Serviços de Processamento Tolerantes a Faltas para Sistemas Distribuídos Assíncronos

Lívia Maria Rodrigues Sampaio

Dissertação submetida à Coordenação do Curso de Pós-Gradução em Informática da Universidade Federal da Paraíba - Campus II como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Sistemas Distribuídos

Francisco Vilar Brasileiro

(orientador)

Campina Grande, Paraíba, Brasil

S192S

SAMPAIO, Lívia Maria Rodrigues

Serviços de Processamento Tolerantes a Faltas para Sistemas Distribuídos Assíncronos.

Dissertação de Mestrado, Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande - Pb, Outubro de 2000.

81 p. Il.

Orientador: Francisco Vilar Brasileiro

Palavras-chave: 1. Sistemas Distribuídos Assíncronos 2. Mecanismos para Tolerância a Faltas 3. Confiança no Funcionamento

CDU - 681.3.066D

SERVIÇOS DE PROCESSAMENTO TOLERANTES A FALTAS PARA SISTEMAS DISTRIBUÍDOS ASSÍNCRONOS

LÍVIA MARIA RODRIGUES SAMPAIO

DISSERTAÇÃO APROVADA EM 29.08.2000

PROF. FRANCISCO VILAR BRASILEIRO, Ph.D
Orientador

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

PROF. JONI DA SILVA FRAGA, Dr.
Examinador

CAMPINA GRANDE – PB.

“Nada neste mundo pode substituir a persistência.
Nem o talento - pois nada é mais comum do que talentos fracassados;
Nem a genialidade - pois o gênio incompreendido é quase um pleonasmo;
Nem a educação - pois o mundo está cheio de pessoas cultas marginalizadas.
Só a persistência e a determinação tudo podem.”

(Autor Desconhecido)

Resumo

Disponibilizar mecanismos para tolerância a faltas na forma de serviços, pode diminuir a complexidade no desenvolvimento de aplicações distribuídas com requisitos de confiança no funcionamento. Isto porque, seus projetistas poderão utilizar os serviços sem preocupar-se com qualquer detalhe de implementação referente aos mesmos. Neste trabalho foram propostos serviços de processamento tolerantes a faltas de hardware e software, os quais estão inseridos no contexto de sistemas distribuídos de prateleira. Estes sistemas caracterizam-se por não apresentarem limites conhecidos para os atrasos associados à transmissão de mensagens e escalonamento de tarefas, portanto, são sistemas assíncronos. Complementando a discussão sobre os serviços de processamento, foi feito um estudo acerca do nível de confiança no funcionamento que pode ser obtido através dos mesmos e abordadas algumas estratégias de implementação, além da definição do protocolo de ordenação responsável pela gerência da redundância no grupo de processadores replicados a partir do qual os serviços propostos são providos.

Aplicações com requisitos de confiança no funcionamento são, em maior ou menor grau, críticas, dessa forma, exigem que a implementação dos serviços sobre os quais se apoiam seja devidamente validada. A fim de facilitar esta tarefa, vários modelos de sistema foram propostos na literatura, cada um apresentando vantagens e desvantagens. No caso dos serviços de processamento em questão, utilizou-se uma abordagem híbrida para facilitar o processo de validação. Esta abordagem combina as facilidades dos modelos de sistema assíncrono temporizado e assíncrono com detectores de falhas não confiáveis, já conhecidos, eliminando seus respectivos inconvenientes. A idéia é construir um modelo de sistema assíncrono temporizado estendido com serviços para detecção de falhas e difusão confiável de mensagens, permitindo a especificação de soluções práticas e simples. Os protocolos que implementam tais serviços foram definidos e validados, assegurando as características do modelo estendido requerido.

Abstract

Providing fault tolerance mechanisms through services can decrease the complexity in developing dependable distributed applications. This is because the application programmer will be able to use the services without needing to know how these services were implemented . In this work we propose hardware and software fault-tolerant processing services for off-the-shelf distributed systems. In these systems there no upper bound for the message passing and communication delays, so, they are asynchronous systems. Further, we study the dependability degree that can be achieved using these services and present some implementation strategies. Finally, we defined a protocol for message ordering which is required for managing redundancy into the group of replicated processors over which the processing services are built.

Dependable applications are, in a lesser or greater extent, critical. This fact yields the necessity of validating the implementation of all the services being used by these applications. In order to facilitate this task, a number of system models has been proposed in the literature, each one having its own advantages and disadvantages. In the case of the processing services being proposed in this work, we followed a hybrid approach that gathers the facilities of well know system models, the asynchronous system model with unreliable failure detectors and the timed asynchronous system model, eliminating their respective inconveniences. The objective is to obtain an extended timed asynchronous system model that allows the specification of simple and practical solutions. Such a model incorporates two extra services: an unreliable failure detection service and a reliable broadcast service. The protocols that implement these services are defined and validated, assuring the characteristics of the referred extended model.

Agradecimentos

O aprendizado ao longo deste curso de mestrado foi muito significativo, não somente pelo conhecimento científico adquirido, mas também, pela experiência de vida acumulada. De fato, esta é uma grande conquista para mim, porém, reconheço com muita alegria, que não haveria de consegui-la sem a colaboração de várias pessoas. Então, o meu muito obrigada...

A Deus, que está presente de forma incansável e renovadora em todos os momentos de minha vida, realizando maravilhas e abençoando cada um dos meus passos.

A minha família, por saber que minhas conquistas serão sempre comemoradas como suas próprias conquistas. Especialmente, aos meus pais (Clóvis e Graça) e irmãos (Lincoln, Lígia e Lilia): vocês são o meu eterno porto seguro.

Ao professor Fubica, por ter me orientado na realização deste trabalho de forma eficaz e sempre com muita responsabilidade.

Aos meus queridíssimos amigos, pela torcida e carinho sempre dispensados a minha pessoa. Todas as palavras e gestos foram muito importantes. Sintam-se lembrados nesse pequeno espaço através de Ana Karla, Raissa, Flavinha, Michel, Márcia e Jane.

Aos professores do DSC, por toda competência e dedicação com que desempenham seu ofício. Não posso deixar de registrar a minha admiração por alguns destes: Jorge César, Dalton, Fellipe, Peter, Camilo, Roberto Faria e Jacques Sauvé.

Aos amigos dos cursos de mestrado da COPIN e doutorado da COPELE, pelas experiências compartilhas e todo apoio nos momentos de maior dificuldade. Especialmente, a vocês: Érica, Marcinha, Guga, Góis, Kyller, Marinaldo, Gilene, Tarig e Giggio.

Aos funcionários do DSC, COPIN e Miniblio, por todos os serviços prestados. Como também, ao pessoal da cantina, pela amizade e lanchinhos gostosos.

Conteúdo

1 Introdução **1**

1.1 Motivação 1

1.2 Objetivos 4

1.3 Organização do Trabalho 5

2 Mecanismos para Tolerância a Faltas **7**

2.1 Introdução 7

2.2 Alguns Conceitos sobre Tolerância a Faltas 8

2.3 Tolerância a Faltas de Hardware 10

2.3.1 Replicação Ativa 10

2.3.2 Replicação Passiva 12

2.4 Tolerância a Faltas de Software 13

2.4.1 Blocos de Recuperação 14

2.4.2 Programação em N-versões 17

3 Serviços de Processamento Tolerantes a Faltas para Sistemas Distribuídos **21**

3.1 Introdução 21

3.2 Descrição dos Serviços 22

3.2.1 Tolerando Faltas da Aplicação 24

3.2.2 Tolerando Faltas dos Processadores 25

3.3 Estratégias para Provimento dos Serviços 26

3.3.1 Viabilizando a Implementação do Mecanismo para Tolerância a Faltas dos Processadores 27

3.3.2 Viabilizando a Implementação dos Mecanismos para Tolerância a Faltas da Aplicação 32

3.4 Comparação entre os Serviços quanto à Disponibilidade e à Confiabilidade 40

3.5 Conclusões 44

4 Gerenciando a Redundância dos Processadores 46

4.1 Introdução 46

4.2 Estendendo o Modelo Assíncrono Temporizado 49

4.2.1 O Modelo Assíncrono Temporizado 49

4.2.2 Suposições Extras para Resolver o Problema da Ordenação de Mensagens 50

4.3 Implementando os Serviços Extras do Modelo Assíncrono Temporizado Estendido 53

4.3.1 Um Protocolo para Difusão Confiável de Mensagens 53

4.3.2 Um Protocolo para Detecção de Falhas 56

4.4 Validando os Serviços Extras do Modelo Assíncrono Temporizado Estendido 58

4.4.1 Validando o Protocolo para Difusão Confiável de Mensagens 59

4.4.2 Validando o Protocolo para Detecção de Falhas 62

5 Conclusões 64

A O Problema do Consenso em Sistemas Assíncronos 73

A.1 Introdução 73

A.2 O Modelo de Sistema Assíncrono 74

A.2.1 O Resultado de Impossibilidade FLP 74

A.3 O Modelo de Sistema Assíncrono com Detectores de Falhas Não Confiáveis 76

A.4 O Protocolo de Consenso CT 77

A.4.1 Utilizando o Protocolo de Consenso CT para Resolver o Problema da Difusão Atômica de Mensagens 80

Lista de Figuras

2.1 Hierarquia de Falhas 9
2.2 Abordagem dos Blocos de Recuperação 15
2.3 Blocos de Recuperação - Estrutura 16
2.4 Abordagem da Programação em N-versões 18

3.1 Processamento Tolerante a Faltas 26
3.2 Interface do Objeto Remoto *HelloServer* 29
3.3 Implementação da Interface do Objeto Remoto *HelloServer* 30
3.4 Um Cliente para o Objeto Remoto *HelloServer* 31
3.5 Classes para Manipulação de Exceções em Blocos de Recuperação 33
3.6 Interface para Blocos de Recuperação 34
3.7 Classe para Implementação dos Blocos de Recuperação 35
3.8 Classe para Execução dos Blocos de Recuperação 36
3.9 Classes para Manipulação de Exceções em Conjuntos de N-versões 37
3.10 Interface para Conjuntos de N-versões 37
3.11 Classe para Implementação dos Conjuntos de N-versões 38
3.12 Classe para Execução dos Conjuntos de N-versões 39
3.13 Utilizando Blocos de Recuperação e Conjuntos de N-versões 40

Capítulo 1

Introdução

1.1 Motivação

É crescente a necessidade de se construir sistemas computacionais tolerantes a faltas, haja vista o uso destes sistemas em aplicações que exigem um bom nível de confiança no funcionamento[1]. Nesse caso, o custo de falhas no sistema, seja nos componentes de hardware ou software, pode ser alto, trazendo conseqüências indesejáveis, tais como: perda de dinheiro, perda de produção, perda de clientes, perda de confidencialidade e, até mesmo, perda de vidas humanas.

Sistemas distribuídos, onde uma coleção de computadores são interligados por uma rede de comunicação sem que exista compartilhamento de memória, são freqüentemente utilizados como base para a provimento de tolerância a faltas [MISHRA & SCHLICHTING, 1992]. Isto porque, muitas aplicações com requisitos de confiança no funcionamento possuem características inerentemente distribuídas. Por exemplo, num sistema de automação industrial 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 [VASCONCELOS, 1997].

Uma outra razão favorável ao uso de sistemas distribuídos é que estes oferecem um "esqueleto" (*framework*) natural a partir do qual podem ser implementados mecanismos para

[1]Confiança no funcionamento é o termo adotado por [LEMOS & VERÍSSIMO, 1991] como tradução para o termo em inglês *dependability* cuja definição é apresentada em [LAPRIE, 1989] e corresponde à confiança depositada no serviço a ser fornecido pelo sistema. Confiança no funcionamento é um conceito genérico que possui diferentes atributos, nesse caso, aplicações diferentes exigem dos sistemas computacionais atributos de confiança no funcionamento diferentes. Os atributos mais significativos deste conceito são confiabilidade e disponibilidade, os quais se referem, respectivamente, à probabilidade do sistema fornecer o serviço correto num determinado momento; e à probabilidade do sistema estar pronto para executar num dado instante.

tolerância a faltas. A idéia fundamental dos referidos mecanismos é utilizar alguma forma de redundância e, assim, mascarar ou detectar falhas. Dessa forma, os diversos processadores, memórias, canais de comunicação e unidades de disco rígido encontrados num sistema distribuído provêem redundância que pode ser aproveitada para fins de tolerância a faltas. Note que, a redundância natural dos sistemas distribuídos deve estar associada a mecanismos adequados para tolerância a faltas, a fim de que todo este potencial seja concretizado num sistema capaz de garantir confiança no funcionamento.

Geralmente, os mecanismos para tolerância a faltas são implementados no nível da aplicação, aumentando a complexidade no desenvolvimento da mesma. Seguindo esta filosofia, observa-se também o não reaproveitamento do esforço desprendido na implementação dos mecanismos. De fato, o ideal seria disponibilizar estes mecanismos na forma de serviços que pudessem ser utilizados pelo projetista da aplicação sem que este se preocupasse com questões de implementação. Isto facilitaria a tarefa do projetista e ainda permitiria que os serviços fossem utilizados por diferentes aplicações.

A fim de reduzir a complexidade no desenvolvimento de aplicações distribuídas com requisitos de confiança no funcionamento, os projetistas de tais aplicações têm empregado duas abordagens complementares, quais sejam: 1) utilização de ferramentas de apoio à construção de sistemas tolerantes a faltas, por exemplo, *toolkits* de programação [BIRMAN, 1985] [SHRIVASTAVA ET AL., 1991], bibliotecas de funções [HUANG & KINTALA, 1993] e serviços especializados do sistema operacional [VASCONCELOS, 1997]; 2) restrição da semântica de falha dos componentes que formam a infra-estrutura de processamento e comunicação sobre a qual a aplicação será executada [GALLINDO, 1998]. O ambiente operacional Seljuk [BRASILEIRO ET AL., 1997] contempla as duas abordagens citadas, provendo serviços de alto nível, que implementam ou auxiliam a implementação dos principais mecanismos para tolerância a faltas requeridos por aplicações distribuídas com requisitos de confiança no funcionamento e serviços de baixo nível, que permitem restringir a semântica de falha dos componentes que formam a infra-estrutura sobre a qual a aplicação irá executar.

Aplicações com requisitos de confiança no funcionamento são, em maior ou menor grau, críticas. Desta maneira, a implementação dos serviços sobre os quais tais aplicações se apoiam deve ser validada. No sentido de facilitar esta tarefa, normalmente utiliza-se um modelo de sistema conhecido para descrever a infra-estrutura de execução dos serviços. Na

literatura, podem ser encontrados diferentes modelos de sistema, tais como: modelo síncrono [LAMPORT ET AL., 1982], modelo assíncrono [FISCHER ET AL., 1985], modelo assíncrono temporizado [CRISTIAN & FETZER, 1999] e modelo quasi-síncrono [VERISSIMO & ALMEIDA, 1995]. Estes modelos diferem, essencialmente, em relação a uma característica particular, as garantias de sincronismo providas por cada um deles. Tais garantias são expressas através de suposições feitas acerca dos atrasos na transmissão das mensagens e escalonamento de tarefas. Numa escala de sincronismo, os modelos síncrono e assíncrono estão em extremidades opostas. Enquanto no modelo síncrono os atrasos na transmissão das mensagens e escalonamento de tarefas são limitados e conhecidos, no modelo assíncrono não é imposta nenhuma restrição em relação aos referidos atrasos.

A principal motivação para o uso de modelos conhecidos na validação de serviços que provêem tolerância a faltas nos sistemas, é a possibilidade de aproveitar os resultados já obtidos sobre tais modelos (soluções de consenso, difusão atômica de mensagens, eleição de líder, etc.), facilitando o processo de validação. Vale reforçar que o modelo de sistema escolhido deve ser adequado para descrever os sistemas reais onde os serviços serão implementados.

