ativo, o que significa que ela deixou de ser suplente e passou a ser primária. Deve haver, portanto, uma eleição entre as réplicas para decidir qual delas será primária do seu grupo. No Seljuk-Amoeba, esta eleição é realizada pelo próprio núcleo, sem que a aplicação precise tomar conhecimento dela.

Uma forma de implementar no Seljuk-Amoeba o processo a ser replicado passivamente é, logo no início da execução, criar duas linhas de execução: uma para executar no modo ativo e outra para executar no modo passivo. A primeira ação da linha de execução ativa é verificar se a réplica é primária ou suplente. Se a réplica é suplente, a linha de execução fica bloqueada até que ela se torne primária (se isto vier a ocorrer). Se

ela é primária, caso haja algum estado para ser recuperado, a ação de atualização de estado é realizada. Em seguida, a linha de execução ativa começa efetivamente a processar os pedidos recebidos dos clientes e a enviar suas respostas. De tempos em tempos, esta linha de execução deve também emitir as salvaguardas que seriam recebidas pelas linhas de execução passivas das outras réplicas. De fato, a única função da linha de execução passiva é receber as salvaguardas enviadas pela linha de execução ativa da réplica primária.

Enquanto a linha de execução ativa fica bloqueada nas réplicas suplentes, a linha de execução passiva fica bloqueada na réplica primária, uma vez que ela não recebe nenhuma mensagem de salvaguarda e fica esperando por uma mensagem que nunca chega. Isto garante que cada réplica opere em apenas um modo (ativo ou passivo) por vez e, mais, que apenas a primária opere em modo ativo, enquanto todas as outras operam no modo passivo.

Os pedidos dos clientes são entregues à camada de comunicação em grupo de todas as réplicas, mas são repassados apenas para a réplica primária, uma vez que apenas ela faz a chamada *getrequest*. Quando a falha da réplica primária é detectada pelo serviço de detecção (a ser detalhado na Seção 7.7), o núcleo elege uma nova primária entre as suplentes e desbloqueia a linha de execução ativa da réplica eleita, que começa então a receber os pedidos dos clientes e fornecer respostas para eles.

O Seljuk-Amoeba oferece uma função padrão para eleição da réplica primária: aquela réplica que possuir menor identificador dentro do grupo é eleita. Porém, o ambiente oferece a flexibilidade do programador escolher sua própria função de eleição. Neste caso, o nome da função é passada como parâmetro (*election*) na primitiva que invoca o serviço de replicação passiva ao *Replicator*, que tomaria a seguinte forma:

PassiveReplicate*(program, port, diversity, resilience, failure-semantics, autoreconfiguration, election)*

Os demais parâmetros têm o mesmo significado explicado para a primitiva genérica *Replicate*, no início desta seção.

Embora a replicação em si não seja de todo transparente para o programador, visto que ele deve implementar cada uma das linhas de execução, a gerência do grupo e a comunicação em grupo são tratadas (e garantidas) pelo ambiente operacional. Além disso, o Seljuk-Amoeba provê uma biblioteca de funções a serem usadas na

implementação de processos seguindo os modelos de replicação passiva e semi-ativa, o que já reduz bastante o trabalho do programador.

No caso da replicação passiva, três funções são oferecidas. A primeira delas, denominada *OnPrimaryUnblock*() é invocada pela linha de execução ativa e não retorna até que a réplica local se torne primária, o que pode eventualmente vir a ocorrer. Ou seja, o que esta função faz é simplesmente bloquear a linha de execução ativa das réplicas suplentes.

As outras duas funções são oferecidas para provimento de salvaguardas da réplica primária para as suplentes: *Checkpoint()* e *GetCheckpoint()*. A linha de execução ativa da réplica primária chama a função *Checkpoint()* passando como parâmetro uma função de empacotamento, responsável por compor um "pacote" que reflita o estado interno atual da réplica primária. *Checkpoint()* invoca a função de empacotamento, compõe uma mensagem contendo o pacote resultante e o número de seqüência da última mensagem processada pela réplica, e a envia para as réplicas suplentes, fazendo uso do serviço de comunicação em grupo confiável oferecido pelo Seljuk-Amoeba. Esta mensagem é recebida por meio da chamada *GetCheckpoint()* na linha de execução passiva das réplicas suplentes e, quando uma réplica suplente se torna primária, tal mensagem é passada, pela linha de execução ativa, como parâmetro para uma função de desempacotamento, capaz de decompor o pacote contido na mensagem e atualizar o estado da réplica a partir da informação nele contida. O espaço de endereçamento de um processo é compartilhado por todas as suas linhas de execução; por isto a mensagem recebida na linha de execução passiva pode ser acessada pela linha de execução ativa.

As funções de empacotamento e desempacotamento são específicas da aplicação, devendo ser implementadas pelo próprio programador da aplicação. Fica também sob responsabilidade deste definir a estratégia de salvaguarda a ser usada, considerando a quantidade de retrocesso necessário quando uma suplente deve assumir o processamento *versus* a carga de comunicação gerada pela freqüência da emissão de salvaguardas, conforme discutido na Seção 3.7. De qualquer modo, uma salvaguarda deve ser obrigatoriamente emitida sempre que um processamento não-determinístico é realizado.

A fim de evitar que a réplica suplente, em caso de falha da réplica primária, solicite o reenvio das mensagens de entrada já processadas por esta, quando um pedido para o grupo é recebido por um núcleo que executa uma réplica suplente, em vez de repassar o

comunicação em grupo do Seljuk-Amoeba garante entrega atômica, as mensagens a serem processadas (isto é, os pedidos dos clientes) são enviadas diretamente do cliente para todas as réplicas do servidor, não havendo necessidade da líder ditar a ordem de processamento das mensagens, como sugere o modelo padrão, o que torna a comunicação transparente para a aplicação.

Como na replicação passiva, o núcleo deve executar um algoritmo de eleição no início do processamento para decidir qual réplica será a líder. Em caso de falha da réplica líder, uma nova eleição deve ser realizada para selecionar uma nova líder entre as seguidoras. Do mesmo modo, o Seljuk-Amoeba oferece a mesma função padrão para eleição usada na replicação passiva, mas também permite que a aplicação defina sua própria função.

Como para a replicação ativa, o Seljuk-Amoeba oferece também as duas abordagens para disseminação das mensagens de saída apresentadas na Seção 7.5: no primeiro caso, apenas a réplica líder propaga mensagens de saída; e, no segundo caso, todas propagam e as duplicatas são descartadas pelo receptor. Esta última abordagem provê melhor grau de continuidade de serviço às custas de tráfego extra na rede. A escolha entre uma e outra é, também nesta técnica, determinada pelo parâmetro *continuity* da primitiva para solicitação do serviço de replicação semi-ativa.

A primitiva para solicitação do serviço de replicação semi-ativa teria então a seguinte forma:

SemiActiveReplicate*(program, port, diversity, resilience,*
failure-semantics, autoreconfiguration, continuity, election)

Como descrito acima, o parâmetro *continuity* tem o mesmo significado daquele usado na primitiva *ActiveReplicate* e parâmetro *election* é como explicado na *PassiveReplicate*. Como em ambas as primitivas, todos os demais parâmetros têm o mesmo significado explicado para a primitiva genérica *Replicate*.

Como as mensagens são recebidas pelas diversas réplicas do servidor diretamente do cliente, quando não há processamento não-determinístico, as seguidoras prosseguem de forma independente da líder. Esta diferença de sincronização entre as réplicas permite, portanto, que as seguidoras possam prossiguir em algum momento da execução um pouco à frente da líder. Se é adotada a abordagem em que apenas a réplica líder propaga

mensagens e se, no momento em que a réplica primária falha, a seguidora que assume seu papel está em um ponto mais adiantado da execução, algumas mensagens de saída podem deixar de ser disseminadas. Para evitar isto, todo núcleo que executa uma réplica seguidora guarda todas as mensagens por ela produzidas e o núcleo que executa a réplica líder envia, de tempos em tempos, uma mensagem especial notificando as suas seguidoras sobre o número seqüencial da última mensagem por ela disseminada. Quando uma mensagem de notificação enviada pela líder chega, as mensagens no diário de saída que possuem identificador menor ou igual aquele indicado pela notificação são eliminadas da fila, pois elas não precisam mais ser propagadas caso a seguidora assuma o papel de líder.

Quando uma seguidora assume o papel de líder, o diário de mensagens de saída pode se encontrar em duas situações: (a) o diário está vazio, o que indica que a líder estava à frente da seguidora ou que a seguidora e a líder estavam aproximadamente sincronizadas; ou (b) o diário não está vazio, o que indica que aparentemente a seguidora estava processando mais rápido que a líder.

Assim, quando uma seguidora passa a ser líder, o núcleo que a executa deve verificar se o diário está vazio ou não; se não está, ele primeiro envia todas as mensagens no diário, para só depois começar a enviar as novas mensagens produzidas pela nova líder. Por outro lado, se o diário está vazio, ele deve verificar qual foi o valor indicado na última mensagem de notificação da antiga líder e só enviar, dentre as novas mensagens produzidas, aquelas que tenham um número de seqüência maior que aquele indicado na mensagem de notificação. Este último procedimento, embora não seja imprescindível, é interessante ser adotado à medida que permite a redução do tráfego extra na rede gerado por mensagens duplicadas.

Note que as mensagens enviadas pela réplica líder entre o envio de uma mensagem de notificação e a sua falha seriam novamente enviadas pela nova líder, mas o próprio receptor destas mensagens se encarrega de descartar as que forem duplicadas.

Como visto no Capítulo 3, uma das principais vantagens da técnica de replicação semi-ativa é que ela resolve o problema da divergência de estado entre as réplicas, gerado pelo potencial comportamento não-determinístico que a aplicação pode apresentar, fazendo com que a réplica líder tome as decisões que afetam o determinismo e comunique suas decisões para suas seguidoras.

A fim de prover suporte para a solução do não-determinismo, o Seljuk-Amoeba provê três funções de biblioteca, que são chamadas quando um processamento não-determinístico deve ser realizado. A primeira função, denominada *WhatsMyRole()*, retorna para a réplica um valor indicando se ela é líder ou seguidora. Caso o retorno da função indique que a réplica é líder, ela executa o processamento não-determinístico e comunica o resultado às demais réplicas do seu grupo por meio da função *Notify()*. Quando a réplica é seguidora, ao invés de executar o processamento, ela espera o resultado enviado pela líder, chamando a função *GetNotify()*, e usa este resultado para dar prosseguimento à sua execução. Observe que apesar de o programador ter que inserir estas chamadas no código da aplicação, a tarefa de garantir a entrega das mensagens aos membros do grupo fica por conta do ambiente operacional. A aplicação sequer precisa tomar conhecimento sobre a afiliação ou endereço do grupo. Subseção

A próxima subseção mostra o esboço de uma aplicação replicada de forma semi-ativa cujo não-determinismo é resolvido usando as funções de biblioteca oferecidas pelo Seljuk-Amoeba para tal fim.

Embora a solução apresentada acima seja a forma mais natural de tratar o não-determinismo no modelo de replicação semi-ativa (onde o líder dita o resultado), o programador está livre para adotar uma outra solução, na qual a aplicação deve ser projetada de forma que toda operação que apresente comportamento não-determinístico tenha seu resultado enviado através de uma mensagem para a própria aplicação. Uma vez que todas as réplicas da aplicação escutam a mesma porta, esta mensagem seria, *a priori*, recebida por todas elas. Como todas as réplica seguem a mesma implementação, várias mensagens serão produzidas. Estas mensagens serão, então, ordenadas normalmente pelo protocolo de comunicação em grupo do Seljuk-Amoeba e repassadas para a camada RPC, que se encarrega entregá-las as réplicas. Como todas as mensagens geradas possuem um mesmo número de seqüência e um mesmo endereço de origem (a porta da aplicação), apenas um dos valores calculados será efetivamente entregue e utilizado por todas as réplicas (aquele valor calculado pela réplica que teve seu valor ordenado em primeiro lugar); os outros valores serão descartados como duplicatas de uma mensagem já recebida.

Esta última solução, que pode inclusive ser adotada pelo modelo de réplicas ativas, gera mais tráfego na rede de comunicação e requer mais processamento do que a

primeira, porém é um pouco mais transparente que aquela, uma vez que nenhuma função especial precisa ser chamada. Uma outra vantagem desta solução é que ela pode igualmente ser usada para tratar o não-determinismo gerado pela ocorrência de eventos assíncronos. Neste caso, os eventos são transformados em mensagens especiais, que são enviadas para o grupo. Os *handlers* destes eventos são implementados na forma de linhas de execução que bloqueiam à espera de uma mensagem sinalizando o evento correspondente. Novamente, o mecanismo de ordenação de mensagens garante que os mesmos eventos serão tratados no mesmo ponto de execução por todas as réplicas. Um seqüenciador de eventos é usado para possibilitar que eventos duplicados, já tratados, sejam descartados.

7.6.4. Exemplos de Processamento Replicado no Seljuk-Amoeba

Esta seção mostra um esboço de um programa a ser replicado utilizando as técnicas de replicação passiva e semi-ativa, que utiliza as funções oferecidas pela biblioteca do Seljuk-Amoeba. Para a replicação ativa, como foi dito, nenhuma mudança precisa ser adicionada no código do programa.

A Tabela 7.3 descreve resumidamente a atividade de cada uma das funções oferecidas pela biblioteca para tolerância a faltas do Seljuk-Amoeba. As três primeiras são utilizadas para replicação passiva, enquanto as três últimas são usadas na replicação semi-ativa.

As Figuras 7.5 e 7.6 esboçam, em alto nível, as linhas de execução ativa e passiva, respectivamente, de um servidor replicado passivamente, que fica em um *loop* infinito recebendo pedidos dos clientes, processando-os e enviando as respostas de volta para os clientes.

A Figura 7.5 mostra que a linha de execução ativa começa chamando a função *OnPrimaryUnblock*, que fica bloqueada se a réplica que a chamou não é primária e só retorna se esta réplica se tornar primária em algum instante de sua execução; conseqüentemente o seu chamador também fica bloqueado até que ela retorne. Depois disso, a primeira ação a ser tomada pela réplica primária é atualizar seu estado a partir do último pacote de salvaguarda recebido. Como no início da execução nenhuma salvaguarda foi ainda emitida (e, portanto, checkpack é vazio), a primeira réplica primária entra direto no *loop* sem realizar qualquer atualização de estado. As funções

getrequest e *putreply* são primitivas RPC oferecidas pelo ambiente para receber pedidos e enviar respostas, respectivamente. Após receber um pedido a aplicação realiza algum processamento específico sobre ele e gera uma resposta. A função *Checkpoint* é responsável por estabelecer salvaguardas e ela deve ser chamada de acordo com a estratégia escolhida pelo programador da aplicação. A função *PackFunction* é responsável por montar o pacote de salvaguarda e é também específica da aplicação.

Função	**Descrição**
Checkpoint	Chama a função de empacotamento passada como parâmetro; constrói uma mensagem contendo o pacote por ela gerado e o número de seqüência da última mensagem processada; envia a mensagem para o restante do grupo.
GetCheckpoint	Armazena a mensagem recebida da réplica primária; retira do diário de mensagens de entrada aquelas com número de seqüência menor ou igual ao indicado na mensagem recebida.
OnPrimaryUnblock	Fica bloqueada até que a réplica que a chamou se torne primária.
Notify	Envia uma mensagem para o restante do grupo, comunicando o resultado de um processamento não-determinístico.
GetNotify	Espera o recebimento de uma notificação da réplica líder sobre o resultado de um processamento não-determinístico.
WhatsMyRole	Retorna um código indicando se a réplica que o chamou é líder ou é seguidora.

Tabela 7.3: Funções para Tolerância a Faltas do Seljuk-Amoeba

A Figura 7.6 mostra a linha de execução passiva padrão para um servidor replicado passivamente. Neste caso, as réplicas suplentes ficam em um *loop* recebendo as salvaguardas enviadas pela linha de execução ativa da réplica primária. Nenhum outro processamento é executado. Como o protocolo de comunicação em grupo garante que o transmissor de uma mensagem por difusão restrita não recebe sua própria mensagem, a réplica primária nunca receberá salvaguardas e, portanto, sua linha de execução passiva se bloqueia logo na primeira chamada a *GetCheckpoint*. Nas réplicas suplentes, esta chamada retorna um pacote que contém a salvaguarda emitida pela réplica primária.

```
/* Linha de Execução Ativa */

OnPrimaryUnblock();          /* Retorne apenas se sou a réplica primária */
Se checkpack não está vazio Então  /* Há estado a recuperar? */
        UnpackFunction(checkpack); /* Atualiza o estado */
Fim-Se
Enquanto 1   /* Execute indefinidamente */
        getrequest(header, request, size);     /* Recebe pedido do cliente */
        ...
        Processa o pedido (request) e produz uma resposta (reply)
        ...
        putreply(header, reply, size);          /* Envia a resposta */
        Se é hora de enviar salvaguarda Então
                Checkpoint(PackFunction);   /* Envia salvaguarda */
Fim-Enquanto
```

Figura 7.5: Linha de Execução Ativa de um Servidor Replicado Passivamente

```
/* Linha de Execução Passiva */

Enquanto 1   /* Execute indefinidamente */
        checkpack = GetCheckpoint();        /* Recebe salvaguarda */
Fim-Enquanto
```

Figura 7.6: Linha de Execução Passiva de um Servidor Replicado Passivamente

A Figura 7.7 mostra o esboço de um trecho de aplicação que trata não-determinismo usando as funções da biblioteca do Seljuk-Amoeba para replicação semi-ativa. Como na replicação ativa e ao contrário da passiva, todas as réplicas recebem todas as mensagens de entrada e executam o mesmo código.