Teoricamente, um modelo consegue descrever um determinado sistema real quando as garantias de sincronismo especificadas para o mesmo podem ser satisfeitas pelo sistema real durante toda a execução da aplicação, com probabilidade suficientemente alta dos requisitos de confiança no funcionamento desta aplicação serem garantidos. Então, um modelo sem nenhuma restrição quanto às garantias de sincronismo pode ser utilizado para descrever qualquer tipo de sistema. Note que, um modelo livre de qualquer restrição de sincronismo pode apresentar uma semântica bastante simples, entretanto, não permite a resolução de determinados problemas com soluções determinísticas. Por exemplo, o problema do consenso não possui solução determinística para o modelo assíncrono, mesmo quando um único processador pode falhar e esta falha é por parada [FISCHER ET AL., 1985]. Entretanto, a maioria dos sistemas existentes (incluindo os sistemas distribuídos de prateleira) caracterizam-se pela ausência de limites para os atrasos na transmissão de mensagens e escalonamento de tarefas, ou seja, são sistemas assíncronos, porém, requerem alguma noção de tempo. Então, o ideal é que os sistemas reais sejam descritos por modelos assíncronos que incorporem algumas restrições de sincronismo (na escala de sincronismo, um modelo intermediário entre o

síncrono e o assíncrono livre de restrições).

Neste trabalho propõe-se uma solução que objetiva auxiliar o projetista no processo de desenvolvimento de aplicações com requisitos de confiança no funcionamento (requisitos de confiabilidade e disponibilidade), sendo baseada nas duas abordagens citadas anteriormente. Esta solução diferencia-se daquelas que seguem uma mesma linha, tal como o ambiente Seljuk, por considerar um sistema distribuído de prateleira e prover tolerância a faltas tanto do hardware quanto do software. O processo de validação da solução apoia-se numa abordagem híbrida cuja idéia é combinar as facilidades dos modelos assíncrono temporizado [CRISTIAN & FETZER, 1999] e assíncrono com detectores de falhas não confiáveis [CHANDRA & TOUEG, 1996], eliminando seus respectivos inconvenientes.

Seguindo esta abordagem, utiliza-se o modelo assíncrono temporizado para descrever os sistemas distribuídos de prateleira (infra-estrutura de execução), mas, ao invés de projetar e validar soluções complexas considerando tal modelo, identifica-se, inicialmente, as abstrações que podem ser usadas para facilitar o projeto e validação da solução requerida (tais como, detectores de falhas não confiáveis, serviço para difusão confiável de mensagens, etc). Então, projeta-se protocolos simples que implementem tais abstrações, validando os mesmos sobre o modelo assíncrono temporizado. Finalmente, utiliza-se o modelo assíncrono temporizado estendido com os serviços extras para projetar soluções distribuídas de alto nível, as quais serão simples e práticas.

A solução proposta neste trabalho provê tolerância a faltas de hardware e software utilizando mecanismos conhecidos, tais como: replicação ativa [SCHNEIDER, 1990], blocos de recuperação [RANDELL, 1975] e programação em N-versões [AVIZIENIS, 1985]. Portanto, a principal contribuição observada refere-se à estratégia empregada para combinar estes mecanismos a fim de prover confiabilidade e disponibilidade de maneira flexível. Além disso, vale salientar que a abordagem híbrida sugerida para validar a solução é nova e pode ser aplicada a qualquer protocolo distribuído projetado para sistemas assíncronos. Sendo assim, é uma abordagem genérica.

1.2 Objetivos

O objetivo deste trabalho é propor serviços de processamento capazes de tolerar faltas de hardware e software em sistemas distribuídos de prateleira (assíncronos), apresentando a

semântica de falha por parada. Mais especificamente, pode-se apontar os seguintes objetivos:

- propor uma forma de combinar mecanismos para tolerância a faltas de hardware e software no sentido de obter os serviços de processamento desejados;
- discutir uma estratégia para implementação dos serviços propostos;
- analisar (de forma simplificada) os serviços propostos quanto ao nível de confiabilidade e disponibilidade providos, como também, o nível de segurança na semântica de falha escolhida; e
- utilizar uma abordagem híbrida na validação dos serviços de processamento; isto envolve definir e validar um modelo assíncrono temporizado estendido que ofereça as abstrações necessárias para implementar os mecanismos para tolerância a faltas incorporados aos serviços propostos.

1.3 Organização do Trabalho

O restante deste trabalho está organizado da seguinte forma. No capítulo 2 são descritos alguns mecanismos, já desenvolvidos, para tolerância a faltas de hardware e software, quais sejam: as estratégias de replicação ativa e passiva (tolerância a faltas de hardware); as abordagens dos blocos de recuperação e programação em N-versões (tolerância a faltas do software). Na descrição de cada mecanismo são enfocados as características mais importantes (estruturais e funcionais), os pontos fracos e os pontos fortes.

No capítulo 3 tem-se a apresentação dos serviços de processamento propostos, que são dois: serviço de processamento replicado baseado em blocos de recuperação e serviço de processamento replicado baseado em programação em N-versões. A estratégia de replicação do hardware utilizada foi replicação ativa. Ainda neste capítulo, são discutidas algumas alternativas para implementação dos serviços propostos e como estas podem ser empregadas. De fato, as alternativas discutidas descrevem soluções para implementação dos mecanismos que provêem a base de tolerância a faltas dos serviços. Estas soluções concentram-se no nível do sistema operacional ou no nível da linguagem de programação. Por fim, apresenta-se um estudo comparativo sobre os serviços de processamento, a fim de conhecer os fatores que influenciam nos níveis de confiabilidade e disponibilidade que podem ser obtidos através

dos mesmos, como também, no nível de segurança relativo à semântica de falha escolhida; e o custo para prover tolerância a faltas considerando tais fatores. O estudo comparativo é feito a partir das probabilidades acerca do correto funcionamento e operacionalidade dos serviços.

No capítulo 4 é descrito um protocolo de ordenação baseado em consenso que vai garantir a gerência da redundância no grupo de processadores replicados a partir do qual os serviços de processamento são providos. O protocolo de ordenação é projetado sobre um modelo de sistema assíncrono temporizado (abordagem híbrida) cuja definição e validação aparecem neste capítulo. Isto envolve, a definição das principais características do modelo, como também, dos protocolos que implementam os serviços de detecção de falhas e difusão confiável de mensagens, assumidos para o modelo em questão, além da validação de tais protocolos.

As conclusões são apresentadas no capítulo 5. Em seguida, no Apêndice A, são abordadas algumas questões relacionadas ao problema do consenso em sistemas assíncronos, englobando o conceito de detectores de falhas não confiáveis como uma abordagem para permitir a resolução do consenso em sistemas assíncronos, protocolos que se apoiam num serviço para detecção de falhas não confiável e aplicabilidade do consenso na resolução de problemas de acordo sobre o modelo de sistema assíncrono com detectores de falhas não confiáveis.

Capítulo 2

Mecanismos para Tolerância a Faltas

2.1 Introdução

Um sistema pode ser definido como um conjunto de subsistemas que interagem a fim de fornecer um determinado serviço [LEE & ANDERSON, 1990]. Cada um destes subsistemas é, de fato, um sistema. Portanto, o modelo de sistema em questão é recursivo e está organizado a partir da relação sistema/subsistema que desdobra-se até o nível em que não é mais desejável ou possível especificar os detalhes do sistema. Os subsistemas desse nível são denominados componentes do sistema ou componentes atômicos.

Um sistema é dito ser tolerante a faltas se a ocorrência de falhas nos componentes deste sistema não interfere no seu correto funcionamento, nesse caso, o comportamento do sistema será sempre consistente com sua especificação. Para aumentar o nível de tolerância a faltas dos sistemas utilizam-se mecanismos específicos, os quais são baseados em redundância. Estes mecanismos gerenciam e organizam a redundância introduzida no sistema a fim de tolerar faltas dos componentes de hardware e software.

Redundância corresponde à parte do sistema desenvolvida com o único propósito de tolerar faltas. A redundância introduzida no sistema pode ser de hardware, software ou tempo [AVIZIENIS, 1976]. A redundância de hardware compreende os componentes de hardware (processadores, canais de comunicação, disco rígido, etc.) adicionados no sentido de assumir a função daqueles que apresentarem falhas. A redundância de software, por sua vez, inclui todo o software adicional utilizado para gerenciar a redundância do hardware, como também, tolerar as faltas de concepção do software. O tempo extra requerido para executar uma operação repetidas vezes (por exemplo, na retransmissão de mensagens perdidas num sistema de comunicação) constitui a redundância de tempo. Geralmente, utilizam-se os três

tipos de redundância quando se deseja construir sistemas tolerantes a faltas.

Neste capítulo serão discutidos os principais mecanismos utilizados para tolerar faltas de hardware e software.

O restante deste capítulo está organizado da seguinte forma. Na seção 2.2, são abordados alguns conceitos importantes sobre tolerância a faltas. Em seguida, nas seções 2.3 e 2.4, são apresentados, respectivamente, os principais mecanismos para tolerar faltas de hardware e software, a saber: replicação ativa e passiva (hardware); blocos de recuperação e programação em N-versões (software). Sobre os mecanismos, são enfatizados algumas características, limitações e facilidades providas.

2.2 Alguns Conceitos sobre Tolerância a Faltas

Alguns termos utilizados na área de tolerância a faltas serão freqüentemente referenciados ao longo deste capítulo e nos subseqüentes, portanto, faz-se necessário defini-los com clareza.

Uma **falha** do sistema é observada quando o comportamento apresentado pelo mesmo desvia-se daquele descrito na sua especificação. Sendo assim, a ocorrência de uma falha impedirá que o sistema ofereça o serviço desejado. As falhas acontecem devido à presença de erros no estado do sistema, nesse caso, diz-se que o sistema está incorreto. Se existe um **erro** no sistema, então, a execução de uma determinada sequência de ações pode levá-lo a falhar. Um erro é causado por uma **falta**. Por exemplo, a incidência de raios γ sobre a memória do computador pode danificar alguns bits lá armazenados (por exemplo, um bit originalmente com o valor 0 passar a ter o valor 1). O evento "incidência de raios sobre a memória" constitui uma falta, o que torna o sistema incorreto. Nesse caso, o erro corresponde à posição de memória na qual os bits danificados localizam-se. Quando esta posição de memória for referenciada, poderá ocorrer uma falha no sistema. Portanto, uma falha é causada por um erro que, por sua vez, foi gerado a partir de uma falta.

O comportamento do sistema em situações de falha é descrito através da sua **semântica de falha**. É possível que um sistema falhe de diferentes maneiras [JALOTE, 1994] [CRISTIAN, 1991]: interrompendo o fornecimento do serviço, omitindo as saídas para determinados valores de entrada, corrompendo valores de saída, produzindo resultados fora do tempo previsto ou apresentando um comportamento arbitrário. Tais semânticas de falhas são descritas abaixo:

Falhas por Parada. O serviço sendo provido pelo sistema é interrompido, não havendo saída para qualquer valor de entrada, nesse caso, nenhuma requisição de serviço será atendida até que o sistema volte ao seu estado normal (livre de falhas).

Falhas por Omissão. O sistema não produz saída para alguns valores de entrada, isto significa que ocorrem omissões no fornecimento do serviço.

Falhas por Valor. O sistema responde de forma incorreta a algumas requisições de serviço, ou seja, a saída produzida para uma determinada entrada não corresponde ao valor esperado.

Falhas por Desempenho. O sistema responde a uma requisição de serviço de forma correta, mas, fora de hora, isto é, além ou aquém do tempo especificado para tal.

Falhas Bizantinas. O sistema comporta-se de maneira arbitrária durante uma falha. Esta classe de falhas engloba as demais (parada, omissão, valor e desempenho).

As falhas supracitadas formam uma hierarquia ao longo da qual estão dispostas de acordo com o nível de restrição das suas semânticas. Nesse caso, em extremidades opostas da hierarquia encontram-se as falhas mais/menos restritivas (ou bem definidas) quanto aos impactos sobre o funcionamento do sistema, representadas, respectivamente, pelas falhas por parada e Bizantinas. A figura 2.1 ilustra esta hierarquia.

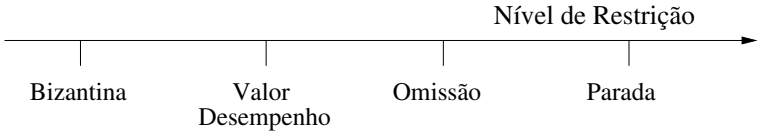

Figura 2.1: Hierarquia de Falhas

As faltas, por sua vez, podem ser classificadas em várias categorias, dentre as quais: quanto às falhas que podem causar [EZHILCHELVAN & SHRIVASTAVA, 1986] e quanto à origem [JALOTE, 1994]. Considerando a classificação de falhas apresentada acima, na primeira categoria tem-se as faltas por parada, omissão, valor, desempenho e arbitrárias. A segunda categoria engloba as faltas físicas e de concepção, descritas a seguir.

Faltas Físicas. São faltas causadas por fenômenos físicos adversos, os quais podem ser externos (radiação, alta temperatura, etc.) ou internos (desgaste natural, curto-cirtuito,

etc.) ao componente de hardware.

Faltas de Concepção. São faltas que ocorrem durante as fases de concepção e implementação do sistema, podendo atingir componentes de hardware e software. São geralmente causadas por erros humanos.

2.3 Tolerância a Faltas de Hardware

As falhas do hardware são causadas por faltas físicas e faltas de concepção, sendo estas menos freqüentes do que aquelas. Isto sugere que o processo de fabricação do hardware vem se tornando cada vez mais confiável. Em se tratando de tolerância a faltas de hardware, o uso de replicação constitui uma estratégia muito comum em sistemas distribuídos.

Considere um sistema distribuído, onde os processadores falham de forma independente e os componentes de software podem receber requisições de serviço provenientes de vários clientes, além disso, tais componentes não são acometidos por faltas de concepção. Sendo assim, os componentes de software falham em decorrência das falhas do processador sobre o qual executam. Nesse contexto, quando apenas faltas físicas devem ser toleradas, utiliza-se replicação dos componentes de software em processadores distintos a fim de mascarar as falhas dos processadores. Por outro lado, se existe a necessidade de considerar as faltas de concepção do hardware, então, os processadores do sistema distribuído devem ser fabricados a partir de projetos distintos.

A seguir serão discutidas duas estratégias de replicação, quais sejam: replicação ativa e replicação passiva[1].

2.3.1 Replicação Ativa

Na replicação ativa, todas as réplicas de um componente de software trabalham em paralelo processando os valores de entrada recebidos e, na ausência de faltas, passam pelas mesmas transições de estado, produzindo saídas idênticas e na mesma ordem. As saídas de todas as réplicas são submetidas a um elemento votador, que realiza votação majoritária para decidir o resultado final do processamento. Nesse caso, é possível mascarar falhas arbitrárias (falhas Bizantinas) dos processadores. Em contrapartida, se a semântica de falha dos processadores

[1]Em [VASCONCELOS, 1997], pode ser encontrada uma explicação mais detalhada sobre as estratégias de replicação ativa e passiva.

é por parada, a replicação ativa pode ser utilizada sem votação, porque qualquer saída produzida será um valor correto. Vale salientar que, para utilizar esta estratégia de replicação faz-se necessário garantir as seguintes condições [SCHNEIDER, 1990]:

Consistência de entrada. O conjunto de valores de entrada deve ser idêntico para todas as réplicas corretas.

Determinismo do grupo de réplicas. Partindo do mesmo estado inicial e processando o mesmo conjunto ordenado de valores de entrada, todas as réplicas corretas produzem o mesmo conjunto ordenado de valores de saída.