Quando algum processamento determinístico deve ser executado, a função *WhatsMyRole* é chamada, retornando qual o papel atual da réplica dentro do grupo. Se a réplica é líder, ela executa o processamento não determinístico e comunica o resultado para as seguidoras, através da função *Notify*. Se *WhatsMyRole* retorna que a réplica é

seguidora, ao invés de realizar o processamento não-determinístico, ela espera o resultado enviado pela líder por meio da chamada à função *GetNotify*. A partir de então, todas as réplicas - líder ou seguidoras - continuam a executar o mesmo código até que um novo processamento não-determinístico tenha que ser executado, quando o processo descrito acima se repete.

```
/* Aplicação Semi-Ativa */

Enquanto 1   /* Execute indefinidamente */

        ...

        getrequest(header, request, size);      /* Recebe pedido do cliente */

        Se há um processamento não determinístico a ser executado Então

                Se WhatsMyRole() = 'líder' Então

                        ...

                        Executa o processamento não determinístico e
                        Coloca o seu resultado em result;
                        ...
                        Notify(result);         /* Envia resultado para a seguidoras */
                Senão
                        result = GetNotify();   /* Recebe o resultado da líder */
        Fim-Se

        ...

        Processa pedido (request) de acordo com result e produz uma resposta

        ...

        putreply(header, reply, size);          /* Envia a resposta */

Fim-Enquanto
```

Figura 7.7: Aplicação Replicada de Forma Semi-Ativa

7.7. Serviço de Diagnóstico de Falta e Detecção de Falhas

Há dois níveis de falhas a se considerar: a falha das réplicas e a falha dos nodos onde elas são executadas. O protocolo de comunicação em grupo nativo do Amoeba [Kaashoek-Tanenbaum 94] provê detecção de falha silenciosa dos nodos onde as réplicas de um processo estão executando de acordo com o grau de resiliência especificado na chamada *CreateGroup()*. A falha de um nodo é detectada pelo núcleo rodando em um

outro nodo por meio do envio de uma mensagem BC_ALIVEREQ para o nodo que está há um certo tempo sem enviar qualquer mensagem. Se após algumas tentativas nenhuma mensagem de resposta BC_ALIVE chega, o nodo solicitante assume que o destino falhou e entra no modo de recuperação, marcando o grupo que possuía réplica executando naquele nodo como não-usável. Todas as chamadas *ReceiveFromGroup()* subseqüentes, feitas pela réplica local daquele grupo, retornam um erro, que leva a réplica local a chamar *ResetGroup()* para reorganizar o grupo. Tal réplica se torna coordenadora do processo de recuperação, ficando responsável, entre outras coisas, por atualizar os diários de mensagens de todas as réplicas do grupo. Neste ponto, todos os membros do grupo revertem para um modo de recuperação e apenas mensagens do protocolo de recuperação são aceitas durante este processo.

Esta mesma idéia poderia ser utilizada em um nível mais alto para detectar falhas das réplicas. Neste caso, sempre que uma réplica fizesse uma chamada *ReceiveFromGroup()*, ela inicializaria um temporizador. Se nenhuma mensagem chegasse dentro do tempo estabelecido, uma falha seria assumida e uma chamada *ResetGroup()* seria feita. Esta funcionalidade, porém, não é transparente para a aplicação, que inevitavelmente toma conhecimento da ocorrência da falha.

Por este motivo, optamos por executar em cada nodo um servidor de detecção - o *Detector* - que funciona como um *daemon* que, de tempos em tempos, inspeciona o funcionamento das réplicas locais, verificando se os processos que as representam ainda estão ativos, e executa um algoritmo para diagnóstico de faltas baseado no DSD Adaptativo (vide Seção 5.4), para descobrir a falha das demais réplicas do grupo ou dos nodos que as hospedam.

Os nodos do sistema são organizados numa lista circular e cada nodo fica responsável por testar o próximo nodo da lista até encontrar um nodo livre de falha. Como a semântica de falha silenciosa é garantida, o teste realizado consiste simplesmente do envio de uma pequena mensagem para o *Detector* do nodo a ser testado e a espera do recebimento de um reconhecimento da mensagem. Se o reconhecimento não chega dentro de um certo intervalo de tempo *t*, o nodo testado é suspeito de ter falhado. Juntamente com as informações relativas aos testes realizados sobre os nodos do sistema, cada nodo participando do algoritmo de diagnóstico envia informações sobre a falha de réplicas locais, caso alguma falha tenha sido detectada.

Determinado o estado de todo o sistema e descoberta a falha de nodos ou das próprias réplicas, as informações sobre os grupos são pesquisadas e aqueles grupos aos quais as réplicas falhas pertenciam ou aqueles que possuíam réplicas executando em um dos nodos falhos devem ser reorganizados. Para tanto, o *Detector* faz uma chamada *ResetRepGroup(group-id, member-list)*, solicitando ao núcleo que ele inicie o processo de recuperação para reorganizar o grupo identificado por *group-id*. Esta chamada retira do grupo os membros falhos, identificados em *member-list*, e incrementa o número de encarnação do grupo, informação esta também armazenada no objeto de configuração do grupo. Esta informação é útil para evitar que vários processos de recuperação sejam realizados para a mesma encarnação do grupo, pois, como todos os servidores de detecção realizam o diagnóstico do sistema, vários deles podem fazer uma chamada *ResetRepGroup*; porém, o processo de recuperação para cada encarnação só deve ser realizado uma única vez. Feita a reconfiguração, todos os nodos sobreviventes guardam informação atualizada sobre os grupos que eles hospedam.

Além disso, caso o tipo de replicação adotado pelo grupo seja passiva ou semi-ativa e a réplica primária/líder tenha sido detectada como falha, *ResetRepGroup* leva o núcleo a executar novamente a função de eleição e a tomar as ações de recuperação pertinentes, conforme discutido na seção anterior. Por exemplo, no caso da replicação passiva, a função *OnPrimaryUnblock* chamada anteriormente pela réplica eleita é desbloqueada para que ela continue o processamento como primária.

7.8. Serviço de Reconfiguração do Sistema

Cada vez que uma réplica (ou o nodo no qual ela está executando) falha e uma chamada *ResetRepGroup()* é feita, o número de membros do grupo diminui e, conseqüentemente, a capacidade de tolerar novas faltas vai sendo paulatinamente reduzida. A fim de manter o grau de resiliência da aplicação durante todo o seu tempo de missão, a reconfiguração do processo replicado se faz necessária. Tal reconfiguração implica que um novo nodo deve ser alocado e que uma nova réplica do processo deve ser iniciada naquele nodo. Fica a cargo do programador decidir se o processo replicado por ele solicitado deve ou não ser automaticamente reconfigurado pelo Seljuk-Amoeba, o que é comunicado ao ambiente através do parâmetro *autoreconfiguration* da primitiva genérica *Replicate*.

A atividade de reconfiguração é realizada pelo próprio *Replicator*. Para tanto, após todos os procedimentos de recuperação terem sido concluídos, o *Detector* do nodo que hospeda a réplica sobrevivente de menor identificador dentro de cada grupo verifica no objeto de configuração do grupo se ele deve ou não ser reconfigurado. Em caso positivo, o *Detector* faz uma chamada a seguinte primitiva do *Replicator*:

RestoreReplicate*(group-id, incarnation)*

Esta chamada é uma rotina *stub* responsável por realizar uma RPC com o *Replicator* solicitando o restabelecimento do nível de resiliência do grupo. Ao receber tal pedido, o *Replicator* deve examinar as informações armazenadas para o grupo identificado por *group-id*. Primeiro, ele verifica se a encarnação atual do grupo combina com o parâmetro *incarnation* e só prossegue se isto for verdade. Depois, ele verifica quantos membros compõem atualmente o grupo. Caso este número seja menor que o grau de resiliência do grupo mais um (o que certamente vai ocorrer após um processo de recuperação), novas réplicas precisam ser criadas. Neste caso, *Replicator* passa a agir da mesma forma descrita na Seção 7.6. Resumidamente, ele requisita os serviços do *FT Run Server* para a criação de novos nodos, passando como parâmetro, entre outras coisas, as versões e as arquiteturas que já estão sendo utilizadas na configuração atual do grupo (para cada grupo, o *Replicator* mantém informações sobre as versões de cada uma das réplicas e as arquiteturas sobre as quais elas executam). Depois disso, o *Replicator* faz uma chamada à nova primitiva de gerência de grupo:

JoinRepGroup*(group-id, node-list)*

Esta primitiva é bastante semelhante a *CreateRepGroup()*, com a diferença que, ao invés de criar um novo grupo, ela acrescenta ao grupo identificado por *group-id* os membros a serem executados nos nodos indicados em *node-list*.

Obtendo sucesso na chamada *JoinRepGroup*(), o serviço de processamento dos nodos criados pelo *FT Run Server* é requisitado para executar as novas réplicas da aplicação. Neste momento, um ponto deve ser considerado: a atualização dos estados das novas réplicas, pois elas devem ter os seus estados corretos e atualizados de modo que suas ações sejam consistentes com aquelas do resto do sistema.

Devido à dificuldade encontrada em realizar a atualização de estado das novas réplicas de uma forma transparente para a aplicação, uma decisão tomada para esta

primeira implementação do Seljuk-Amoeba é oferecer reconfiguração de um servidor replicado que guarda estado apenas se o grupo que o implementa adota o modelo de replicação passiva. Neste caso, as novas réplicas começam executando no modo passivo, de modo que elas recebem salvaguardas da réplica primária e a partir dos quais podem atualizar seus estados, se necessário. É possível, porém, que o próprio programador se encarregue de resolver o problema de recuperação de estado para as outras técnicas, utilizando recursos próprios, como, por exemplo, algum mecanismo de armazenagem estável. Versões futuras do ambiente se propõem a solucionar o problema de atualização de estado para as técnicas de replicação ativa e semi-ativa. Vale lembrar que no caso de servidores que não guardam estado, a reconfiguração sempre é possível, independente da técnica de replicação adotada.