A primeira condição é garantida incorporando-se ao sistema de comunicação um protocolo de comunicação que assegure a atomicidade entre os receptores livres de falhas, tal como os protocolos de difusão atômica apresentados em [CHANG & MAXEMCHUK, 1984]. Já o determinismo do grupo de réplicas pode ser conseguido forçando-se o determinismo de cada réplica. Desta maneira, a estratégia de replicação ativa é baseada na abordagem da máquina de estado [SCHNEIDER, 1990].

A abordagem da máquina de estado é um método genérico para suportar tolerância a faltas através de replicação. Uma máquina de estado consiste de variáveis de estado e comandos, os quais, respectivamente, guardam e transformam o estado da máquina. Cada comando é implementado por um programa determinístico, além disso, a execução de um comando é atômica em relação a outros comandos e modifica variáveis de estado e/ou produz alguma saída. Um cliente da máquina de estado requisita a execução de um comando. As respostas para requisições de processamento são direcionadas a um ativador (processo de controle), periférico (terminal, disco rígido) ou ao cliente que enviou a requisição.

Uma versão tolerante a faltas de uma máquina de estado (no caso, um componente de software baseado neste modelo) pode ser implementada replicando-se aquela máquina em processadores distintos de um sistema distribuído. Se as réplicas corretas, isto é, as réplicas executando em processadores livres de falhas, partem do mesmo estado inicial e processam o mesmo conjunto ordenado de requisições, então, serão produzidas saídas idênticas para o processamento realizado por cada réplica. Quando faltas arbitrárias são consideradas, um grupo de réplicas implementando uma máquina de estado tolerante a faltas, deve conter, pelo

menos, $2 * f + 1$ réplicas, onde f é o número de réplicas que podem falhar no sistema[2]. Nesse caso, o resultado do processamento realizado no grupo será a saída produzida pela maioria dos seus membros. Por outro lado, se as réplicas falham, apenas, em decorrência de faltas por parada, então, é suficiente que o grupo seja composto por $f + 1$ réplicas. Desta maneira, o resultado do processamento será a saída produzida por qualquer membro do grupo.

Para implementar uma máquina de estado tolerante a faltas faz-se necessário garantir os seguintes requisitos:

Acordo. Todas as réplicas corretas que implementam uma máquina de estado recebem as requisições de clientes para esta máquina.

Ordem. Todas as réplicas corretas que implementam uma máquina de estado processam as requisições recebidas na mesma ordem.

Tais requisitos determinam a consistência dentro do grupo de réplicas e devem ser atendidos por protocolos específicos, executados por cada réplica.

2.3.2 Replicação Passiva

Na replicação passiva, um componente de software é implementado por um grupo de réplicas em que um único elemento, a réplica primária, recebe, processa e responde a todas as requisições de clientes. Os demais membros do grupo, as réplicas suplentes, só serão ativados no caso de falha da réplica primária, com o intuito de dar continuidade ao serviço interrompido. De acordo com esta estratégia, as réplicas devem apresentar semântica de falha por parada, desse modo, todas as saídas produzidas por uma réplica serão corretas, não havendo necessidade de utilizar nenhum mecanismo para validação de resultados.

As réplicas suplentes de um componente de software guardam uma cópia de alguns estados anteriores deste componente, a partir dos quais a execução pode ser continuada quando a réplica primária falhar. Nesse sentido, a réplica primária emite, periodicamente, uma salvaguarda[3] para todas as réplicas suplentes a fim de atualizar os estados internos dessas réplicas.

[2]O número de componentes do sistema que podem falhar constitui a sua suposição de falha (f). Tal suposição estabelece que o sistema permanecerá funcionando corretamente, enquanto não ocorrerem f falhas de componentes.

[3]De acordo com [POWELL, 1992], uma salvaguarda (*checkpoint*), no contexto de replicação passiva, corresponde a uma "fotografia" do estado interno da réplica primária. Esta "fotografia" pode conter, por exemplo, o valor das variáveis manipuladas pela réplica quando do estabelecimento do ponto de salvaguarda.

Note que, na ausência de faltas, as réplicas suplentes não executam qualquer processamento, exceto a atualização dos seus estados internos.

Quando ocorre uma falha da réplica primária, as suplentes escolhem entre si uma réplica para assumir o papel de primária e esta reinicia a execução interrompida a partir do último ponto de salvaguarda estabelecido, ou seja, a partir do estado interno atual. Isto significa um retrocesso no estado do sistema, consequentemente, as ações realizadas pela réplica primária, antes de sua falha, serão executadas novamente.

Em relação à emissão de salvaguardas, se as réplicas não são determinísticas, então, a réplica primária deve emitir uma salvaguarda sempre que houver mudanças no seu estado interno. Tal procedimento não acarreta uma excessiva sobrecarga de comunicação no sistema, pois, as mudanças ocorridas na réplica primária, entre pontos de salvaguarda consecutivos, são pequenas. Por outro lado, se o determinismo das réplicas pode ser assumido, a freqüência na emissão de salvaguardas diminui. Nesse caso, as réplicas suplentes guardam as mensagens enviadas e recebidas pela réplica primária, desde o último ponto de salvaguarda estabelecido, para que, quando assumirem o papel de primária, processem as mensagens de entrada diretamente, alcançando o mesmo estado da antiga réplica primária, antes da sua falha.

Utilizando replicação passiva não se faz necessário garantir que o grupo de réplicas seja determinístico, além disso, é possível economizar a capacidade de processamento do sistema. Porém, observam-se as seguintes restrições: 1) carga extra de comunicação (permanente), gerada pela necessidade de enviar salvaguardas para as réplicas suplentes e 2) carga extra de processamento (temporária), gerada quando uma réplica suplente assume o papel de réplica primária, tendo que executar o procedimento para recuperação de erros, já discutido. Devido à recuperação de erros, existirá um atraso no fornecimento do serviço e este atraso pode não ser compatível com os requisitos de tempo real de muitas aplicações. Considerando os itens 1 e 2, a replicação passiva oferece menor desempenho em relação à replicação ativa, abordada na seção 2.3.1.

2.4 Tolerância a Faltas de Software

O software não possui propriedades físicas, ao contrário do hardware, dessa forma, pode ser acometido apenas por faltas de concepção. Esse tipo de falta é causada por erros humanos,

introduzidos durante o processo de desenvolvimento do software. As principais abordagens disponíveis para tolerar faltas do software utilizam o conceito de diversidade de projeto.

Diversidade de projeto em software significa desenvolver módulos redundantes utilizando esforços diversos. Nesse caso, os módulos implementam a mesma especificação, mas, são projetados de forma diferente. A idéia é que a diversidade dos módulos redundantes garanta a independência de falhas, isto é, os módulos não irão falhar em decorrência das mesmas faltas de concepção. Na verdade, o objetivo é diminuir a dependência de falhas, impedindo que todos os módulos redundantes falhem da mesma maneira. Para tal, estes módulos devem ser desenvolvidos por várias equipes, utilizando diferentes linguagens, compiladores e algoritmos, além disso, é interessante eliminar o compartilhamento de código entre os módulos e a comunicação excessiva entre os membros das equipes.

A seguir serão discutidas as abordagens dos blocos de recuperação e programação em N-versões, normalmente utilizadas para prover tolerância a faltas de software nos sistemas.

2.4.1 Blocos de Recuperação

A abordagem dos blocos de recuperação foi definida em [RANDELL, 1975] e tem como idéia prover uma forma bem estruturada de introduzir redundância no sistema, a fim de impedir a ocorrência de falhas arbitrárias causadas por faltas de concepção do software. Seguindo esta abordagem, é possível construir componentes de software tolerantes a faltas, os quais podem ser aninhados dentro de um programa seqüencial. Considere que, um componente de software corresponde a um módulo em software, escrito para implementar alguma especificação e, construir um componente de software tolerante a faltas significa estruturá-lo como um bloco de recuperação.

Um bloco de recuperação é constituído de um módulo principal, que desempenha a tarefa especificada para o componente de software; um teste de aceitação, cuja finalidade é validar a saída produzida após a execução da tarefa; e k módulos alternativos ($k > 0$), os quais serão executados seqüencialmente e seguindo uma certa ordem de prioridade, caso a saída produzida por módulos anteriores sejam recusadas pelo teste de aceitação. Os módulos alternativos executam a mesma tarefa especificada para o módulo principal e partindo do mesmo estado inicial, além disso, são desenvolvidos com diversidade de projeto. A figura 2.2 mostra o funcionamento de um bloco de recuperação.

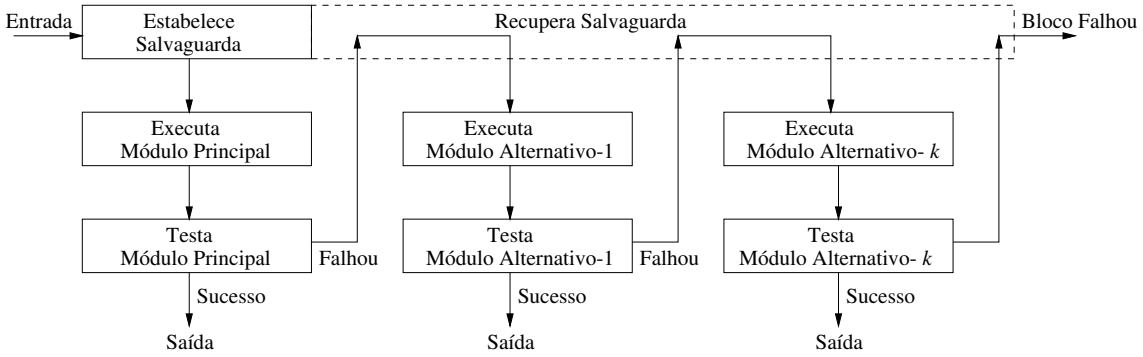

Figura 2.2: Abordagem dos Blocos de Recuperação

Na entrada do bloco é estabelecido um ponto de salvaguarda a fim de guardar o estado atual do sistema, considerado livre de erros. Após a execução do módulo principal, os resultados produzidos por este módulo são submetidos ao teste de aceitação. Se o teste for satisfeito, isto significa que o estado do sistema continua livre de erros (nenhum erro foi detectado) e a computação realizada pelo módulo principal transcorreu de maneira aceitável. Nesse caso, a operação do bloco será finalizada, retornando o controle para o programa no qual está inserido, e a informação guardada sobre o estado do sistema pode ser descartada (remove a salvaguarda).

Por outro lado, se o teste não for satisfeito, assume-se que o módulo principal falhou e o sistema contém erros. Desse modo, utiliza-se a estratégia de recuperação de erros para trás (*backward error recovery*), no sentido de restabelecer o estado do sistema àquele guardado na entrada do bloco. Ao término da recuperação de erros, inicia-se a execução do primeiro módulo alternativo, procedendo da mesma forma como descrito para o módulo principal. Esta seqüência continua até que o teste de aceitação seja satisfeito ou todos os módulos alternativos falhem em tal teste. Nesse último caso, quando o estado do sistema for restabelecido, será sinalizada uma exceção[4] para o programa que invocou o bloco de recuperação, indicando que o bloco falhou. Note que, ao final da operação de um bloco de recuperação, o sistema retorna ao seu estado inicial ou progride para um estado aceitável (livre de erros).

Os diferentes módulos de um bloco de recuperação podem ser organizados de modo que o serviço oferecido por cada um deles apresente depreciação gradativa. Nesse caso, o módulo principal implementa o algoritmo mais eficiente e, provavelmente, mais complexo,

[4]Manipulação de exceções é uma forma de propagar informações sobre falhas através dos vários níveis de abstração do sistema, como também, mascarar falhas de baixo nível [CRISTIAN, 1991].

o que aumenta a possibilidade deste módulo conter faltas de concepção. Por outro lado, se o módulo principal não falhar durante a operação do bloco de recuperação, o serviço é oferecido de forma eficiente. Os módulos alternativos, por sua vez, implementam algoritmos menos eficientes, porém, mais simples e menos susceptíveis a falhas. Esta estratégia aumenta a possibilidade de que existirá, pelo menos, um módulo correto no bloco que oferecerá o serviço desejado e, na maioria das vezes, esse serviço será o mais eficiente.

A abordagem dos blocos de recuperação pode ser suportada através de construções da linguagem de programação. A figura 2.3 mostra um tipo de construção normalmente utilizada para expressar blocos de recuperação. Nesta construção, a cláusula **ensure** especifica o teste de aceitação, como também, indica o início do bloco e dispara a operação de salvaguarda. O módulo principal é especificado pela cláusula **by** e os módulos alternativos, pela cláusula **else by**. Se todos os módulos disponíveis forem testados e nenhum deles passar pelo teste de aceitação, uma exceção será sinalizada, esta ação é representada pela cláusula **else error**.

ensure <teste de aceitação>
by <módulo principal>
else by <módulo alternativo-1>
else by <módulo alternativo-2>
.
.
.
else by <módulo alternativo- k>
else error

Figura 2.3: Blocos de Recuperação - Estrutura

É interessante ressaltar que os blocos de recuperação podem ser aninhados. Isto significa que qualquer módulo de um bloco (principal ou alternativo) pode conter um outro bloco de recuperação. Nesse caso, quando o bloco interno sinalizar uma exceção, considera-se que o módulo no qual este bloco está inserido falhou. Então, inicia-se o procedimento para recuperação de erros no bloco externo, como já foi discutido anteriormente.

O sucesso da abordagem dos blocos de recuperação relaciona-se, principalmente, à confiabilidade do teste de aceitação. De modo geral, o teste de aceitação decide se o bloco de recuperação foi executado de maneira aceitável para o restante do sistema, fornecendo o serviço requerido para a correta operação do programa no qual está inserido. Faz-se

necessário que este teste seja simples, caso contrário, poderá conter faltas de concepção e, consequentemente, falhar. Isto comprometeria a confiabilidade do teste de aceitação, haja vista a possibilidade de avaliações incorretas sobre a computação realizada no bloco, tanto identificando erros falsos quanto ignorando erros verdadeiros no estado do sistema. Além disso, a carga de processamento introduzida pela execução do teste de aceitação pode tornar-se inaceitável com o aumento da complexidade do mesmo. Porém, desenvolver um teste de aceitação simples e eficiente nem sempre é uma tarefa fácil, o que vai depender da sua especificação, ou seja, quão rigoroso o teste deve ser.

Note que, a finalidade do teste de aceitação não é decidir se um determinado resultado é correto, mas, aceitável. Esta característica favorece a especificação de testes menos rigorosos e, portanto, mais simples. O fato do teste de aceitação ser simples não significa que a detecção de erros no bloco de recuperação será menos eficaz. De fato, esse teste não deve ser considerado como único meio para detecção de erros num bloco de recuperação. O ideal é que o mesmo seja reforçado por estratégias de verificação em tempo de execução (*run-time checks*), suportada em hardware, e uso de comandos de teste (*executable assertion statements*), inseridos nos módulos [RANDELL & XU, 1997].

Uma das facilidades oferecidas pela abordagem discutida acima, refere-se à simplicidade de implementação e justifica-se pelo fato de que, tendo um único módulo executando por vez não é necessário utilizar redundância de processadores ou outros recursos do sistema. Outra facilidade, equivale ao ônus da tolerância a faltas quando o módulo principal é correto, nesse caso, o ônus é mínimo e engloba o custo de executar o teste de aceitação e realizar a operação de salvaguarda (estabelecer e recuperar salvaguarda). Além disso, existe o ganho de desempenho, pois o módulo principal é o mais eficiente e, normalmente, o único a ser executado.

2.4.2 Programação em N-versões

A abordagem da programação em N-versões foi definida em [AVIZIENIS, 1985] e tem como proposta utilizar múltiplas computações com diferentes projetos a fim de tolerar as faltas do software.

Seguindo a abordagem da programação em N-versões, um componente de software será implementado por um conjunto de N módulos redundantes (versões), os quais são desen-

volvidos com diversidade de projeto a partir da mesma especificação. Estas N-versões são executadas em paralelo e as saídas produzidas são submetidas a um módulo de validação que decide o resultado final da computação realizada pelo conjunto de N-versões. O módulo de validação, geralmente, implementa um esquema de votação majoritária, sendo denominado de elemento votador. A figura 2.4 ilustra a idéia da programação em N-versões aplicada a um componente de software qualquer.

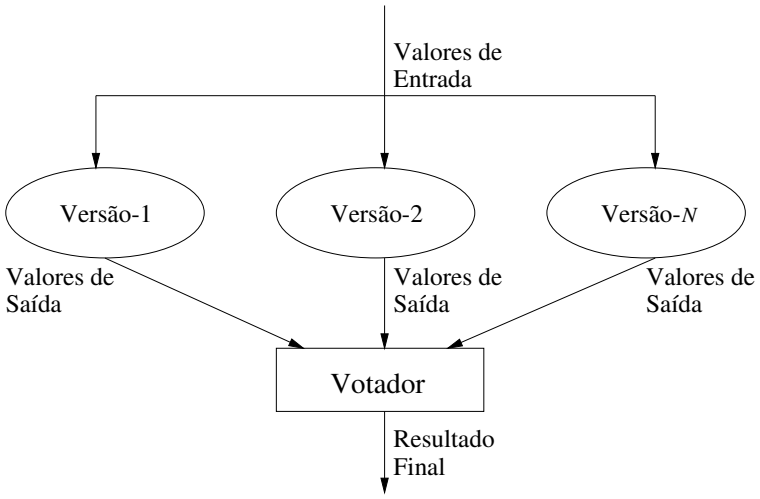

Figura 2.4: Abordagem da Programação em N-versões

A operação do conjunto de N-versões é controlada por um processo de gerência (*driver process*) [JALOTE, 1994]. Este processo tem a função de invocar cada versão, fornecendo os mesmos valores de entrada. Além disso, espera que todas as N-versões terminem suas execuções e compara as saídas produzidas a fim de decidir o resultado final do processamento realizado, nesse caso, desempenhando o papel do módulo de validação.

Ao contrário do que acontece com o teste de aceitação, o resultado final decidido por meio de votação majoritária é considerado correto, ao invés de aceitável. Além disso, a validação através desse esquema parece uma forma mais genérica de detecção de erros, pois, normalmente, o elemento votador pode ser definido independentemente do tipo de aplicação a que está relacionado.

Utilizando o esquema de votação majoritária é possível tolerar faltas de $\lfloor (N-1)/2 \rfloor$ versões, desde que, a maioria remanescente esteja funcionando corretamente e execuções corretas gerem saídas idênticas. Entretanto, nem sempre é possível garantir este requisito, pois algumas computações permitem resultados similares ou múltiplas respostas corretas

[JALOTE, 1994].

Resultados similares ocorrem, principalmente, em operações envolvendo aritmética em ponto flutuante. Por exemplo, as operações $(3.0 * (1.0/3.0))$ e $((3.0 * 1.0)/3.0)$ podem não produzir resultados idênticos devido à precisão das máquinas onde são executadas. Nesse caso, a votação majoritária deve ser substituída por um esquema de votação inexata, no qual o resultado final é decidido a partir de um conjunto de resultados similares. Estes resultados diferem um do outro de um certo valor, portanto, para empregar votação inexata faz-se necessário especificar a discrepância permitida entre resultados corretos. Em contrapartida, quando múltiplas respostas corretas são permitidas para uma determinada computação, o mecanismo de validação utilizado deve ser diferente da votação. Por exemplo, encontrar uma raiz de uma equação de ordem n é um problema que pode apresentar n respostas corretas. Note que, a ausência de um resultado majoritário não significa que a computação falhou.

O sucesso da programação em N-versões está associado à validade do seguinte pressuposto: o uso da diversidade de projeto para desenvolver um conjunto de N-versões produz diversidade de faltas, minimizando a possibilidade das versões falharem de maneira idêntica, ou seja, em decorrência das mesmas faltas de concepção (independência de falhas) [LYU, 1995].

A diversidade de faltas pode ser obtida através da utilização do paradigma da programação diversa [AVIZIENIS ET AL., 1988] [LYU & HE, 1993], cujo objetivo é aumentar a qualidade e independência dos processos de desenvolvimento de software com diversidade de projeto, diminuindo os riscos da ocorrência de faltas comuns[5]. Este paradigma requer a identificação e especificação de pontos de decisão (*cross-check points*) onde será realizada votação entre as versões do software, como também, definição de um protocolo eficiente para a comunicação entre as equipes de desenvolvimento e o coordenador do projeto.

Considerando os estudos de Knight e Leveson descritos em [KNIGHT & LEVESON, 1986], é impossível assumir independência de falhas para faltas de concepção. Além disso, teoricamente, qualquer variação no grau de dificuldade para computar um determinado valor de entrada pode causar dependência de falhas entre as várias versões que realizam tal computação. Isto não significa que a abordagem da programação em N-versões seja ineficaz na tolerância a faltas do software, como será justificado a seguir.

[5]Faltas comuns são causadas, principalmente, por erros de implementação (ou omissões) e ambigüidades na especificação do software.

Através da utilização de testes de comparação (*back-to-back testing*) [KNIGHT & LEVESON, 1989] para validar as N-versões de um componente de software, é possível eliminar faltas de concepção. Testes de comparação constituem um método alternativo, bastante eficaz, para detectar faltas de concepção do software, como demonstra os experimentos apresentados em [LYU, 1995] e [LEVESON & SHIMEALL, 1988]. Outro aspecto que deve ser ressaltado, refere-se ao fato da dependência de falhas só representar um problema quando um número majoritário de versões apresentarem os mesmos erros, ou seja, sofrerem as mesmas faltas de concepção. Dessa forma, combinando boas estratégias de controle de qualidade ao uso de diversidade de projeto é possível amenizar este risco.

Uma das facilidades providas pela abordagem da programação em N-versões refere-se à possibilidade de mascarar faltas dissimilares, dessa forma, não é requerido que as versões sejam livres de erros, mas apenas que apresentem erros diferentes.

Toda a discussão em torno da independência de falhas para faltas de concepção também reflete sobre a abordagem dos blocos de recuperação. Entretanto, a influência deste fator é bem maior sobre a abordagem da programação em N- versões. Note que, utilizando a abordagem dos blocos de recuperação, quando todos os módulos redundantes de um bloco falharem em decorrência da mesma falta de concepção, violando o pressuposto da independência de falhas, será sinalizada uma exceção (o que é uma semântica de falha controlada). Por outro lado, se esta mesma situação ocorrer com um conjunto de N-versões, pode não ser possível garantir a tolerância a faltas provida através da abordagem da programação em N-versões, isto porque, a computação realizada pelas N-versões poderá gerar um resultado inválido e o mesmo ser considerado correto (semântica de falha não controlada).

Capítulo 3

Serviços de Processamento Tolerantes a Faltas para Sistemas Distribuídos

3.1 Introdução

Para tolerar faltas em sistemas sobre os quais executam aplicações com requisitos de confiança no funcionamento, faz-se necessário utilizar mecanismos específicos que possam garantir tais requisitos. Geralmente, estes mecanismos são implementados no nível da aplicação, entretanto, o ideal seria disponibilizá-los na forma de serviços, os quais seriam utilizados pelo projetista da aplicação sem que este precisasse se preocupar com questões relacionadas à implementação de tais mecanismos.

No que se refere à tolerância a faltas em sistemas distribuídos é interessante que as falhas ocorridas em cada nível de abstração do sistema sejam mascaradas ou propagadas para o nível superior como uma falha mais restritiva. Por exemplo, considere uma falha por valor ocorrida na camada física de uma rede de comunicação, causando erro em dois bits de uma mensagem. Se a camada de enlace (acima da camada física) utiliza um código com dois bits para detecção de mensagens corrompidas e descarta as mensagens corrompidas, então uma falha por valor será propagada como uma falha por omissão (que é mais restritiva do que uma falha por valor) [CRISTIAN, 1991]. Dessa forma, o tratamento de falhas no nível da aplicação será mais simples quanto mais restritiva a semântica de falha dos serviços de baixo nível.

Neste capítulo serão apresentados serviços de processamento tolerantes a faltas, cuja finalidade é facilitar a implementação de aplicações distribuídas[1] com requisitos de confiança

[1] Uma aplicação distribuída consiste de um conjunto de processos concorrentes que cooperam entre si a fim de realizarem alguma tarefa [JALOTE, 1994]. Como um processo corresponde à execução de um programa seqüencial, pode-se dizer que uma aplicação distribuída é constituída por um conjunto de programas.

no funcionamento, mais especificamente, requisitos de confiabilidade e disponibilidade. De forma simples, o serviço de processamento será tratado como uma interface entre as aplicações e a infra-estrutura sobre a qual tais aplicações executam (hardware e software do sistema). Em outras palavras, o serviço de processamento é responsável por executar o software da aplicação. É interessante ressaltar que as aplicações distribuídas consideradas são estruturadas em termos de clientes e servidores, então, tais aplicações são compostas por programas-cliente e programas-servidor. Neste trabalho, quando se falar em software da aplicação (ou, simplesmente, aplicação), isto deve ser entendido como servidores que constituem uma aplicação. Sendo assim, a execução dos programas-clientes está fora do escopo dos serviços de processamento em questão.

Os serviços de processamento propostos foram projetados no sentido de utilizar a replicação natural de recursos existente num sistema distribuído, adicionando a estes recursos os mecanismos necessários para tolerar as faltas que possam ocorrer tanto no hardware quanto no software do sistema. A idéia é restringir a semântica de falha dos componentes de hardware (processadores) e software (aplicação), de modo que, a semântica de falha observada pela aplicação seja a mais restritiva (ou simples) possível. No caso, a semântica de falha dos serviços será por parada.

O restante deste capítulo está organizado da seguinte forma. Na seção 3.2, é apresentada a descrição dos serviços de processamento propostos, ressaltando os mecanismos para tolerância a faltas de hardware e software utilizados. Em seguida, na seção 3.3, aborda-se uma estratégia para implementação dos serviços de processamento descritos. Por fim, na seção 3.4, é feito um estudo comparativo sobre os serviços, considerando os níveis de confiabilidade e disponibilidade que os mesmos oferecem, como também, o nível de segurança na semântica de falha escolhida.

3.2 Descrição dos Serviços

Os serviços de processamento descritos nesta seção, estão inseridos no contexto de um sistema distribuído de prateleira, sobre o qual executam aplicações com requisitos de confiança no funcionamento. Estes serviços devem apresentar a semântica de falha por parada, além de oferecer confiabilidade e disponibilidade para as aplicações que os utilizam. De fato, tais atributos são os mais requeridos em aplicações onde é necessário tolerar faltas, ou seja,

aplicações com requisitos de confiança no funcionamento [JALOTE, 1994].

Um sistema distribuído de prateleira caracteriza-se por ser constituído de componentes de prateleira (COTS - *Components Off-The-Shelf*), incluindo o software do sistema operacional e processadores. Um exemplo típico de sistema distribuído de prateleira é um formado por uma rede *Ethernet* de estações de trabalho, onde cada estação executa o sistema operacional UNIX (ou similares) e a comunicação entre os processadores se dá por meio de protocolos Internet (basicamente, TCP/IP). Processadores de prateleira podem falhar de forma arbitrária, muito embora, geralmente, falhem, apenas, por parada (semântica de falha mais restritiva). Observa-se que, a semântica de falha assumida para os processadores depende não somente das características do hardware, mas também, do nível de confiança no funcionamento requerido pela aplicação executada sobre os mesmos. No caso de aplicações críticas (alto grau de criticalidade), não se pode assumir que processadores de prateleira falham, apenas, por parada, faz-se necessário considerar a semântica de falha real dos processadores, utilizando mecanismos que suportem as falhas arbitrárias. Isto porque, para este tipo de aplicação, a ocorrência de falhas pode causar sérios prejuízos (perdas de vidas humanas, por exemplo). O mesmo não acontece em aplicações não críticas (baixo grau de criticalidade), nesse caso, é aceitável assumir que os processadores falham por parada.

Os serviços de processamento definidos neste trabalho destinam-se à execução de aplicações que apresentam requisitos de confiança no funcionamento, mas não são críticas. Além disso, tais aplicações são complexas (programas com muitas linhas de código), sendo assim, é possível que o projetista da aplicação cometa erros durante o processo de codificação, causando as faltas de concepção que podem levar o software da aplicação a falhar de forma arbitrária.

Nesse contexto, os serviços de processamento propostos incorporam mecanismos para tolerar faltas do processador (por parada) e da aplicação (arbitrárias), fortalecendo, respectivamente, os atributos de disponibilidade e confiabilidade requeridos pelas aplicações que os utilizam. Tais mecanismos foram descritos no capítulo 2, quais sejam: blocos de recuperação e programação em N-versões, para tolerar as faltas do software; e replicação ativa, para tolerar as faltas do hardware. Tais serviços serão discutidos a seguir, considerando a introdução, inicialmente, dos mecanismos para tolerar faltas da aplicação e, posteriormente, do mecanismo para tolerar faltas do processador.

3.2.1 Tolerando Faltas da Aplicação

São definidas duas classes de serviços diferentes: uma das classes segue a abordagem dos blocos de recuperação e a outra, da programação em N-versões. Em ambas as classes de serviços, cada servidor de uma aplicação consiste de vários módulos redundantes, os quais obedecem à mesma especificação, mas, são desenvolvidos com diversidade de projeto. Além disso, os serviços de processamento são providos a partir de um único processador.

Seguindo a abordagem dos blocos de recuperação, os módulos redundantes de um servidor (bloco de recuperação) são executados seqüencialmente e as saídas produzidas por cada módulo são avaliadas pelo teste de aceitação. Considera-se que o teste de aceitação é confiável e, qualquer bloco de recuperação é composto por N, $N = f$, módulos redundantes, onde f corresponde ao número de módulos que podem falhar. Dessa forma, é possível que todos os módulos de um bloco falhem para um determinado valor de entrada, nesse caso, será sinalizada uma exceção para o programa onde o bloco está inserido, indicando que o mesmo falhou. Isto significa que a operação do bloco de recuperação não foi finalizada com sucesso, mas o sistema permanece num estado consistente.

Seguindo a abordagem da programação em N-versões, os módulos redundantes de um servidor (conjunto de N-versões) são executados em paralelo no mesmo processador, além disso, versões corretas produzem resultados idênticos para os mesmos valores de entrada, os quais devem ser processados na mesma ordem[2]. A validação dos resultados pode ser feita através de votação majoritária ou comparação. No primeiro caso, as falhas serão mascaradas, sendo assim, para cada servidor tem-se N, $N = 2 * f + 1$, versões, onde f é o número de versões que podem falhar. Por outro lado, utilizando comparação, se algum dos resultados produzidos pelas versões for incompatível com os demais, o serviço de processamento será interrompido (falha por parada), nesse caso, o número de versões para cada servidor deve ser N, $N = f + 1$. Isto significa que todas as N-versões devem funcionar corretamente, caso contrário, o processamento não será concluído com sucesso.

Note que, através dos serviços de processamento discutidos acima, é possível mascarar as falhas da aplicação, silenciar ao detectar uma falha ou, no pior caso, informar sobre a ocorrência de falhas sinalizando uma exceção (o que ainda seria uma semântica de falha controlada). Além disso, como os módulos redundantes de um servidor executam num único

[2] As aplicações consideradas não permitem resultados similares ou múltiplas respostas corretas.

processador, então, se o mesmo falhar, os serviços de processamento também irão falhar, e esta falha é por parada. Em todos os casos, a semântica de falha assumida para os serviços é sempre garantida. Sendo assim, os serviços de processamento funcionam corretamente, isto é, ou estão operacionais ou falham apenas por parada. Tal atributo garante um certo nível de tolerância a faltas para as aplicações que utilizam estes serviços.