7.9. Sumário

Neste capítulo apresentamos nossa proposta para o provimento de serviços para tolerância a faltas no ambiente operacional Seljuk-Amoeba. Vimos que toda a comunicação em grupo é manipulada pelo núcleo do sistema. As aplicações, replicadas ou não, conversam entre si por meio exclusivamente de primitivas RPC; ao contrário do serviço oferecido pelo Amoeba, a comunicação inter-réplicas é também totalmente transparente para a aplicação. Assim, os protocolos de comunicação em grupo garantem que as mensagens dos clientes sejam atomicamente entregues a todas as réplicas do servidor replicado de forma transparente tanto para os clientes quanto para o servidor.

O serviço de replicação, por sua vez, provê primitivas que trabalham com todas as técnicas de replicação e, mais, ele oferece diferentes nuanças para as técnicas de replicação ativa e semi-ativa, oferecendo flexibilidade para se balancear o grau de continuidade do serviço *versus* o tráfego na rede de comunicação. Como podemos observar nos exemplos da Seção 7.6.4, poucas alterações precisam ser feitas no código da aplicação para que ela execute de forma replicada; de fato, se a técnica de replicação ativa é escolhida, nenhuma alteração precisa ser feita no seu código. O Seljuk-Amoeba se encarrega de implementar todas as funcionalidades necessárias e fornecê-las de forma transparente para a aplicação. Nos casos em que transparência total não é possível, funções de biblioteca são oferecidas para auxiliar o trabalho do programador.

Um serviço para detecção de falha é oferecido transparentemente pelo ambiente. Ao contrário do que ocorre no Amoeba, os membros de um grupo não precisam tomar conhecimento da ocorrência de falhas ou recuperações dos outros membros - tudo isto é feito automaticamente pelo servidor de detecção do Seljuk-Amoeba. Finalmente, um serviço de reconfiguração automática é oferecido para aquelas aplicações que seguem o modelo de replicação passiva, independentes de guardarem ou não estado, e para aquelas que seguem os dois outros modelos de replicação, desde que elas não guardem estado.

A Tabela 7.4 ilustra resumidamente como o suporte oferecido pelo Seljuk-Amoeba para cada uma das técnicas de replicação pode ser explorado para atender a alguns dos principais requisitos das aplicações distribuídas robustas.

Técnica de replicação	**Tempo de Recuperação**	**Não-determinismo da réplica**	**Semântica de falha arbitrária**	**Reconfiguração de grupo com estado**
Ativa	Mais baixo	Permitido (quando resolvido pela aplicação).	Tolerado	Permitido (com suporte da aplicação)
Passiva	Mais alto	Permitido	Tolerado	Resolvido
Semi-ativa	Baixo	Resolvido	Tolerado	Permitido (com suporte da aplicação)

Tabela 7.4: Suporte do Seljuk-Amoeba para Processamento Replicado

Como característico do próprio modelo, o tempo de recuperação mais baixo é oferecido pela replicação ativa, mas o programador tem a opção de requisitar um serviço de replicação semi-ativa com tempo de recuperação tão baixo quanto o da replicação ativa, às custas de geração de tráfego extra na rede de comunicação; do mesmo modo, ele pode, para a replicação ativa, economizar a rede se um pequeno aumento no tempo de recuperação não comprometer outros requisitos da aplicação.

Quanto ao não-determinismo, se a aplicação implementa sua própria solução, conforme discutido na Seção 7.6.3, até mesmo o modelo de replicação ativa pode ser usado. Independente disto, porém, o Seljuk-Amoeba oferece funções que dão suporte à solução deste problema para as outras duas técnicas.

A abstração de nodos com semântica de falha silenciosa fornecida pelo Seljuk-Amoeba permite que todos os modelos de replicação sejam adotados independente da semântica de falha da infra-estrutura de processamento.

A reconfiguração de grupos que implementam servidores sem estado é permitida para qualquer uma das técnicas de replicação. Para servidores que guardam estado, exceto quando replicados de forma passiva, a aplicação deve implementar seus próprios meios de recuperar o estado.

Concluindo, com a integração dos quatro serviços oferecidos pelo Seljuk-Amoeba é possível atender uma vasta gama de requisitos das aplicações, como, por exemplo: baixo tráfego na rede, baixo tempo de recuperação, tratamento de não-determinismo e reconfiguração automática do sistema.

Capítulo 8:

Conclusões

8.1. Discussão

Esta dissertação apresentou uma proposta para a implementação de alguns dos principais mecanismos para tolerância a faltas, impulsionada pela consideração que desenvolver aplicações distribuídas é uma tarefa árdua, principalmente quando os processadores e os canais de comunicação que formam a infra-estrutura de execução destas aplicações podem falhar, o que é bastante comum na grande maioria dos sistemas, que são construídos a partir de componentes de prateleira (*off-the-shelf*).

No Capítulo 3, vimos que a replicação de componentes de *software* individuais em diferentes unidades de processamento de um sistema distribuído provê a redundância que é necessária para o processamento do erro, que possivelmente conduziria à ocorrência de uma falha no sistema. Neste contexto, o processamento do erro envolve o gerenciamento das interações entre as várias réplicas do componente de *software* para detecção, recuperação ou compensação do erro a fim de mascarar o fato de que um ou mais componentes do sistema possam ter falhado. Três modelos básicos de computação replicada foram discutidos:

(i) Modelo de Réplicas Ativas. Neste modelo, todas as réplicas processam concorrentemente e na mesma ordem todas as mensagens de entrada; na ausência de faltas, todas elas devem produzir as mesmas mensagens de saída e na mesma ordem. Isto requer que as réplicas apresentem comportamento determinístico na ausência de faltas. Esta técnica pode ser usada sob suposições tanto de falha silenciosa quanto de falha arbitrária.

(ii) Modelo de Réplicas Passivas. Nesta abordagem, uma das réplicas (a *réplica primária*) processa as mensagens de entrada, provê as mensagens de saída e atualiza, por meio de salvaguardas, os estados internos das demais réplicas (as *réplicas suplentes*). Se a réplica primária falha, uma das suplentes é ativada e

começa a executar a partir do ponto de salvaguarda mais recente. Esta técnica não requer que as réplicas sejam determinísticas e os requisitos de processamento são minimizados. Em contrapartida, ela assume semântica de falha silenciosa para os componentes do sistema.

(iii) Modelo de Réplicas Semi-ativas. Este modelo pode ser visto como um híbrido dos dois anteriores. Apenas uma das réplicas (a *réplica líder*) processa as mensagens de entrada e provê as mensagens de saída. Os estados internos das demais réplicas (as *réplicas seguidoras*) são atualizados por processamento direto das mensagens de entrada. Como na replicação passiva, esta técnica trabalha com possíveis comportamentos não-determinísticos, mas assume semântica de falha silenciosa para os componentes do sistema. Ao contrário daquela e como a replicação ativa, ela provê melhor continuidade de serviço às custas da carga extra incorrida da execução replicada.

Os diferentes modelos de replicação possuem diferentes requisitos quanto à comunicação entre as diversas réplicas do componente de *software*. Tais requisitos são satisfeitos por propriedades específicas do serviço de comunicação, que deve prover uma forma consistente de conversação entre o grupo de réplicas que implementam o componente de *software*.

Uma destas propriedades é a unanimidade, que garante que a mensagem enviada para o grupo de réplicas seja recebida por todos os destinos operacionais. Uma outra característica importante dos serviços de comunicação diz respeito à ordem de entrega das mensagens. Há basicamente quatro formas de ordenação: nenhuma ordenação, ordenação FIFO, ordenação causal e ordenação total. As diversas propriedades dos protocolos de comunicação em grupo foram discutidas no Capítulo 4.

O modelo de réplicas ativas necessita de um serviço de comunicação que ofereça as propriedades de *unanimidade* e *ordenação total*, isto é, todas as réplicas livres de falha recebem as mesmas mensagens e na mesma ordem.

Já no caso das outras duas técnicas de replicação, a passiva e a semi-ativa, há uma réplica privilegiada que pode realizar as operações de ordenação e instruir as demais réplicas. Neste caso, um protocolo de comunicação mais simples pode ser usado, provendo apenas *unanimidade* e *nenhuma ordenação* ou, no máximo, *ordenação FIFO*.

A atividade de diagnóstico da falta, discutida no Capítulo 5, é necessária para: (a) encontrar o componente que está apresentando a falta; e (b) decidir se a falta é permanente ou não. Caso seja detectado que a falta é permanente, os componentes apresentando a falta devem ser removidos do sistema. Porém, a exclusão destes componentes vai paulatinamente degradando o nível de redundância do sistema e, conseqüentemente, a capacidade de se tolerar faltas posteriores. Surge então a necessidade de reconfiguração do sistema, que se dá com a inclusão de novas réplicas no grupo responsável pelo fornecimento do serviço.

A reconfiguração do sistema pode ser considerada apenas se houver recursos redundantes suficientes. Esta tarefa acarreta a realocação e reinicialidação das réplicas que falharam a fim de restaurar o nível de redundância necessário para os protocolos de processamento de erro continuarem funcionando corretamente diante de faltas posteriores. O Capítulo 6 discute diversos aspectos relacionados à atividade reconfiguração do sistema e apresenta estratégias para a realização desta tarefa em sistemas que adotam os diversos modelos de replicação.

Na maioria das vezes, a implementação dos mecanismos para tolerância a falta fica sob responsabilidade do programador da aplicação, havendo, desta forma, um grande aumento na complexidade do seu trabalho, visto que tais mecanismos precisam ser implementados em cada nova aplicação construída.

Com o intuito de reduzir a complexidade embutida na construção de aplicações distribuídas robustas, uma abordagem comumente adotada pelos projetistas de sistemas é considerar que os componentes que formam a infra-estrutura de desenvolvimento e execução da aplicação possuem semântica de falha silenciosa, pois isto simplifica a implementação dos mecanismos para tolerância a faltas. Entretanto, devido às suas restrições de alta confiabilidade e disponibilidade, alguns sistemas de propósito especial não podem assumir que seus componentes de *software*, executando sobre componentes de *hardware* de prateleira, apresentam uma semântica de falha controlada, uma vez que o não cumprimento desta suposição poderá conduzir a uma catástrofe com graves conseqüências.

Uma grande redução na complexidade da construção de aplicações distribuídas robustas pode ser conseguida se os mecanismos para tolerância a faltas forem oferecidos à aplicação sob a forma de serviços, de forma que possam ser compartilhados pelas

várias aplicações que possuam requisitos consideráveis de confiança no funcionamento. Neste caso, há um total reaproveitamento do esforço desprendido na implementação destes serviços.

No Capítulo 1, apresentamos resumidamente o projeto de pesquisa intitulado "*Implementando Aplicações Distribuídas Robustas utilizando os Serviços para Tolerância a Faltas de Micronúcleos Distribuídos*", cujo objetivo principal é construir uma plataforma de desenvolvimento, denominada Seljuk, que facilite a construção e execução de aplicações distribuídas robustas através da fornecimento de duas classes de serviços:

(i) serviços que permitem restringir a semântica de falha dos componentes que formam a infra-estrutura de processamento sobre a qual a aplicação irá executar; e

(ii) serviços que implementam mecanismos para tolerância a faltas.

A primeira classe é responsável pelo oferecimento de serviços de processamento com diferentes semânticas de falha, que possam satisfazer a suposição estabelecida pela aplicação sobre o modo de falha dos componentes do sistema distribuído. O fornecimento e a implementação destes serviços é assunto de discussão de uma outra dissertação, que está sendo desenvolvida como parte do projeto supracitado [Gallindo-Brasileiro 97].

A segunda classe, por sua vez, inclui os serviços de replicação do processamento, comunicação em grupo, diagnóstico de faltas e reconfiguração do sistema. Estes serviços de mais alto nível utilizam os serviços da classe anterior para obter uma semântica de falha mais restritiva, diminuindo assim a complexidade da sua implementação. Tais serviços foram assunto de discussão desta dissertação.

A implementação em curso da plataforma Seljuk está sendo desenvolvida sobre o Amoeba - um sistema operacional distribuído baseado na tecnologia de micronúcleos - e, por este motivo, foi batizada de Seljuk-Amoeba. No Capítulo 7, vimos como os serviços da segunda classe do Seljuk serão oferecidos e implementados no ambiente operacional Seljuk-Amoeba.

Os serviços de comunicação em grupo, diagnóstico de faltas e reconfiguração do sistema são oferecidos de forma totalmente transparente para o nível de aplicação,

independente da técnica de replicação adotada. Já o serviço de replicação propriamente dito, não é totalmente transparente para as técnicas de replicação passiva e semi-ativa, pois alguns aspectos como emissão de salvaguardas e tratamento do não-determinismo são específicos da aplicação. No entanto, o Seljuk-Amoeba oferece uma biblioteca de funções genéricas que podem ser usadas na implementação de aplicações que seguem estes dois modelos de replicação, de forma que o programador não precise codificá-las; é o caso, por exemplo, da função para emissão de salvaguardas na replicação passiva. Obviamente, o programador está livre para usar seu próprio algoritmo caso a função oferecida pela biblioteca não atenda a todos os seus objetivos.

O serviço de replicação é oferecido por um servidor especial de replicação - o *Replicator* - que aceita pedidos para criar um grupo de processos para executar uma aplicação com o nível especificado de resiliência. Este servidor também é invocado para propósitos de reconfiguração, onde ele é responsável por restabelecer o nível de resiliência da aplicação.

O serviço de diagnóstico é oferecido por um servidor de detecção - o *Detector* - implementado no núcleo do sistema e, portanto, executado em cada nodo. Tal servidor é responsável por inspecionar o comportamento da réplica local e executar, juntamente com os demais servidores de detecção, um algoritmo distribuído para diagnóstico de faltas do sistema.

Os três serviços acima são baseados no serviço de comunicação em grupo que, aproveitando o protocolo de difusão irrestrita oferecido pelo Amoeba, garante entrega atômica de mensagens aos membros livres de falha do grupo. Diferentemente do Amoeba, tal serviço é oferecido pelo Seljuk-Amoeba de forma totalmente transparente tanto para a aplicação cliente quanto para a aplicação servidora. Neste aspecto, a codificação de uma aplicação robusta, implementada por um grupo de processos replicados, é realizada da mesma forma que uma aplicação que não possua requisitos consideráveis de confiança no funcionamento.

Como na maioria dos sistemas, a implementação dos serviços para tolerância a faltas no Seljuk-Amoeba assume semântica de falha silenciosa para a infra-estrutura de processamento, porém esta suposição é garantida pelo serviço de processamento confiável que o próprio ambiente operacional oferece (serviços da primeira classe).

8.2. Trabalhos Relacionados

O Delta-4 [Powell 92] é uma arquitetura distribuída tolerante a faltas que usa computadores de prateleira, acomoda heterogeneidade do *hardware* básico e *software* do sistema, e provê portabilidade da aplicação para várias plataformas. Tal arquitetura oferece ao usuário níveis especificáveis de confiança no funcionamento e provê suporte para replicação de componentes de software, comunicação em grupo, diagnóstico de faltas e reconfiguração do sistema.

O Seljuk-Amoeba e o Delta-4 compartilham idéias semelhantes, porém aplicáveis a universos distintos. Enquanto o primeiro aproveita o modelo de *hardware* e *software* e algumas facilidades de um sistema operacional distribuído específico - o Amoeba, o segundo pode ser implementado sob uma variedade de sistemas operacionais. Por outro lado, o Delta-4 requer a utilização de *hardware* especializado para prover a semântica de falha silenciosa exigida para a execução do *software* de comunicação e das entidades que gerenciam a replicação. Além disso, tal arquitetura não oferece meios de garantir esta mesma semântica para os processadores que executam as aplicações, conforme é exigido por algumas técnicas de replicação. Em contraposição, a arquitetura Seljuk, além de oferecer os serviços para tolerância a faltas, provê facilidades que implementam em *software* a semântica de falha exigida por estes serviços bem como pela própria aplicação.

O sistema operacional distribuído ROSE [Ng 90] provê algumas abstrações que facilitam a construção de aplicações confiáveis. Um servidor de detecção de falhas permite que falhas de processadores e canais de comunicação sejam detectadas. Uma abstração de espaço de endereçamento replicado permite a replicação de dados voláteis em processadores diferentes, tornando-os altamente disponíveis apesar da ocorrência de falhas. Estes dois serviços fornecem a base para a implementação de uma abstração de processos resilientes, capazes de sobreviver a faltas de *hardware*. O ROSE, porém, apresenta algumas restrições, dentre as quais: (1) tolera apenas faltas de *hardware*; (2) assume semântica de falha silenciosa para os componentes do sistema; e (3) não permite a restauração do grau de resiliência através da adição de novas réplicas.

[Huang-Kintala 93] discute três tecnologias, representadas por componentes de *software* reusáveis, para aumentar tolerância a faltas a nível da aplicação, as quais podem

ser associadas a técnicas de diversidade de projetos (programação N-versões e blocos de recuperação) para tolerar faltas de concepção do *software*. A primeira delas é representada por um processo *daemon* capaz de detectar falhas de processos e nodos, recuperar processos e reconfigurar o sistema. A segunda tecnologia é uma biblioteca que oferece funções para especificar e realizar salvaguarda de dados críticos da aplicação, recuperar a informação de salvaguardas, atualizar diário de eventos, localizar e reconectar servidores, realizar tratamento de exceções, entre outras. Por fim, um sistema de arquivos multidimensional permite os usuários especificarem e replicarem arquivos críticos. Como no ROSE, estas tecnologias assumem, mas não garantem, semântica de falha silenciosa para os componentes do sistema. Além disso, a dificuldade de programação nestes ambientes continua presente, embora em menor escala. O Seljuk-Amoeba visa superar estas desvantagens e tornar a manipulação de tolerância a faltas o mais transparente possível para o programador.

8.3. Direções para Trabalhos Futuros

No Capítulo 7, foi apresentada nossa proposta para o fornecimento dos serviços para tolerância a faltas no ambiente operacional Seljuk-Amoeba. Como foi possível observar durante a leitura daquele capítulo, tais serviços oferecem muitas possibilidades de implementação. A escolha de um método em particular deverá ser feita na etapa de implementação/experimentação do sistema. Inclusive, a implementação pode ser desenvolvida de forma incremental, onde em cada nova etapa mais funcionalidades vão sendo acrescentadas. Por exemplo, numa primeira implementação, apenas um forma de replicação ativa poderia ser oferecida, digamos aquela em que todas as réplicas enviam suas mensagens de saída e o receptor se encarrega de filtrá-las; numa segunda versão, a outra forma, em que as mensagens são filtradas pelo núcleo do transmissor, poderia ser acrescentada. Esta implementação por etapas pode ser útil para a avaliação da carga extra adicionada ao núcleo em cada versão e, conseqüentemente, da relação custo/benefício de cada versão do sistema. A tecnologia de micronúcleos permite que modificações sejam realizadas no sistema de maneira relativamente fácil.