Em suma, os serviços de processamento descritos nesta seção, gerenciam a redundância introduzida para tolerar as faltas da aplicação. Em outras palavras, tais serviços têm a função de controlar a execução dos blocos de recuperação e conjunto de N-versões que implementam as aplicações.

3.2.2 Tolerando Faltas dos Processadores

Uma forma de aumentar o nível de tolerância a faltas das aplicações que utilizam os serviços de processamento abordados na seção 3.2.1 é mascarando as falhas dos processadores a partir dos quais estes serviços são providos. Isto pode ser feito através de replicação ativa.

Os serviços de processamento com replicação ativa são descritos da seguinte forma. Cada bloco de recuperação ou conjunto de N-versões é replicado em processadores distintos, onde executam em paralelo os valores de entrada recebidos, produzindo, na ausência de faltas, saídas idênticas e na mesma ordem. Como as falhas da aplicação já são mascaradas ou silenciadas e os processadores apresentam semântica de falha por parada, o resultado final do processamento replicado será a saída produzida por qualquer uma das réplicas.

Desta maneira todos os processadores corretos devem receber o mesmo conjunto de valores de entrada e na mesma ordem. Tais requisitos são atendidos através de um protocolo para ordenação de mensagens, a ser descrito no próximo capítulo. Este protocolo é responsável pela gerência da redundância introduzida para tolerar as faltas dos processadores.

Note que, os serviços de processamento continuam sendo providos enquanto existirem processadores livres de falhas executando réplicas (conjunto de N-versões ou blocos de recuperação) operacionais. Portanto, os serviços em questão serão mais tolerantes a faltas quanto maior o número de processadores sobre os quais é implementado o processamento replicado. Consequentemente, o grau de tolerância a faltas (grau de confiabilidade e disponibilidade) da aplicação que utiliza tais serviços também aumentará.

3.3 Estratégias para Provimento dos Serviços

A figura 3.1 ilustra, de forma genérica, a execução dos serviços de processamento discutidos na seção 3.2. Nesse caso, o processamento está replicado em N processadores ($P_1, P_2, \ldots, P_n$) e o elemento ordenador implementa o protocolo de ordenação. Além disso, cada réplica R de um grupo i, representa a cópia de um servidor da aplicação implementado como bloco de recuperação ou conjunto de N-versões.

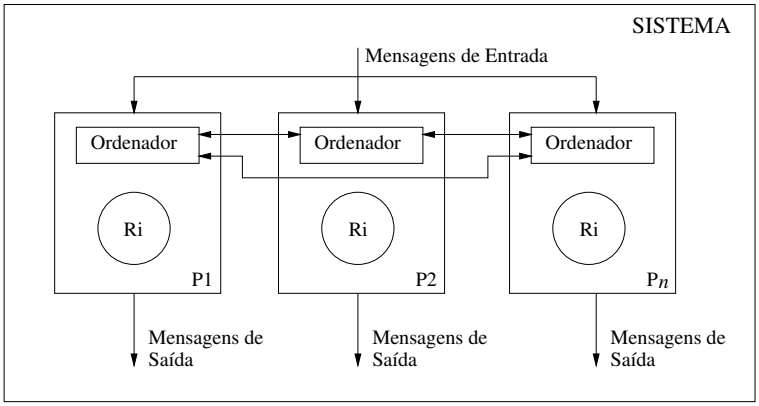

Figura 3.1: Processamento Tolerante a Faltas

A interação cliente-servidor acontece da seguinte forma: o cliente envia mensagens para o servidor e estas são recebidas pelo ordenador executando em cada processador do sistema. Este ordenador garante que todas as réplicas R_i de um servidor recebem as mesmas mensagens e numa mesma ordem[3]. As mensagens de saída geradas por qualquer réplica R são encaminhadas ao cliente que requisitou o referido serviço, o qual filtrará as mensagens duplicadas.

A fim de prover os serviços de processamento em questão, na prática, faz-se necessário viabilizar a implementação dos mecanismos considerados para tolerar as faltas dos processadores (replicação ativa), como também, das aplicações (abordagens dos blocos de recuperação e programação em N-versões). Nas próximas seções serão discutidas algumas estratégias que podem ser empregadas nesse sentido.

[3] A ordenação de mensagens num grupo de processadores replicados envolve a participação dos ordenadores executando em cada um dos processadores do grupo.

3.3.1 Viabilizando a Implementação do Mecanismo para Tolerância a Faltas dos Processadores

As faltas dos processadores são toleradas através de replicação ativa. Nesse caso, para obter um serviço de processamento replicado sugere-se duas estratégias, quais sejam: 1) estender a funcionalidade do serviço de processamento provido por um sistema operacional, como apresentado em [GALLINDO, 1998] ou 2) utilizar as facilidades da orientação a objetos a fim de implementar servidores replicados tolerantes a faltas, como apresentado em [BARATLOO ET AL., 1998]. Note que, a primeira estratégia descreve uma solução no nível do sistema operacional e a segunda, no nível da linguagem de programação.

Considere o sistema operacional Amoeba [MULLENDER ET AL., 1990]. Este sistema é baseado na tecnologia de micronúcleos, nesse caso, todo processador do Amoeba executa uma parte do sistema operacional, denominada micronúcleo, e a funcionalidade que não é provida pelo micronúcleo fica a cargo de processos servidores os quais executam em modo usuário. O serviço de processamento do Amoeba é composto por dois servidores, quais sejam: o servidor de processos (*Process Server*), responsável pela gerência de processos (criação, escalonamento, etc.); e o servidor de execução (*Run Server*), responsável pelo balanceamento da carga nos processadores disponíveis. Estes, respectivamente, executam em modo supervisor (por estar localizado no micronúcleo) e em modo usuário. Vale ressaltar que o *Run Server* faz o balanceamento da carga utilizando os *pools* de processadores[4] disponíveis e levando em consideração fatores tais como memória disponível, velocidade do processador e arquitetura do hardware.

Em linhas gerais a execução de um processo no Amoeba transcorre da seguinte forma. Inicialmente, as tarefas são submetidas ao servidor de execução (*Run Server*) através de um interpretador de comandos (*shell*). Nesse caso, o *shell* identifica na linha de comandos a palavra referente ao programa executável, procura este programa no sistema de arquivos e, se o encontrar, faz com que o mesmo seja executado. Antes de solicitar a execução do programa, o *shell* verifica para quais arquiteturas o programa está disponível, comunica-se com o *Run Server* passando esta informação e solicitando que o mesmo escolha uma arquitetura e uma CPU específica para executar o programa. Ao receber a resposta do *Run*

[4]Um *pool* de processadores é composto por um conjunto de processadores e constitui o poder computacional do sistema operacional Amoeba.

Server, o *shell* comunica-se com o servidor de processos do processador escolhido a fim de que o processo possa ser criado.

O ambiente operacional Seljuk-Amoeba [BRASILEIRO ET AL., 1997] incorpora um serviço de processamento confiável implementado no nível do sistema operacional. Este serviço, descrito em [GALLINDO, 1998], foi obtido estendendo-se o serviço de execução do Amoeba e incorporando-se protocolos para gerência da redundância no serviço de comunicação deste mesmo sistema operacional. No caso, o novo serviço de execução desempenha as mesmas tarefas do serviço de processamento original do Amoeba e provê mecanismos para criação de nodos replicados (conjunto de processadores que executam processamento replicado). O serviço de comunicação proposto faz uso dos protocolos para construção de nodos replicados definidos em [BRASILEIRO, 1995], garantindo assim a consistência no grupo de processadores sobre o qual as aplicações serão executadas.

O serviço de processamento confiável do Seljuk-Amoeba é composto pelas seguintes entidades [GALLINDO, 1998]:

SA-Run Server. Desempenha as mesmas funções do *Run Server* e ainda aquelas referentes à criação de nodos replicados. É de responsabilidade do **SA-Run Server** selecionar os processadores que irão compor o nodo, considerando o nível de confiabilidade requerido pela aplicação que executa sobre o nodo e a semântica de falha assumida para os processadores deste nodo.

SA-Shell. Interface utilizada pela aplicação para acessar o serviço de processamento replicado.

Servidor de Nodos. Gerencia os nodos replicados formados pelo **SA- Run Server**.

Desta maneira, para implementar um serviço de processamento replicado no nível do sistema operacional, faz-se necessário tratar, essencialmente, das questões relacionadas à criação e gerência do grupo de processadores a partir do qual o serviço será provido, como também, do mecanismo de acesso ao referido serviço.

Por outro lado, seguindo uma estratégia orientada a objetos e aproveitando as facilidades da linguagem Java, é possível implementar um esquema de replicação ativa utilizando a tecnologia RMI-Java (*Java Remote Method Invocation*) [MICROSYTEMS, 1997]. Esta

tecnologia é baseada no modelo de objetos distribuídos e vem sendo muito empregada no desenvolvimento de aplicações distribuídas.

Em linhas gerais, uma aplicação RMI é composta por dois programas distintos: um servidor e outro cliente. O servidor cria uma série de objetos remotos[5], registra tais objetos num servidor de nomes RMI (*RMI- Registry*) e espera por invocações de métodos em objetos remotos, provenientes dos clientes. A tecnologia RMI-Java fornece mecanismos a partir dos quais servidores e clientes podem se comunicar, como também, transferir informação de um para o outro e vice-versa, são estes, *stubs* e *skeletons*. Todas estas facilidades são providas na linguagem Java através do pacote **java.rmi**, disponível na API da linguagem Java.

A fim de ilustrar o uso da tecnologia RMI-Java, considere como exemplo a construção de uma aplicação RMI que imprime o texto "Hello World" na tela do computador. No lado servidor, tem-se a interface do objeto remoto (vide figura 3.2) que deve ser uma extensão da classe **java.rmi.Remote**. Após definir a interface do objeto remoto, pode-se escrever uma implementação para o mesmo. A classe que implementa tal interface (vide figura 3.3) é uma extensão da classe **java.rmi.server.UnicastRemoteObject**.

```
public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

Figura 3.2: Interface do Objeto Remoto *HelloServer*

Tendo definido a interface do objeto remoto e uma implementação para o mesmo, é possível criar o *stub* cliente e o *skeleton* servidor. Isto é feito utilizando o compilador **rmi**, como mostrado abaixo:

```
prompt% rmi HelloImpl
```

Um *stub* é uma representação do objeto remoto. Invocações a métodos de objetos remotos, no cliente, são encaminhadas ao servidor, onde os objetos estão localizados, através de um *stub*, cujas funcionalidades são as seguintes: inicia a conexão com o *host* que contém o objeto remoto; grava e transmite os parâmetros para o *host*; espera até que seja recebido o

[5]Um objeto remoto é aquele cujos métodos podem ser invocados a partir de outra máquina virtual Java, normalmente em uma máquina (*host*) diferente. Objetos deste tipo são descritos por interfaces remotas, as quais correspondem a interfaces escritas em Java que contêm a declaração de métodos do objeto remoto.

```
public class HelloImpl extends UnicastRemoteObject implements Hello {
        public HelloImpl() throws RemoteException {
            super();
        }
        public String sayHello() {
            return "Hello World!";
        }
        public static void main(String args[]) {
            try {
                String name = "HelloServer"
                Sytem.out.println("Registering HelloImpl as " + name);
                HelloImpl obj = new HelloImpl();
                Naming.rebind(name, obj);
                System.out.println("HelloServer bound in registry.");
            }
            catch (Exception e) {
                System.out.println("Caught exception while registering: " + e);
            }
        }
}
```

Figura 3.3: Implementação da Interface do Objeto Remoto *HelloServer*

resultado de uma invocação remota; lê o resultado ou exceção retornados; retorna o resultado para o cliente que fez a chamada ao método. No servidor, as invocações de métodos para objetos remotos são recebidas pelo *skeleton* correspondente ao *stub*. Ao receber uma invocação de método, o *skeleton* realiza os seguintes procedimentos: lê os parâmetros para o método remoto; invoca o método no objeto remoto correto; grava e transmite o resultado da execução do método para o cliente.

Os objetos remotos são registrados localmente num servidor de nomes RMI, que deve ser inicializado com o comando,

prompt% rmiregistry <número da porta>

Normalmente, o servidor de nomes escuta na porta de número 1009. Após o servidor de nomes estar rodando, este pode ser utilizado. No caso do exemplo sendo discutido, o cadastro do objeto remoto é dado por,

HelloImpl obj = new HelloImpl();
Naming.rebind(name, obj);

onde, *name* é o nome com o qual o objeto remoto será registrado. O programa cliente que usa o objeto remoto é mostrado na figura 3.4. Para acessar um objeto remoto, o cliente precisa conhecer a referência do objeto, esta é conseguida com o servidor de nomes RMI rodando no *host* onde o objeto está localizado. O pedido do cliente é traduzido pelo trecho,

Hello objStub = (Hello)Naming.lookup(name)

onde, *name* representa o endereço do host onde o objeto remoto está localizado. Através da chamada ao método **lookup**, da interface **Naming**, o *stub* cliente é carregado localmente, possibilitando o acesso ao objeto remoto.