Versões futuras do sistema poderão oferecer novos serviços para tolerância a faltas, como por exemplo a recuperação do estado para os modelos de réplicas ativas e semi-ativas, permitindo assim a reconfiguração transparente do sistema quando estas

duas técnicas de replicação são usadas. Um outro serviço útil seria um mecanismo para armazenagem estável.

Embora diversos protocolos tenham sido desenvolvidos, possibilitando a construção de uma variedade de mecanismos para tolerância a faltas em sistemas distribuídos, a maneira de integrar estes mecanismos no sentido de garantir o provimento dos requisitos de confiança no funcionamento de aplicações distribuídas ainda não é clara. A flexibilidade oferecida pela plataforma de desenvolvimento Seljuk-Amoeba será útil na avaliação dos custos desta integração.

No momento, a implementação do ambiente Seljuk-Amoeba está sendo iniciada. Após sua conclusão, o próximo passo será desenvolver aplicações distribuídas com variados requisitos de confiança no funcionamento, o que permitirá uma melhor avaliação das potencialidades e das limitações do ambiente.

8.4. Contribuições

Construir aplicações distribuídas capazes de tolerar a ocorrência de falhas é uma tarefa difícil. Acreditamos que a plataforma proposta possibilitará uma sensível redução na complexidade de desenvolvimento destas aplicações. Esse sentimento se baseia nas seguintes considerações: i) as aplicações poderão escolher a semântica de falha dos nodos onde serão executadas sem a preocupação de implementar qualquer mecanismo que garanta essa semântica - isso será feito de forma transparente pela plataforma. Além disso, já que essa escolha é feita em tempo de execução, esse serviço também oferece uma maior flexibilidade para as aplicações, que podem a qualquer momento mudar as considerações feitas sobre a semântica de falha dos nodos, sem que seja necessário qualquer modificação no código executável da aplicação; e ii) a plataforma oferece um número de serviços de tolerância a faltas que poderão ser usados de várias formas no desenvolvimento de novas aplicações, facilitando a implementação destas.

Uma outra vantagem da plataforma é que todos os serviços oferecidos são implementados inteiramente em *software*, sem a necessidade de qualquer componente de *hardware* especial. Essa característica possibilita a utilização de diversidade de projeto, tanto no *hardware* quanto no *software*, para tolerância a faltas de concepção.

Considerando o fato que muitas aplicações não requerem alta confiabilidade e disponibilidade de todos os seus serviços e nem sempre podem admitir o custo envolvido na utilização de mecanismos especiais para tolerância a faltas (por exemplo, circuitos de *hardware* especiais), o Seljuk se apresenta como uma arquitetura flexível que oferece ao usuário a opção de obter níveis especificáveis de confiança no funcionamento na base de "serviço por serviço".

As diferentes abordagens para replicação de processamento podem conviver e cooperar no ambiente Seljuk. A replicação é transparente para o ambiente externo, de modo que uma aplicação replicada se comporta aparentemente da mesma forma que uma não-replicada, com a grande diferença que uma aplicação replicada é resiliente, conseguindo prover o serviço adequado mesmo na presença de um certo número de faltas no sistema.

Diante da importância dos sistemas distribuídos, da necessidade da tolerância à falta e da inadequação dos meios existentes para provê-la, encontramos a relevância deste trabalho de dissertação, que visa exatamente propor uma maneira apropriada de oferecer os serviços básicos de tolerância a faltas em sistemas distribuídos, para que eles possam ser usados de várias formas no desenvolvimento de novas aplicações e que possam ainda ser acrescentados às aplicações existentes exigindo modificações mínimas. Ainda, a plataforma oferece flexibilidade suficiente para que a utilização de tais serviços gere ônus apenas para as aplicações que desejem, ou necessitem, fazer uso deles.

As principais contribuições deste trabalho são sumariadas abaixo:

- a apresentação de um material bem estruturado sobre os principais mecanismos para tolerância a faltas em sistemas distribuídos, bem como um resumo dos conceitos básicos na área;
- a apresentação de uma proposta para terminologia em português na área da computação distribuída, que é uma extensão daquela proposta por Lemos e Veríssimo em [Lemos-Veríssimo 91]; e,
- a proposta para a implementação de mecanismos para tolerância a faltas no ambiente operacional Seljuk-Amoeba, que está sendo desenvolvido sobre um sistema operacional distribuído baseado na tecnologia de micronúcleos que já encontra grande destaque na comunidade acadêmica e científica - o Amoeba.

Apêndice:

Terminologia Estendida da Confiança no Funcionamento

Este apêndice apresenta uma proposta para terminologia em português para a área da confiança no funcionamento de sistemas, que estende aqu ela sugerida por Lemos e Veríssimo em [Lemos-Veríssimo 91]. Aqui, além de adicionar alguns novos termos, inclusive adotados em outras áreas da Ciência da Computação, é dada uma pequena descrição dos termos sugeridos em ambas as propostas, seguindo, na maior parte, a definição apresentada em [Laprie 89]. A Tabela A.1 resume a terminologia proposta por Lemos e Veríssimo, enquanto a Tabela A.2 resume a proposta introduzida neste trabalho.

Termo em Português	Termo em Inglês	Descrição
Confiabilidade	**Reliability**	Medida do contínuo fornecimento do serviço adequado, ou, equivalentemente, do tempo para a falha.
Confiança no Funcionamento	**Dependability**	Propriedade de um sistema de computador tal que confiança possa ser justificavelmente depositada no serviço que ele entrega.
Impedimentos à -	- impairments	Circunstâncias indesejáveis, mas não inesperadas, resultantes da desconfiança no funcionamento.
Meios para -	- means	Métodos e técnicas que possibilitam: (a) obter a capacidade de fornecer um serviço no qual de possa depositar confiança; e (b) garantir que essa confiança seja justificada.
Medidas da -	- measures	Atributos que possibilitam estimar a qualidade do serviço resultante dos impedimentos e dos meios que se opõem a estes.

Obtenção da -	- procurement	Métodos e técnicas que provêem ao sistema a capacidade de fornecer o serviço especificado.
Validação da -	- validation	Métodos e técnicas que possibilitam alcançar confiança na habilidade do sistema entregar o serviço especificado.
Disponibilidade	**Availability**	Medida do fornecimento do serviço adequado, com respeito à alternância entre serviço adequado e inadequado.
Erro	**Error**	A parte do estado do sistema que pode conduzir à ocorrência de uma falha.
Compensação do -	- compensation	Forma de processamento do erro onde o estado errôneo tem informação suficiente para permitir a entrega do serviço adequado.
Detecção do -	- detection	Ação de identificar que o estado do sistema está errôneo.
Processamento do -	- processing	Ações realizadas com o intuito de eliminar um erro do sistema.
Mascaramento do -	- masking	Conseqüência da aplicação sistemática da compensação do erro.
- latente	Latent -	Erro não identificado como tal.
- detectado	Detected -	Erro reconhecido como tal por um algoritmo ou mecanismo de detecção.
Recuperação do -	- recovery	Forma de processamento do erro onde um estado errôneo é substituído por um estado livre de erro.
- para trás	Backward -	Forma de recuperação do erro onde a transformação do estado errôneo consiste em trazer o sistema de volta a um estado anterior livre de erro.
- para a frente	Forward -	Forma de recuperação do erro onde a transformação do estado errôneo consiste em encontrar um novo estado livre de erro para o sistema.
Falha	**Failure**	Situação em que o serviço fornecido pelo sistema se desvia das condições especificadas.
- benigna	Benign -	Falha cujas penalidades são da mesma ordem de magnitude (geralmente em termos de custo) do benefício resultante do fornecimento do serviço adequado.

- catastrófica ou - maligna	Catastrophic - or Malign -	Falhas cujas conseqüências são incomensuráveis com relação ao benefício resultante do fornecimento do serviço adequado.
Falta	**Fault**	A causa de um erro.
Prevenção de -	- avoidance	Métodos e técnicas que ajudam a prevenir a introdução de faltas no sistema.
Tolerância a -	- tolerance	Métodos e técnicas que possibilitam o fornecimento do serviço especificado apesar da ocorrência de faltas.
Previsão de -	- forecasting	Métodos e técnicas que ajudam a estimar a presença, a criação e as conseqüências de faltas.
Supressão de -	- removal	Métodos e técnicas que ajudam a minimizar a presença de faltas no sistema.
Tratamento de -	- treatment	As ações tomadas para prevenir a ativação de uma falta.
Diagnóstico de -	- diagnosis	A ação de determinar a causa de um erro.
- ativa	Active -	Falta que produz erro.
- inativa	Dormant -	Falta interna não ativada pelo processo computacional.
- física	Physical -	Falta resultante de fenômenos físicos adversos.
- humana	Human-made -	Falta resultante de erros humanos.
- interna	Internal -	Falta que tem origem na própria estrutura interna do sistema.
- externa	External -	Falta resultante da interferência do sistema com o meio ambiente.
- de concepção	Design -	Falta interna humana introduzidas durante a concepção ou a manutenção do sistema.
- de operação	Interaction -	Falta externa que tem origem na interação de um usuário com o sistema.
- permanente	Permanent -	Falta que, uma vez presente, permanece no sistema até que ela seja removida.
- temporária	Temporary -	Falta que está presente no sistema por um tempo limitado.

- transitória	Transiente -	Falta física externa temporária.
- intermitente	Intermittent -	Falta interna temporária
- acidental	Acidental -	Falta acidentalmente introduzida no sistema.
- intencional	Intentional -	Falta intencionalmente introduzida no sistema.
----------------------	**Maintainability**	Medida do tempo gasto para reposição do sistema após a ocorrência de uma falha.
Manutenção	**Maintenance**	Preservação da habilidade do sistema para entregar o serviço adequado.
- corretiva	Corrective -	Remoção das faltas que produziram erro e que foram registradas.
- preventiva	Preventive -	Remoção de faltas antes delas serem ativadas.
Segurança (vs. faltas acidentais)	**Safety**	Medida da confiança no funcionamento com respeito a não ocorrência de falhas catastróficas devido a faltas acidentais.
Segurança (vs. faltas intencionais)	**Security**	Medida da confiança no funcionamento com respeito à prevenção de acesso ou manipulação de informações não-autorizados.
Serviço	**Service**	Comportamento do sistema conforme percebido pelo(s) seu(s) usuário(s).
- adequado	Proper -	Aquele que é fornecido de acordo com as condições especificadas.
- inadequado	Improper -	Aquele que é fornecido diferentemente das condições especificadas.
Especificação do -	- specification	Uma descrição do serviço esperado.
Reposição do -	- restoration	Transição do serviço inadequado para o serviço adequado.
Sistema	**System**	Conjunto de componentes que interagem para fornecerem um serviço.
Comportamento do -	- behaviour	O que o sistema faz.
Componente do -	- component	Um outro sistema.
Estado do -	- state	A condição em que o sistema se encontra com respeito a um conjunto de circunstâncias.

Tabela A.1: Terminologia Proposta por Lemos e Veríssimo

Termo em Português	Termo em Inglês	Descrição
Afiliação do grupo	**Group Membership**	Composição atual do grupo, isto é, uma relação dos membros do grupo.
Diário	**Log**	Registro de eventos ocorrendo no sistema ou mensagens transmitidas ou recebidas.
Difusão ou - Irrestrita	**Broadcast**	Mecanismo de comunicação "um-para-todos", onde uma mensagem enviada por um transmissor único é entregue a *todos* os possíveis destinos do sistema.
- Restrita	Multicast	Mecanismo de comunicação "um-para-muitos", onde uma mensagem enviada por um transmissor único é entregue a um *subconjunto* dos possíveis destinos do sistema.
Linha de execução	**Thread**	Uma das diferentes linhas de execução de um processo.
Réplica Suplente	**Backup Replica**	Aquela réplica que, na replicação passiva, não executa qualquer processamento, exceto a atualização do seu estado, e que pode ser usada para substituir a réplica primária.
Retrocesso	**Rollback**	Ação de desfazer um conjunto de operações, voltando a execução do sistema para um ponto de salvaguarda anterior.
Salvaguarda	**Checkpoint**	Conjunto de informações que refletem o estado do sistema.
Ponto de -	-	Ponto da execução de um processo no qual uma operação de salvaguarda é realizada.
Operação de - ou Salvaguardar	To - ou Checkpointing	Ação de guardar o estado de um processo em algum meio de armazenagem estável ou comunicar este estado a outros processos do sistema.
Suplente	**Spare**	Componente que pode ser utilizado para substituir um outro componente que deixou de cumprir sua função.

- Ativa	Hot -	Suplente que realiza concorrentemente as mesmas operações da unidade que ela poderá vir a substituir, não necessitando de inicialização quando ocorre a troca de unidades no sistema.
- Passiva	Cold -	Suplente que ou não está ligada ou está sendo usada para outras tarefas, requerendo inicialização quando ela é ativada para substituir uma unidade que falhou.

Tabela A.2: Terminologia Proposta por Vasconcelos e Brasileiro

Referências Bibliográficas

[Avizienis 76] A. Avizienis, "Fault-Tolerant Systems", *IEEE Transactions on Computers*, C-25(12):1304-1312, dez. 1976.

[Avizienis 85] A. Avizienis, "The N-Version Approach to Fault Tolerant Software", *IEEE Transactions on Software Engineering*, SE-11(12):1491-1501, dez. 1985.

[Avizienis 89] A. Avizienis, "Software Fault Tolerance", *IFIP Computer Congress*, San Francisco, ago. 1989.

[Barrett et al. 90] P. A. Barrett, A. M. Hilborne, P. Veríssimo, L. Rodrigues, P. G. Bond, D. T. Seaton e N. A. Speirs, "The Delta-4 Extra Performance Architecture (XPA)", In *Proc. 20th. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20)*, pp. 481-488, IEEE, Newcastle upon Tyne - UK, jun. 1990.

[Bernstein 88] P.A. Bernstein, "Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing", *IEEE Computer*, 21(2):37-45, fev. 1988.

[Bianchini et al. 90] R. Bianchini Jr., K. Goodwin e D. S. Nydick, "Practical Application and Implementation of Distributed System-Level Diagnosis Theory", In *Proc. 20th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20)*, pp. 332-339, IEEE, Newcastle upon Tyne - UK, jun. 1990.

[Bianchini-Buskens 91] R. Bianchini Jr. e R. Buskens, "An Adaptative Distributed System-Level Diagnosis Algorithm and its Implementation", In *Proc. 21st Int. Symp. on Fault-*

Tolerant Computing Systems (FTCS-21), pp. 222-229, IEEE, Montreal - Canadá, jun. 1991.

[Bianchini-Buskens 92] R. P. Bianchini Jr. e R. W. Buskens, "Implementation of on-line Distributed System-Level Diagnosis Theory", *IEEE Transactions on Computers*, 41(5):616-626, mai. 1992.

[Birman 85] K. P. Birman, "Replication and Fault-Tolerance in the ISIS System", In *Proc. 10th Acm Symposium on Operating Systems Principles, ACM Operating Systems Review*, 19(5):79-86, dez. 1985.

[Birman et al. 91] K. P. Birman, A. Shiper e P. Stepheson, "Lightweitht Causal and Atomic Group Multicast", *ACM Transactions on Computer Systems*, 9(3):272-314, ago. 1991.

[Birman-Joseph 87] K. P. Birman e T. A. Joseph, "Reliable Communication in the Presence of Failures", *ACM Transaction on Computer Systems*, 5(1): 47-76, fev. 1987.

[Borg et al. 83] A. Borg, J. Baumbach. e S. Glazer, "A Message System supporting Fault Tolerance", In *Proc. 9th Symp. on Operating System Principles*, ACM, pp. 90-99, 1983.

[Brasileiro 95] F. V. Brasileiro, "Constructing Fail-Controlled Nodes for Distributed Systems", Tese de Doutorado, University of Newcastle upon Tyne, mai. 1995.

[Brasileiro 96] F. V. Brasileiro, "Implementando Aplicações Distribuídas Robustas Utilizando os Serviços para Tolerância a Faltas de Micronúcleos Distribuídos", Projeto de Pesquisa, Universidade Federal da Paraíba, Campina Grande - PB, 1996.

[Brasileiro 97] F. V. Brasileiro, "Seljuk: Um ambiente para Suporte ao Desenvolvimento e à Execução de Aplicações Distribuídas

Robustas", In *Anais do VII Simpósio de Computadores Tolerantes a Falhas*, pp. 45-59, Campina Grande - PB, jul. de 1997.

[Brasileiro et al. 96] F.V. Brasileiro, P.D. Ezhilchelvan, S.K. Shrivastava, N.A. Speirs, e S. Tao, "Efficient Protocols for Fail-Silent Nodes In Distributed Systems," *IEEE Transactions on Computers*, 45(11):1226-1238, nov. 1996.

[Campbell et al. 79] R. H. Campbell, K. H. Horton e G. G. Belford, "Simulations of a Fault Tolerant Deadline Mechanism", In *Proc. 9th. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-9)*, Madison - Wisconsin, pp. 95-101, jun. 1979.

[Chang-Maxemchuk 84] Jo-Mei Chang e N. F. Maxemchuk, "Reliable Broadcast Protocols", *ACM Transactions on Computer Systems*, 2(3):251-273, ago. 1984.

[Chen-Avizienis 78] L. Chen e A. Avizienis, "N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation", In *Proc. 8th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-8)*, 1978.

[Chérèque et al. 92] M. Chérèque, D. Powell, P. Reynier, J-L Richier e J. Voiron, "Active Replication in Delta-4", In *Proc. 22nd. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-22)*, pp. 28-37, IEEE, Boston - USA, jun. 1992.

[Cheriton-Zwaenepoel 85] D. R. Cheriton e W. Zwaenepoel, "Distributed Process Groups in the V Kernel", *ACM Transactions Computer Systems*, 3(2): 77-107, mai. 1985.

[Cooper 85] E. Cooper, "Replicated Distributed Programs", In *Proc.10th ACM Symposium on Operating Systems Principles, Oper. Syst. Rev.*, 19(5):63-78, dez. 1985.