```
public class HelloClient {
        public static void main(String argv[]) {
                // Busca uma referência para a classe Hello
                String name = "rmi://objhost.myorg.com/HelloServer";
                System.out.println("Looking up " + name + "...");
                Hello objstub = null;
                try {
                    objstub = (Hello)Naming.lookup(name);
                }
                catch (Exception e) {
                    System.out.println("Caught an exception looking up Server.");
                    System.exit(1);
                }
                try {
                    System.out.println(objstub.sayHello());
                }
                catch (RemoteException e) {
                    System.out.println("Caught remote exception.");
                    System.exit(1);
                }
        }
}
```

Figura 3.4: Um Cliente para o Objeto Remoto *HelloServer*

Nesse contexto, é possível prover tolerância a faltas através da replicação ativa dos objetos servidores (objetos remotos) em diferentes *hosts*. No caso, faz-se necessário atentar para algumas questões, quais sejam: 1) a invocação de métodos em objetos remotos replicados deve ser recebida pelo skeleton associado a cada réplica do objeto remoto e 2) invocadas na mesma ordem em todas as réplicas do objeto remoto; 3) o stub que fez a invocação do método deve filtrar resultados duplicados, dessa forma, o processamento replicado será transparente

para o cliente; 4) as réplicas de um objeto replicado devem ser registradas com o mesmo nome no servidor de nomes RMI. As questões 1 e 2 visam atender, respectivamente, aos requisitos de acordo e ordem da estratégia de replicação ativa.

Em [BARATLOO ET AL., 1998] é proposto um pacote Java, **Filterfresh**, com classes e interfaces para a construção de servidores replicados tolerantes a faltas. A gerência do grupo de servidores replicados é suportada por uma classe, **Group Manager**, instanciada junto a cada réplica a fim de formar um grupo lógico de gerenciadores. Esta classe implementa mecanismos de comunicação (*multicast* confiável) e gerenciamento de grupo (criação, inserção e remoção de membros, etc.). É interessante ressaltar que algumas das classes e interfaces do pacote **Filterfresh** são extensões de classes e interfaces do pacote **java.rmi**.

O pacote **Filterfresh** foi utilizado na definição de um serviço de nomes RMI tolerante a faltas (*FTRegistry*) que, dentre outras facilidades, permite o registro de objetos remotos replicados com o mesmo nome. Além disso, foi mostrado como empregar as classes do pacote em questão para implementar, no lado cliente, um mecanismo que permite mascarar falhas dos servidores de maneira transparente para o cliente. Tal mecanismo, denominado *FTUnicast*, é baseado na estratégia de replicação passiva, entretanto, este pode ser facilmente estendido no sentido de possibilitar a implementação de servidores replicados (objetos remotos replicados) por meio de replicação ativa, como sugere os autores do artigo referenciado.

3.3.2 Viabilizando a Implementação dos Mecanismos para Tolerância a Faltas da Aplicação

Em se tratando dos mecanismos para tolerar as faltas da aplicação, cada servidor pode ser implementado seguindo uma estratégia orientada a objetos. A idéia é disponibilizar as facilidades para uso das abordagens dos blocos de recuperação e programação em N-versões no nível da linguagem de programação, através de classes/interfaces que seriam estendidas/implementadas pelo projetista da aplicação. A seguir, apresenta-se uma maneira de empregar a estratégia sugerida, tomando como base a resolução de um problema específico.

Considere o problema de implementar um servidor para multiplicação de dois valores inteiros[6], seguindo as abordagens já conhecidas. No caso dos blocos de recuperação, a resolução deste problema consiste em definir os módulos redundantes que implementam

[6]Esta aplicação é bastante simples. Isto permite maior concentração na explicação dos mecanismos para tolerância a faltas de software e não na solução particular de um problema.

o servidor da aplicação; definir o teste de aceitação; definir os procedimentos responsáveis pela operação de salvaguarda (estabelece e recupera salvaguarda); definir o procedimento para executar os módulos redundantes de acordo com o esquema de funcionamento de um bloco de recuperação; e definir os procedimentos para indicação de exceções no caso de falha do bloco. Note que, tais definições, com exceção das duas últimas, dependem do serviço sendo implementado, porém, são requeridas para todo servidor estruturado como bloco de recuperação.

O esquema de funcionamento de um bloco de recuperação engloba os seguintes passos:

1. Estabelece salvaguarda
2. Executa módulo i
3. Submete saída ao teste de aceitação
   a) se teste for satisfeito, então, vá para o item 4.
   b) se teste não for satisfeito, então, vá para o item 5.
4. Finaliza operação do bloco de recuperação retornando o resultado aceito pelo teste
5. Recupera salvaguarda
6. Tem outros módulos para executar? Sim: vá para 2; Não: vá para 7
7. Sinaliza exceção

Utilizando as facilidades da linguagem Java, é possível implementar uma solução baseada na abordagem dos blocos de recuperação para o problema discutido acima. Isto é feito através de um conjunto de classes e interfaces, cuja descrição é apresentada logo abaixo.

A classe **RecoveryBlockFailureException** (vide figura 3.5) identifica o objeto que trata das falhas do bloco de recuperação e de seus módulos redundantes, isoladamente. Note que esta classe não tem corpo, então, os métodos acessíveis neste objeto são aqueles herdados da classe **RuntimeException**.

A interface **RecoverableBlock** (vide figura 3.6) contém a definição dos métodos *getNbAlternates()*, fornece o número de módulos redundantes de um bloco de recuperação; *invokeAlternate(-,-)*, invoca cada módulo redundante passando os parâmetros de entrada específicos da aplicação sendo implementada; *isAcceptable(-,-)*, realiza o teste de

```
public class RecoveryBlockFailureException extends RuntimeException {
}
public class AlternateExecutionFailureException extends RuntimeException {
}
```

Figura 3.5: Classes para Manipulação de Exceções em Blocos de Recuperação

aceitação sobre os resultados produzidos pela execução de cada módulo redundante i; além de *saveCheckpoint()* e *restoreCheckpoint()*, responsáveis, respectivamente, pelo estabelecimento e recuperação de salvaguardas.

```
public interface RecoverableBlock {
    public int getNbAlternates();
    public Object invokeAlternate(int i, Object[] params) throws AlternateExecutionFailureException;
    public boolean isAcceptable(Object[] params, Object result);
    public void saveCheckpoint();
    public void restoreCheckpoint();
}
```

Figura 3.6: Interface para Blocos de Recuperação

A classe **RecoverableBlockExample** (vide figura 3.7) é um exemplo de implementação da interface **RecoverableBlock**. Além disso, a referida classe contém métodos que implementam os módulos redundantes dos servidores da aplicação, os quais compõem um bloco de recuperação.

A classe **RecoveryBlock** (vide figura 3.8) contém o método *execute(-)* cuja finalidade é executar um bloco de recuperação através dos passos enumerados anteriormente.

Por outro lado, no caso da programação em N-versões, a solução para o problema da multiplicação de dois números inteiros, enunciado anteriormente, consiste em definir os módulos redundantes (versões) que implementam o servidor da aplicação; definir o mecanismo de validação (votação majoritária, comparação ou votação inexata); definir o procedimento para executar o conjunto de N-versões e definir os procedimentos para indicação de falhas na operação do conjunto de N-versões. As observações sobre tais definições, colocadas para o caso dos blocos de recuperação, também são válidas neste caso. Em relação à execução do conjunto de N-versões, isto acontece da seguinte forma (assuma o mecanismo de validação como sendo comparação):

```
public class RecoverableBlockExample implements RecoverableBlock {
    public RecoverableBlockExample() { \\construtor
    }

    public int getNbAlternates() {
        return 2;
    }

    public Object invokeAlternate(int i, Object[] params) throws AlternateExecutionFailureException {
        if (params.length != 3) throw new AlternateExecutionFailureException();
        switch(i) {
            case 0:
              return new Integer(mul1(((Integer) params[0]).intValue(), ((Integer)params[1]).intValue(), ((Integer) params[2]).intValue()));
            case 1:
              return new Integer(mul2(((Integer) params[0]).intValue(), ((Integer)params[1]).intValue(), ((Integer) params[2]).intValue()));
            default:
              throw new AlternateExecutionFailureException();
        }
    }
    private int mul1(int delay, int x, int y) { \\módulo principal
        try {
            System.out.println("RecoverableBlockExample: mul1 sleeping for " + delay);
            Thread.sleep(delay);
        }
        catch (InterruptedException e) {
        }
        System.out.println("RecoverableBlockExample: mul1 returning for " + x*y);
        return x*y;
    }

    private int mul2(int delay, int x, int y) { \\módulo alternativo
        try {
            System.out.println("RecoverableBlockExample: mul2 sleeping for " + delay);
            Thread.sleep(delay);
        }
        catch (InterruptedException e) {
        }
        int result = 0;
        for (int i = 0; i < x; i++) {
            result += y;
        }
        System.out.println("RecoverableBlockExample: mul2 returning for " + result);
        return result;
    }
    public boolean isAcceptable(Object[] params, Object result) {
        return (((Integer) result).intValue()/((Integer) params[2]).intValue() == ((Integer) params[1]).intValue());
    }
    public void saveCheckpoint() {
    }

    public void restoreCheckpoint() {
    }
}
```

Figura 3.7: Classe para Implementação dos Blocos de Recuperação

```
public class RecoveryBlock {
        private RecoverableBlock recoverableBlock;

        public RecoveryBlock(RecoverableBlock block) { //construtor

              this.recoverableBlock = block;
        }

        public Object execute(Object[] params) throws RecoveryBlockFailureException {

              Object result;
              int i = 0;
              this.recoverableBlock.saveCheckpoint();
              while (i < this.recoverableBlock.getNbAlternates()) {
                    try {
                          result = this.recoverableBlock.invokeAlternate(i, params);
                          if (this.recoverableBlock.isAcceptable(params, result))
                              return result;
                     }
                     catch (AlternateExecutionFailureException e) {
                           System.out.println("RecoveryBlock: " + e);
                           this.recoverableBlock.restoreCheckpoint();
                     }
                     finally {
                           i++;
                     }
              }
              throw new RecoveryBlockFailureException();
        }
}
```

Figura 3.8: Classe para Execução dos Blocos de Recuperação

1. Invoca todas as versões do servidor, as quais são executadas em paralelo

2. Espera até que a execução de todas as versões seja finalizada

3. Submete saídas produzidas ao mecanismo de validação

   a) se o resultado da validação for positivo, então, vá para o item 5.

   b) se o resultado da validação for negativo, então, vá para o item 4.

4. Interrompe a operação do conjunto de N-versões (falha por parada)

5. Finaliza a operação do conjunto de N-versões retornando a saída produzida por qualquer uma das versões

A implementação de uma solução baseada na abordagem da programação em N-versões e considerando o problema enunciado, segue a mesma estratégia discutida para os blocos de recuperação. Nesse caso, utiliza-se outras classes e interfaces, as quais são descritas a seguir.

As classes **NVersionModuleFailureException** e **ModuleExceptionFailureException** (vide figura 3.9) implementam os procedimentos para indicação de exceções que refletem a ocorrência de falhas na operação do conjunto de N-versões e de cada uma das versões, separadamente.

```
public class NVersionModuleFailureException extends RuntimeException {
}

public class ModuleExecutionFailureException extends RuntimeException {
}
```

Figura 3.9: Classes para Manipulação de Exceções em Conjuntos de N-versões

Já a interface **NVersionedModule** (vide figura 3.10), contém a definição dos métodos *getNbModules()*, fornece o número de versões de um conjunto de N-versões; *invokeModule(-,-)*, invoca cada versão de um conjunto passando os parâmetros de entrada corretos; e *validate(-)*, responsável por validar as saídas produzidas ao final da execução das N-versões de cada conjunto.

```
public interface NVersionedModule {
    public int getNbModules();
    public Object invokeModule(int i, Object[] params) throws ModuleExecutionFailureException;
    public Object validate(Object[] results) throws NVersionModuleFailureException ;
}
```

Figura 3.10: Interface para Conjuntos de N-versões

A classe **NVersionedModuleExample** (vide figura 3.11) é um exemplo de implementação da interface **NVersionedModule**, contendo ainda os métodos que implementam as versões do servidor da aplicação (conjunto de N-versões).

A classe **NVersionModule** (vide figura 3.12) contém o método $execute(-)$ responsável pela execução do conjunto de N-versões, seguindo os passos enumerados anteriormente.

Desta maneira, caberá ao projetista da aplicação implementar os métodos das interfaces **RecoverableBlock** e **NVersionedModule**, além dos métodos para os módulos redundantes contidos num bloco de recuperação e conjunto de N-versões. Note que, os outros métodos citados já estarão implementados nas suas respectivas classes.

```
public class NVersionedModuleExample implements NVersionedModule {
    public NVersionedModuleExample() { \\construtor
    }

    public int getNbModules() {
        return 2;
    }

    public Object invokeModule(int i, Object[] params) throws ModuleExecutionFailureException {
        if (params.length != 3) throw new ModuleExecutionFailureException();
        switch(i) {
            case 0:
              return new Integer(mul1(((Integer) params[0]).intValue(), ((Integer)params[1]).intValue(), ((Integer) params[2]).intValue()));
            case 1:
              return new Integer(mul2(((Integer) params[0]).intValue(), ((Integer)params[1]).intValue(), ((Integer) params[2]).intValue()));
            default:
              throw new ModuleExecutionFailureException();
        }
    }
    private int mul1(int delay, int x, int y) { \\versão 1
        try {
            System.out.println("NVersionModuleExample: mul1 sleeping for " + delay);
            Thread.sleep(delay);
        }
        catch (InterruptedException e) {
        }
        System.out.println("NVersionModuleExample: mul1 returning for " + x*y);
        return x*y;
    }

    private int mul2(int delay, int x, int y) { \\módulo alternativo
        try {
            System.out.println("NVersionModuleExample: mul2 sleeping for " + delay);
            Thread.sleep(delay);
        }
        catch (InterruptedException e) {
        }
        int result = 0;
        for (int i = 0; i < x; i++) {
            result += y;
        }
        System.out.println("NVersionModuleExample: mul2 returning for " + result);
        return result;
    }
    public Object validate(Object[] results) throws NVersionModuleFailureException {
        if (((Integer) results[0]).intValue() == ((Integer) results[1]).intValue()) {
            return results[0];
        }
        else  throw new NVersionModuleFailureException();
    }
}
```

Figura 3.11: Classe para Implementação dos Conjuntos de N-versões

Finalizando esta discussão, tem-se a classe **ApplicationTest** (vide figura 3.13) que contém o método *main(-)*, o qual mostra comoum bloco de recuperação e um conjunto de N-

```
public class NVersionModule {
    protected NVersionedModule nVersionedModule;
    public NVersionModule(NVersionedModule module) {
        this.nVersionedModule = module;
    }
    public Object execute(Object[] params) throws NversionModuleFailureException{
        int nVersions = this.nVersionedModule.getNbModules();
        Object[] results = new Object[nVersions];
        ThreadGroup group = new ThreadGroup("theGroupOfReplicas");
        for (int i = 0; i < nVersions; i++) {
            try {
                (new Thread(group, new RunnableModule(this.nVersionedModule, i, params, results))).start();
            }
            catch (Exception e) {
                results[i] = null;
            }
        }
        while (group.activeCount() != 0); //Espera até que todos as threads disparadas terminem suas execuções
        return this.nVersionedModule.validate(results);
    }
    private class RunnableModule implements Runnable {
        private NVersionedModule myModule;
        private int myNb;
        private Object[] params;
        private Object[] results;
        //Construtor
        public RunnableModule(NVersionedModule module, int moduleNb, Object[] params, Object[] results) {
            this.myModule = module;
            this.myNb = moduleNb;
            this.results = results;
            this.params = params;
        }
        //Executa cada uma das threads criadas
        public void run() {
            try {
                this.results[this.myNb] = this.myModule.invokeModule(this.myNb, this.params);
            }
            catch (ModuleExecutionFailureException e) {
                System.out.println("ParallelNVersionModule: " + e);
                this.results[this.myNb] = null;
            }
        }
    }
}
```

Figura 3.12: Classe para Execução dos Conjuntos de N-versões

versões podem ser instanciados e utilizados dentro de uma aplicação.

```
public class ApplicationTest {
    public static void main(String argv[]) {
        RecoverableBlock block = new RecoverableBlockExample( );
        NVersionedModule module = new NVersionedModuleExample();
        RecoveryBlock rb = new RecoveryBlock(block);
        NVersionModule nvm = new NVersionModule(module);
        Integer[] params = new Integer[3];

        // Inicializacao da variavel params
        params[0] = new Integer(1000);
        params[1] = new Integer(3);
        params[2] = new Integer(4);

       // Inicializacao da variavel params com valores do usuario
            if (argv.length > 0) {
                params[0] = new Integer(argv[0]);
                if (argv.length > 1) {
                    params[1] = new Integer(argv[1]);
                    if (argv.length > 2) {
                        params[2] = new Integer(argv[2]);
                    }
                }
            }
        System.out.println("Recovery block result: " + rb.execute(params));
        System.out.println("NVersion result: " + nvm.execute(params));

    }
}
```

Figura 3.13: Utilizando Blocos de Recuperação e Conjuntos de N-versões

3.4 Comparação entre os Serviços quanto à Disponibilidade e à Confiabilidade

Nesta seção será apresentado um estudo comparativo sobre as classes de serviços já discutidas (ver seção 3.2), considerando as probabilidades dos serviços funcionarem corretamente e apresentarem suas respectivas semânticas operacionais. Através deste estudo, objetiva-se identificar os principais fatores que influenciam os níveis de confiabilidade e disponibilidade dos serviços. Tal informação pode ser útil no sentido de estimar os custos de usar cada serviço para garantir um determinado requisito de confiança no funcionamento (confiabilidade ou disponibilidade). Sabendo dos custos associados a cada serviço de processamento, o projetista da aplicação terá subsídios adicionais para escolher o serviço mais adequado.

A probabilidade de um serviço funcionar corretamente, isto é, comportar-se de acordo com sua especificação (semântica operacional) ou falhar por parada (semântica de falha), é representada pela constante PC. Já a probabilidade de um serviço apresentar sua respectiva

semântica operacional, isto é, comportar-se de acordo com a abordagem dos blocos de recuperação ou programação em N-versões, é representada pela constante PO e toma como pressuposto que o referido serviço está correto ($PC = 1$)

Os valores de PC e PO sofrem influência direta das probabilidades associadas às suposições feitas sobre o comportamento dos processadores e dos mecanismos para tolerância a faltas de software, como também, do nível de redundância utilizado, ou seja, o número de processadores (N_{hw}) e o número de módulos redundantes ou versões (N_{sw}), utilizados na implementação dos respectivos mecanismos para tolerância a faltas de hardware e software.

A probabilidade dos processadores falharem por parada é muito alta, entretanto, é possível (mas, pouco provável) que estes falhem de forma arbitrária. Se um processador puder apresentar uma semântica de falha diferente daquela esperada, o mesmo acontecerá com o serviço de processamento provido através deste processador. Nesse caso, o fato de existir a possibilidade do processador não funcionar corretamente reflete sobre a probabilidade do serviço de processamento funcionar corretamente. Em outras palavras, PC nunca poderá ser superior à probabilidade de um processador apresentar a semântica de falha por parada (P_{fp}).

Em relação aos mecanismos para tolerância a faltas de software - blocos de recuperação (br), programação em N-versões com comparação dos resultados ($pnvC$) e programação em N-versões com votação majoritária dos resultados ($pnvV$)[7], a probabilidade destes funcionarem corretamente recai, no primeiro caso, sobre a probabilidade do teste de aceitação ser confiável (P_{ta}), e nos últimos, sobre a probabilidade da suposição de falha para o conjunto de N-versões ser satisfeita (P_{sf}), ou seja, não mais que f versões apresentarem faltas de concepção.

Desta maneira, as probabilidades PC_{pnvC}, PC_{pnvV}, e PC_{br} para os serviços de processamento em questão podem ser expressas da seguinte forma:

$$PC_{pnvC} = PC_{pnvV} = P_{fp} * P_{sf}; \text{ e}$$
$$PC_{br} = P_{fp} * P_{ta}.$$

Assumindo que a probabilidade dos processadores falharem por parada é muito alta

[7]As abreviaturas br, $pnvC$ e $pnvV$ serão usadas para associar o valor de uma probabilidade ou de um parâmetro a um serviço particular. Dessa forma, PC_{br}, por exemplo, representa o valor de PC para o serviço de processamento baseado em blocos de recuperação, enquanto f_{pnvC} representa o número máximo de versões incorretas, toleradas pelo serviço baseado em N-versões com comparação dos resultados.

($P_{fp} \approx 1$), para maximizar os valores de PC_{pnvC} e de PC_{pnvV} deve-se maximizar o valor de P_{sf}. Isto pode ser feito aumentando-se o valor de N_{sw}, dessa forma, será mais fácil satisfazer a suposição de falha para cada conjunto de N-versões. Em outras palavras, será mais difícil violar a suposição de falha f, onde $f_{pnvC} = N_{sw} - 1$ e $f_{pnvV} = \lfloor (N_{sw} - 1)/2 \rfloor$.

Seguindo um raciocínio análogo, é possível maximizar o valor de PC_{br} aumentando-se a confiabilidade do teste de aceitação, ou seja, maximizando o valor de P_{ta}. Isto pode ser feito através da simplificação do teste de aceitação, fazendo o mesmo verificar as condições mínimas que validam o resultado da computação realizada pelo bloco. De fato, simplificar o teste de aceitação pode implicar em torná-lo menos rigoroso, o que não seria aceitável para algumas aplicações. Portanto, esta estratégia pode comprometer a validade dos resultados produzidos por um bloco de recuperação.

Considerando que o serviço de processamento oferecido sempre funciona corretamente ($PC = 1$), será discutida a seguir, qual a probabilidade destes apresentarem suas respectivas semânticas operacionais (PO). Note que, para o serviço apresentar a semântica operacional, pelo menos um dos processadores sobre o qual este serviço executa, deve apresentar a semântica operacional. Além disso, o mecanismo utilizado para tolerar as faltas do software também deve apresentar a semântica operacional. As probabilidades de tais fatos ocorrerem são representadas, respectivamente, por PO_{hw} e PO_{sw}.

Supondo que a probabilidade de um processador qualquer não apresentar a semântica operacional é F_p, então, a probabilidade de pelo menos um processador apresentar a semântica operacional é dada por:

$$PO_{hw} = 1 - (F_p)^{N_{hw}}.$$

Para o serviço baseado no mecanismo dos blocos de recuperação, a probabilidade PO_{sw} é dada por:

$$PO_{sw(br)} = 1 - \prod_{i=1}^{N_{sw}} F_{br(i)},$$

onde, $F_{br(i)}$ é a probabilidade do i-ésimo módulo de um bloco de recuperação conter uma falta de concepção.

Considerando o serviço que utiliza o mecanismo da programação em N-versões, tem-se as seguintes expressões para $PO_{sw(pnvC)}$ e $PO_{sw(pnvV)}$:

$$PO_{sw(pnvC)} = 1 - max(F_{v(1)}, F_{v(2)}, \ldots, F_{v(N_{sw})}),$$

onde, $F_{v(i)}$ é a probabilidade da i-ésima versão conter uma falta de concepção; e

$$PO_{sw(pnvV)} = 1^{\dagger\dagger}.$$

Desta maneira, as probabilidades PO_{pnvC}, PO_{pnvV} e PO_{br} para os serviços de processamento definidos podem ser expressas da seguinte forma:

$$PO_{pnvC} = (1 - (F_p)^{N_{hw}}) * (1 - max(F_{v(1)}, F_{v(2)}, \ldots, F_{v(N_{sw})}));$$
$$PO_{pnvV} = 1 - (F_p)^{N_{hw}}; \text{ e}$$
$$PO_{br} = (1 - (F_p)^{N_{hw}}) * (1 - \prod_{i=1}^{N_{sw}} F_{br(i)}).$$

Para maximizar o valor de PO_{pnvV}, faz-se necessário diminuir a ocorrência de faltas físicas, as quais podem causar falhas do processador (ou seja, diminuir F_p). Isto pode ser feito através do isolamento físico e elétrico dos processadores, o que já é característico de um sistema distribuído. Outra possibilidade seria aumentar o número de processadores redundantes (N_{hw}). Dessa forma, seria mais difícil ocorrer uma situação em que nenhum dos processadores estivesse livre de falhas. Por outro lado, considerando os resultados apresentados em [GRAY & SIEWIOREK, 1991][9], provavelmente, o valor de F_p será menor que o valor de F_v. Nesse caso, PO_{pnvC} deve sofrer maior influência da probabilidade PO_{sw} do que da probabilidade PO_{hw}. Sendo assim, só será possível maximizar o valor de PO_{pnvC}, se o valor de F_v puder ser minimizado e isto é, perfeitamente, possível. De fato, a probabilidade de se obter uma versão livre de falhas varia entre 60% e 90% [LYU, 1995], então, F_v pode ser reduzido a valores próximos de 0.

Em relação a PO_{br}, quanto maior o valor de N_{sw} maior será a probabilidade de que pelo menos um módulo redundante esteja correto, permitindo que a operação do bloco seja finalizada com sucesso. No sentido de maximizar o valor de PO_{br}, deve-se adotar a mesma estratégia sugerida anteriormente, qual seja, além de aumentar o número de módulos redundantes em cada bloco (N_{sw}), aumentar também o número de processadores que executam réplicas do bloco de recuperação (N_{hw}).

[††] Note que, utilizando votação majoritária, as possíveis falhas serão mascaradas, portanto, a probabilidade da semântica operacional ser observada é igual a 1.

[9] Os resultados apresentados neste artigo indicam que as falhas nos sistemas computacionais são, em grande parte (entre 60% e 90%), causadas por faltas de concepção do software.

Os valores de PC e PO refletem, respectivamente, o nível de segurança na semântica de falha escolhida e os níveis de confiabilidade/disponibilidade dos serviços de processamento em questão. Sendo assim, maximizando estes valores é possível oferecer um maior grau de tolerância a faltas para as aplicações que utilizam tais serviços.

3.5 Conclusões

Os serviços de processamento propostos neste capítulo serão utilizados por aplicações com requisitos de confiança no funcionamento, isto é, aplicações que apresentam certa criticalidade e precisam tolerar faltas. Dessa forma, faz-se necessário garantir que os mecanismos para tolerância a faltas de hardware e software disponibilizados à aplicação através dos serviços em questão funcionem corretamente. Em outras palavras, os serviços de processamento devem ser validados.

A validação dos mecanismos para tolerância a faltas do software, consiste em garantir a correta execução dos blocos de recuperação e conjunto de N-versões que implementam os servidores da aplicação. Ou seja, validar o método $execute(-)$ das classes **RecoveryBlock** e **NVersionModule**. Note que o método $execute(-)$ é simples (poucas linhas de código) e portanto, as chances do mesmo conter faltas de concepção são pequenas. Além disso, este método não implementa um protocolo distribuído, todo o processamento é realizado localmente. Sendo assim, o método $execute(-)$ pode ser validado facilmente.

Por outro lado, a validação do mecanismo para tolerância a faltas de hardware, consiste em garantir a correta execução do processamento replicado de forma ativa (de acordo com a estratégia de replicação ativa). Essencialmente, isto significa validar os protocolos de gerência da redundância no grupo de processadores a partir do qual os serviços de processamento são providos. No caso dos serviços em questão, já que a semântica de falha de um bloco de recuperação ou conjunto de N-versões é, no pior caso, por parada (tal como a semântica de falha do processador), esta gerência é feita por meio de um protocolo para ordenação de mensagens que apresente as propriedades de acordo e ordem, requisitos da estratégia de replicação ativa. Esta tarefa já não é tão simples quanto a primeira (validar o método $execute(-)$), e normalmente apoia-se num modelo de sistema já definido. Vale salientar que o modelo de sistema escolhido deve descrever precisamente, ou o mais precisamente possível, o sistema real para o qual os protocolos são definidos. No caso dos serviços de pro-

cessamento em questão, deve-se optar por um modelo de sistema adequado para descrever sistemas distribuídos de prateleira.

No sentido de validar os serviços de processamento propostos, no próximo capítulo, será definido um protocolo para ordenação de mensagens, considerando a discussão acima sobre o modelo de sistema assumido.

Capítulo 4

Gerenciando a Redundância dos Processadores

4.1 Introdução

Para tolerar faltas dos processadores utilizando replicação ativa, faz-se necessário garantir que os processadores replicados tenham acesso às mesmas mensagens de entrada e numa mesma ordem. Estes requisitos podem ser atendidos por meio de um protocolo para ordenação de mensagens.

A ordenação de mensagens num grupo de processadores replicados consiste, essencialmente, no acordo sobre a ordem na qual as mensagens serão processadas por todos os membros do grupo. Sendo assim, a ordenação pode ser tratada como um problema de acordo. Como o consenso constitui uma abstração para os problemas de acordo [HURFIN ET AL., 1998], soluções para o problema da ordenação de mensagens podem ser obtidas através de soluções para o problema do consenso[1]. Nesse caso, isto não significa que um protocolo de consenso irá resolver o problema da ordenação, mas, será requisito fundamental para tal (informações mais detalhadas sobre o problema do consenso podem ser encontradas no Apêndice A).

O problema do consenso não possui solução determinística em sistemas assíncronos, mesmo considerando que apenas um único processador falha por parada. Isto é conhecido como o resultado de impossibilidade FLP (Fischer-Lynch-Paterson) [FISCHER ET AL., 1985]. Tal resultado é fundamentado na seguinte observação: devido à ausência de limites para os atrasos na transmissão de mensagens e escalonamento de tarefas (como é caracterís-

[1] O consenso pode ser utilizado em soluções para problemas de acordo, tais como, difusão atômica de mensagens (*atomic broadcast*) [CHANDRA & TOUEG, 1996] e comprometimento atômico (*atomic commitment*) [GERRAOUI, 1995].

tico do modelo de sistema assíncrono), é impossível diferenciar um processador que falhou por parada de outro com processamento lento, mas livre de falhas. A partir desta observação é possível estabelecer uma relação de causa-efeito entre a incerteza nos atrasos para transmissão de mensagens/escalonamento de tarefas e a impossibilidade de detectar falhas por parada dos processadores num determinado sistema.

Vale salientar que, o resultado de impossibilidade FLP não se aplica aos ao modelo de sistema síncrono, entretanto, a maioria dos sistemas reais não são síncronos. Por esta razão, muitos pesquisadores têm trabalhado no sentido de descobrir as mínimas restrições que devem ser impostas sobre as garantias de sincronismo providas por um modelo de sistema a fim de possibilitar a resolução do problema do consenso, com soluções determinísticas, sobre tal modelo. Deste esforço, foram definidos vários modelos que restringem o modelo de sistema assíncrono, dentre os quais figuram o modelo de sistema assíncrono temporizado [CRISTIAN & FETZER, 1999] e o modelo de sistema assíncrono com detectores de falhas não confiáveis [CHANDRA & TOUEG, 1996]. Os modelos citados utilizam abordagens diferentes para contornar o resultado de impossibilidade FLP.

O modelo assíncrono com detectores de falhas não confiáveis trata o efeito, ou seja, assume-se que existe um "oráculo", denominado detector de falhas, que é capaz de informar sobre os processadores suspeitos de terem sofrido uma falha por parada. As informações providas por este oráculo nem sempre são confiáveis, pois o mesmo pode cometer erros, seja suspeitando dos processadores corretos (livres de falhas) ou não suspeitando dos processadores que realmente falharam. Mesmo assim, tais informações são suficientemente precisas para permitir a resolução do problema do consenso com soluções determinísticas.

As soluções para o problema do consenso projetadas sobre o modelo assíncrono com detectores não confiáveis caracterizam-se pela simplicidade, o que facilita o processo de validação dessas soluções. Isto acontece porque os detectores de falhas não confiáveis constituem uma abstração poderosa e de alto nível. Porém, observa-se que é impossível implementar tal abstração num sistema assíncrono, caso contrário, o resultado de impossibilidade FLP seria inválido. Sendo assim, soluções práticas exigiriam suposições extras para o modelo de sistema em questão.

Por outro lado, o modelo assíncrono temporizado trata a causa, ou seja, assume-se a existência de limites não confiáveis para os atrasos na transmissão de mensagens e escalon-

amento de tarefas. Este modelo foi definido com base na observação de que, para muitos sistemas é plausível estabelecer limites que serão obedecidos apenas na maior parte do tempo, além disso, na prática, tais sistemas alternam entre curtos períodos de instabilidade e longos períodos de estabilidade. Durante um período de instabilidade, os atrasos na transmissão de mensagens e escalonamento de tarefas podem ultrapassar seus respectivos limites o que determina a ocorrência de uma falha por desempenho. Em contrapartida, durante um período de estabilidade o sistema comporta-se de forma síncrona, garantindo o progresso na computação realizada pelo mesmo. Sendo assim, é possível resolver o problema do consenso sobre o modelo assíncrono temporizado, sabendo que as soluções implementadas funcionam corretamente quando o sistema estiver estável. Nesse caso, faz-se necessário especificar um modo de execução seguro para os protocolo projetados sobre tal modelo, haja vista a existência dos períodos de instabilidade.