[Cristian 91] F. Cristian, "Understanding Fault-Tolerant Distributed System", *Communications of the ACM*, 34(2):56-78, fev. 1991.

[Cristian et al. 85] F. Cristian, H. Aghali, R. Strong e D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement", In *Proc. 15th. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-15)*, pp. 200-206, IEEE, Ann Arbor - USA, jun. 1985.

[Cristian et al. 86] F. Cristian, H. Aghali e R. Strong, "Clock Synchronization in the Presence of Omissions e Performance Faults, and Processor Joins", In *Proc. 16th. Int. Symp. on Fault-Tolerant Computing (FTCS-16)*, Vienna - Aústria, jun. 1986.

[Dolev-Strong 83] D. Dolev e H. R. Strong, "Authenticated Algorithms for Byzantine Algorithms", *SIAM Journal of Computing*, 12(4):658-666, nov. 1983.

[Elnozahy-Zwaenepoel 92] E. N. Elnozahy e W. Zwaenepoel, "Replicated Distributed Processes in Manetho", In *Proc. 22nd Int. Symp. on Fault-Tolerant Computing Systems (FTCS-22)*, pp. 18-27, IEEE, Boston - USA, jun. 1992.

[Gallindo-Brasileiro 97] E. L. Gallindo e F. V. Brasileiro, "Processamento Confiável no Ambiente Operacional Seljuk-Amoeba", In *Anais do VII Simpósio de Computadores Tolerantes a Falhas*, pp. 61-74, Campina Grande - PB, jul. de 1997.

[Garcia-Molina 82] H. Garcia-Molina, "Elections in a Distributed Computing System", *IEEE Transactions on Computers*, C-31(1):48-59, jan. 1982.

(FTCS-20), pp. 466-473, IEEE, Newcastle upon Tyne - UK, jun. 1990.

[Kopetz-Merker 85] H. Kopetz, and W. Merker, "The Architecture of MARS", In *Proc.* 15*th. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-15)*, Ann Arbor - USA, pp. 274-279, jun. 1985.

[Kuhl-Reddy 80] J. G. Kuhl e S. M. Reddy, "Distributed Fault-Tolerance for Large Multiporcessor Systems", In *Proc. 7th Annual Symp. Computer Architeture*, pp. 23-30, mai. 1980.

[Kull-Reddy 81] J. G. Kuhl, "Fault-Diagnosis in Fully Distributed Systems", In *Proc. 11th Int. Symp.on Fault-Tolerant Computing Systems (FTCS-11)*, pp. 100-105, jun. 1981

[Lala 86] J.H. Lala, "A Byzantine Resilient Fault Tolerant Computer for Nuclear Power Plant Applications", In *Proc.* 16*th. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-16)*, Vienna - Aústria, pp. 338-343, jul. 1986.

[Lamport 78] L. Lamport, "Time, Clocks and Ordering of Events", *Communications of the ACM*, 21(7):558-565, jul. 1978.

[Lamport el al. 82] L. Lamport, R. Shostak e M. Pease, "The Byzantine Generals Problem", *ACM Trans. Prog. Languages and Systems*, 4(3):382-401, jul. 1982.

[Lampson 81] B. W. Lampson, "Atomic Transactions", In B. W. Lampson, M. Paul e H. J. Siegert, ed. *Distributed Systems - Architeture and Implementation*, pp. 246-265, Springer-Verlag, 1981.

[Laprie 89] J. C. Laprie, "Dependability: a Unifying Concept for Computing and Fault Tolerance", In T. Anderson, ed.

Dependabilty of Resilient Computers, BSP Professional Books, 1989.

[Laprie 92] J. C. Laprie, Dependability: Basic Concepts and Terminology - In English, French, German and Japanese, Vienna, Springer-Verlag, 1992.

[Lee-Anderson 90] P.A. Lee. e D. A. Anderson, Fault Tolerance - Principles and Practice, Spring-Verlag, 1990.

[Lemos-Veríssimo 91] R. de Lemos e P. Veríssimo, "Confiança no Funcionamento - Proposta para uma Terminologia em Português", Comunicação Pessoal, dez. de 1991.

[Liang et al. 90] L. Liang, S. T. Chanson e G. W. Neufeld, "Process Group and Group Communications", *IEEE Computer*, pp. 56-65, fev. 1990.

[Lisbôa-Cavalheiro 95] M. L. B. Lisbôa e G. G. H. Cavalheiro, "Reflexão Computacional sobre Técnicas de Tolerância a Falhas em Software", In Anais do VI Simpósio de Computadores Tolerantes a Falhas, Canela - RS, pp. 405-416, ago. 1995.

[Maes 87] P. Maes, "Concepts and Experiments in Computational Reflexion", In *Proceedings of OOPLAS'87, ACM SIGPLAN Notices*, 22(12):147-155, out. 1987.

[Mishra-Schlicthig 92] S. Mishra e R. D. Schlicthig, Abstraction for Constructing Dependable Distributed Systems, Technical Report TR 92-19, Departament of Computer Science, University of Arizona, Tucson - EUA, 1992.

[Mullender et al. 90] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, e H. van Staveren, "Amoeba: A Distributed Operating System for the 1990's", *IEEE Computer*, 23(5):44-53, mai. 1990.

[Nelson 81] B. J. Nelson, "Remote Procesure Call", Tese de Doutorado, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1981.

[Nelson 90] V. P. Nelson, "Fault-Tolerant Computing: Fundamental Concepts", *IEEE Computer*, pp. 19-25, jul 1990.

[Ng 90] T.P. Ng, "The Design and Implementation of a Reliable Distributed Operating System - ROSE", In *Proceedings of ICDCS*, pp. 2-11, 1990.

[Powell 92] D. Powell (Ed.), Delta-4 - A Generic Architeture for Dependable Distributed Computing, Springer-Verlag, 1992, ISBN 3-540-54985-4.

[Powell et al. 88] D. Powell, P. Veríssimo, G. Bonn, F. Waselynck e D. Seaton, "The Delta-4 Approach to Dependability in Open Distibuted Computing System", In *Proc. 18th. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-18)*, pp. 246-251, IEEE, Tokio - Japan, jun. 1988.

[Randell 75] B. Randell, "System Structure for Software Fault Tolerance", *IEEE Transcations on Software Engineering*, SE-1:220-232, jun. 1975.

[Rivest et al. 78] R. L. Rivest, R. Shamir e L. A. Adleman, "A Method for obtaining Digital Signatures and Public Key Cryptosystems", *Communications of ACM*, 21(2):120-126, fev. 1978.

[Rozier 92] M. Rozier, "Chorus", In *Proc. USENIX Workshop on Microkernels and other Kernel Architectures*, USENIX Association, 1992.

[Schneider 90] F. B. Schneider, "Implementing Fault Tolerant Services using the State Machine Approach: a Tutorial", *ACM Computing Surveys*, 22 (4):299-319, dez. 1990.

[Shin-Ramanathan 87] K. G. Shin e P. Ramanathan, "Diagnosis of Processors with Byzantine Faults in a Distributed Computing System", IEEE, pp. 55-60, 1987

[Shrivastava et al. 91a] S.K. Shrivastava, G.N. Dixon e G.D. Parrington, "An Overview of the Arjuna: A Proggraming System for Reliable Distributed Computing", IEEE *Software*, 8(1):63-73, jan. 1991.

[Shrivastava et al. 91b] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs e D. T. Seaton, "Fail-Controlled Computer Architetures for Distributed Systems", Technical Report No. 333, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne - UK, jul. 1991.

[Shrivastava et al. 92] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs, S. Tao e A. Tully, "Principal Features of the VOLTAN Family Node Architetures for Distributed Systems", *IEEE Transactions on Computers*, 41(5):452-549, mai. 1992.

[Speirs-Barrett 89] N. A. Speirs e P. A. Barrett, "Using Passive Replicates in Delta-4 to Povide Dependable Distributed Computing", In *Proc. 19th. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-19)*, pp. 184-190, IEEE, Chicago - USA, jun. 1989.

[Strong et al. 90] R. Strong, D. Dolev e F. Cristian, "New Latency Bounds for Atomic Broadcast (Extended Abstract)", In *Proc. 11th Real Time System Symposium*, pp. 156-165, Lake Buena Vista - USA, dez. 1990.

[Sullivan-Masson 90] G. F. Sullivan e G. M. Masson, "Using Certification Trails to Achieve Software Fault Tolerance", In *Proc. 20th. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20)*, pp. 423-431, IEEE, Newcastle upon Tyne - UK, jun. 1990.

[Tanenbaum 92] A. S. Tanenbaum, Modern Operating Systems, Prentice-Hall, 1992, ISBN 0-13-595752-4.

[Tully-Shrivastava 90] A. Tully e S. K. Shrivastava, "Preventing State Divergence in Replicated Distributed Programs", in *Proc. 9th. Symp. on Reliable Distributed Systems (SRDS-9)*, pp. 104-113, Huntsville, AL, USA, out. 1990.

[Webber-Beirne 91] S. Webber, and J. Beirne, "The Stratus Architecture", In *Proc. 21st. Int. Symp. on Fault-Tolerant Computing Systems (FTCS-21)*, Montreal - Canadá, pp. 79-85, jun. 1991.

[Wensley et al. 78] J.H. Wensley et al., "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control", *Proceedings of the IEEE*, 66(10):1240-1255, out. 1978.