O modelo assíncrono temporizado também incorpora o conceito de suposições de progresso cuja finalidade é prover uma medida da probabilidade do sistema estar estável num certo instante de tempo. Outro aspecto importante refere-se ao fato de que o referido modelo é considerado o mais apropriado para descrever os sistemas distribuídos de prateleira, além disso, a noção de tempo provida possibilita a especificação de serviços temporizados os quais não poderiam ser especificados em outro modelo assíncrono.

Se por um lado o modelo assíncrono temporizado permite resolver na prática o problema do consenso, observa-se que as soluções propostas são complexas (ou, pelo menos, mais complexas do que aquelas apresentadas para o modelo assíncrono com detectores de falhas não confiáveis) e, consequentemente, difíceis de serem validadas. Adicionalmente, faz-se necessário modificar a especificação do problema a fim de incorporar um modo de operação seguro para os períodos de instabilidade e, ainda, deve-se definir uma suposição de progresso apropriada.

Face a esta realidade, no presente capítulo será proposta uma abordagem híbrida para projetar e validar o protocolo de ordenação requerido. A idéia é combinar os dois modelos discutidos acima, aproveitando suas respectivas facilidades. Nesse caso, utiliza-se o modelo de sistema assíncrono temporizado para descrever o sistema distribuído de prateleira sobre o qual será implementado o protocolo d e ordenação, mas, ao invés de projetar e validar o protocolo considerando este modelo, identifica-se, inicialmente, as abstrações que podem

facilitar o projeto e validação do protocolo (por exemplo, detectores de falhas não confiáveis, serviços para comunicação tolerante a faltas, etc). Então, projeta-se protocolos simples que implementem tais abstrações (serviços), os quais serão validados considerando o modelo assíncrono temporizado. Finalmente, utiliza-se o modelo assíncrono temporizado estendido com os serviços extras (já válidados), a fim de projetar um protocolo de ordenação que seja simples e prático.

O restante deste capítulo está organizado da seguinte forma. Na seção 4.2, apresenta-se o modelo de sistema assíncrono temporizado e os serviços extras requeridos para resolver o problema da ordenação de mensagens sobre este modelo. Em seguida, na seção 4.3, mostra-se os protocolos que implementam os serviços extras requeridos. Na seção 4.4, tem-se a validação desses protocolos.

4.2 Estendendo o Modelo Assíncrono Temporizado

4.2.1 O Modelo Assíncrono Temporizado

O modelo assíncrono temporizado foi definido formalmente em [CRISTIAN & FETZER, 1999] e tem como principais características:

Especificação dos serviços. Todos os serviços são temporizados, ou seja, a especificação de um serviço descreve a saída produzida como resultado da computação realizada sobre cada entrada de dados (o quê), a seqüência de passos executados a fim de obter tais resultados (como) e o tempo gasto para que cada resultado seja produzido (quando).

Garantias de sincronismo. São definidos temporizadores para os atrasos na transmissão de mensagens e escalonamento de tarefas, cujos limites são expressos através das constantes δ e σ, respectivamente. Quando o atraso na transmissão de mensagens ou no escalonamento de tarefas ultrapassa esses limites, caracteriza-se, então, a ocorrência de uma falha por desempenho. Note que, diferentemente do modelo síncrono, os valores de δ e σ não são confiáveis, pois, em determinadas circunstâncias estes valores podem ser ultrapassados.

Relógios Físicos. Todos os processos têm acesso ao relógio físico do processador onde estes executam. Assume-se que se um processador é correto, então seu relógio físico é

correto, isto é, apresenta uma taxa de desvio em relação ao tempo real limitado por uma constante ρ.

Semântica de Falha. A comunicação entre os processadores é dada através de um serviço de datagrama cuja semântica de falha é por desempenho ou omissão. Dessa forma, o atraso na entrega de uma mensagem pode ultrapassar o valor especificado para tal (falha por desempenho) ou a mensagem pode não alcançar o seu destinatário (falha por omissão), mas o conteúdo da mensagem nunca será violado. Em relação aos processadores (e processos que executam sobre os mesmos), estes podem falhar por parada ou desempenho.

Suposição de Progresso. Uma suposição de progresso (*progress assumption*) estabelece que, após um período de instabilidade, o sistema torna-se estável por um intervalo de tempo limitado, suficiente para haver progresso na computação realizada pelo mesmo. Em outras palavras, durante o tempo de missão do sistema, sempre haverá condições favoráveis para o mesmo progredir em sua execução. O sistema é dito estável quando, pelo menos, E processadores corretos comunicam-se entre si e processam mensagens obedecendo aos atrasos δ e σ especificados para tal. De fato, o uso de suposições de progresso pode ser visto como uma maneira de exigir maior rigor na escolha dos valores das constantes δ e σ, no modelo assíncrono temporizado.

4.2.2 Suposições Extras para Resolver o Problema da Ordenação de Mensagens

Uma solução para o problema da ordenação de mensagens pode ser implementada a partir da execução de difusões atômicas disparadas por um conjunto de processos ordenadores, os quais recebem alguma mensagem da aplicação ou de outro processo ordenador [BRASILEIRO, 1995]. Sendo assim, é possível resolver tal problema a partir de uma solução para o problema da difusão atômica de mensagens, como aquela definida em [CHANDRA & TOUEG, 1996].

A solução de Chandra e Toueg (informações mais detalhadas sobre esta solução podem ser encontradas no Apêndice A) foi projetada sobre o modelo de sistema assíncrono com detectores de falhas não confiáveis e requer um serviço de consenso que usa detectores de falhas da classe $\Diamond S$ e outro para difusão confiável de mensagens. Ainda sobre o serviço de

consenso, é interessante ressaltar que o mesmo faz uso de um serviço para difusão confiável de mensagens. De acordo com a referida solução, o problema da ordenação num grupo de processadores replicados será resolvido da seguinte forma: o serviço para difusão confiável de mensagens garante que todos os processadores corretos do grupo irão receber o mesmo conjunto de mensagens e o consenso garante que a decisão sobre a ordem de processamento das mensagens no grupo de processadores replicados será a mesma.

Aplicando a abordagem híbrida, o protocolo de ordenação requerido será projetado e validado da seguinte forma: inicialmente, assume-se o modelo assíncrono temporizado a fim de projetar e validar um serviço para difusão confiável de mensagens e outro para detecção de falhas equivalente àquele provido por um detector da classe $\Diamond S$, então, assumindo o modelo assíncrono temporizado estendido com tais serviços pode-se utilizar o mesmo protocolo apresentado em [CHANDRA & TOUEG, 1996] para resolver o problema da ordenação.

Todas as soluções disponíveis para o problema do consenso que são baseadas em detectores de falhas da classe $\Diamond S$ requerem um número majoritário de processadores corretos, sendo assim, a **suposição de falha** para o sistema é dada por $f = \lfloor N/2 \rfloor$, onde N é o número de processadores responsáveis pelo provimento do serviço de consenso e f equivale ao número de processadores que podem falhar.

Em relação à **suposição de progresso**, faz-se necessário conhecer as restrições de um sistema estável. Nesse caso, um período de estabilidade será caracterizado pela existência pela existência de um conjunto majoritário de processadores corretos que se comunicam de forma síncrona (atraso na comunicação não ultrapassa o valor da constante Δ[2]), durante um intervalo de tempo suficientemente longo para que seja observado progresso na computação sendo realizada pelo sistema. Este período pode ser representado pela tupla (P, I), onde P é um conjunto de processadores estáveis, composto por, pelo menos $f + 1$ processadores corretos e conectados entre si; e I é o intervalo de tempo no qual o sistema permanece estável, ou seja, existe um conjunto P.

O intervalo de tempo I é definido pelos limites TI e TF, tal que $I = [TI, TF]$, onde, $TF - TI > \beta$[3]. Nesse caso, o n-ésimo período de estabilidade pode ser representado da

[2]A constante Δ representa o atraso máximo na transmissão de uma mensagem, desde sua origem até o recebimento no destinatário. Esta constante engloba os atrasos na transmissão da mensagem pela rede (δ) e no escalonamento do processo destino envolvido na comunicação (σ)), sendo assim, tem-se $\Delta = \delta + \sigma$.

[3]A constante β representa o temporizador para o tempo mínimo de estabilidade, ou seja, o menor intervalo de tempo I no qual o sistema permanece estável.

seguinte forma: $PE_n = (P_n, I_n)$, $I_n = [TI_n, TF_n]$; além disso, $TF_x \leq TI_{x+1}$.

É também importante definir o significado de processador correto e processador defeituoso. Um processador correto é aquele livre de falhas e que, freqüentemente (em intervalos de tempo inferiores a Υ[4]), integra um conjunto de processadores estáveis P. Isto é, se o processador p_i é correto, então, existem P_x e P_y, $y > x$, tal que, p_i pertence a P_x e a P_y e $TI_y - TI_x \leq \Upsilon$. Note que, não necessariamente tem-se $y = x + 1$. Dessa forma, em qualquer período de estabilidade $PE = (P, I)$ podem existir processadores corretos que não pertencem ao conjunto P. Vale salientar que, um processador correto pode falhar durante um período de estabilidade ou instabilidade, desde que a suposição de falha não seja violada. Além disso, quando um processador correto falha durante um período de estabilidade PE_x, este processador não mais será considerado como um processador que pertence a P_x.

Por outro lado, um processador defeituoso é aquele que sofreu uma falha por parada ou permaneceu um intervalo de tempo muito longo (superior a Υ) sem pertencer a algum conjunto P. Nesse caso, se um processador falhar, significa que o mesmo tornou-se defeituoso e não mais poderá pertencer a um conjunto de processadores estáveis. A **semântica de falha** estabelecida para os processos do sistema é também por parada, sendo assim, pode-se considerar que o software do sistema e o processador sobre o qual o mesmo executa, formam uma única entidade que falha por parada. Como os protocolos a serem definidos nesse capítulo fazem parte do software do sistema, nas seções subseqüentes, referências a um processador devem ser entendidas como referências ao conjunto processador mais respectivo software de sistema. Já os processos da aplicação podem falhar de forma arbitrária, devido às faltas de concepção.

O **serviço para difusão confiável de mensagens** assumido satisfaz as propriedades descritas abaixo:

Acordo. Se um processador correto entrega uma mensagem m, então, todos os processadores corretos também entregam esta mensagem.

Validade. Se um processador correto difunde uma mensagem m, então, todos os processadores corretos entregam esta mensagem.

[4]A constante Υ representa o temporizador para o tempo máximo de desconexão, ou seja, o maior intervalo de tempo no qual um processador correto pode ficar sem pertencer a um conjunto de processadores estáveis P.

Integridade. Para qualquer mensagem m, todo processador entrega m uma única vez e, apenas, se esta mensagem foi difundida por algum processador.

Terminação. Uma mensagem m difundida por um processador p_i no tempo real t será entregue por todo processador correto p_j no tempo real t_j, tal que $\mid t - t_j \mid \leq \Sigma$.

O **serviço para detecção de falhas** assumido satisfaz as seguintes propriedades:

Completude. Se um processador falha por parada no tempo t_p, então, esta falha é detectada por todos os processadores corretos até, no máximo, $t_p + \Omega$.

Precisão. Para todo instante de tempo t, existe um tempo t', $t' > t$, a partir do qual uma maioria de processadores corretos não irá suspeitar de algum processador correto por, no mínimo, Θ unidade de tempo e $t' - t < \Gamma$.

Na verdade, as propriedades de completude e precisão descritas acima são semelhantes àquelas que caracterizam os detectores de falhas da classe B [CHANDRA ET AL., 1996b], portanto, são propriedades mais restritivas do que aquelas da classe $\Diamond S$. Entretanto, qualquer protocolo de consenso baseado na classe $\Diamond S$ também resolve o consenso utilizando uma classe de detectores mais restritiva [CHANDRA & TOUEG, 1996].

4.3 Implementando os Serviços Extras do Modelo Assíncrono Temporizado Estendido

4.3.1 Um Protocolo para Difusão Confiável de Mensagens

Sob o ponto de vista prático, resolver o problema da difusão confiável significa garantir que nenhuma mensagem transmitida pela rede física, por meio de difusão, será perdida. No caso do modelo de sistema enfocado neste trabalho, deve-se levar em consideração que o serviço de comunicação pode falhar por omissão[5], sendo assim, mensagens transmitidas pela rede podem ser perdidas. Nesse caso, é preciso incorporar ao sistema um mecanismo para recuperação de falhas na comunicação entre processadores, a fim de garantir a confiabilidade do serviço de difusão.

[5]O serviço de comunicação pode falhar por desempenho também, mas, o mecanismo usado para tratar das falhas por omissão é suficiente para tratar também de falhas de desempenho.

A solução proposta para resolver o problema da difusão confiável de mensagens será descrita por meio do protocolo 1 que é relativamente simples. Este protocolo é responsável pela difusão de mensagens e recuperação de falhas na transmissão de mensagens pela rede.

O mecanismo para recuperação de falhas utilizado é baseado na recuperação de mensagens perdidas[6]. A idéia deste mecanismo é que, periodicamente, todos os processadores corretos informem, uns aos outros, as mensagens recebidas e entregues. Dessa forma, quando um processador correto p_i perceber que algum outro processador p_j entregou uma mensagem que ainda não foi entregue por p_i, será requisitada a retransmissão desta mensagem a p_j.

O protocolo 1 divide-se em cinco módulos (executados em paralelo) e possui uma primitiva de acesso, a saber:

Primitiva $difunde(m)$**.** Inicia a difusão confiável de uma mensagem m. Mensagens difundidas são entregues localmente através de uma chamada à função de retorno registrada pela aplicação que deseja receber estas mensagens (linha 13). Mensagens entregues são armazenadas no conjunto $MsgEntregues$.

Módulo para Entrega de Mensagens (EntregaMsg). Este módulo é responsável pela entrega das mensagens que foram difundidas por algum processador do grupo.

Módulo para envio do estado interno de um processador (TransmiteEstado). Este módulo é responsável pela difusão (não confiável) do estado interno de cada processador. O estado interno de um processador é representado pelos identificadores das mensagens pertencentes ao conjunto $MsgEntregues$ e será enviado a todos os processadores do grupo a cada δ_{R1} unidades de tempo.

Módulo para recepção do estado interno de um processador (RecebeEstado). Este módulo é responsável pela recepção do estado interno dos outros processadores, isto é, o conjunto dos identificadores das mensagens já entregues pelos outros processadores. A informação recebida é utilizada para atualizar o conteúdo do conjunto $IdNaoEntregues$.

[6]Mecanismos para recuperação de falhas podem ser implementados utilizando outras estratégias. Uma opção, por exemplo, seria fazer o remetente da mensagem (emissor) retransmiti-la, a cada t unidades de tempo, até receber uma mensagem de confirmação, proveniente do receptor (destinatário) daquela mensagem, indicando que a mesma foi entregue com sucesso.

Protocolo 1 Protocolo para Difusão Confiável de Mensagens Executado por p_i

(1) /* Variáveis Globais */
(2) $MsgEntregues = \{\}$
(3) $IdNaoEntregues = \{\}$

(4) difunde(m)
(5) **início**
(6) envia m $\forall p_j$ $(j \neq i)$
(7) $MsgEntregues = MsgEntregues \cup \{m\}$
(8) entrega m para a aplicação a qual se destina
(9) **fim**

(10) || Módulo: EntregaMsg
(11) **quando** recebe a mensagem m
(12) $MsgEntregues = MsgEntregues \cup \{m\}$
(13) entrega m para a aplicação a qual se destina
(14) **fim quando**

(15) || Módulo: TransmiteEstado
(16) **enquanto** VERDADE **faça**
(17) $m = (i, \{m.id \mid m \in MsgEntregues\})$
(18) envia m $\forall p_j$ $(j \neq i)$
(19) **espera até** δ_{R1} expirar
(20) **fim enquanto**

(21) || Módulo: RecebeEstado
(22) **enquanto** VERDADE **faça**
(23) recebe $e = (emissor, Ids)$
(24) $IdEntreguesLocal = \{m.id \mid m \in MsgEntregues\}$
(25) **paratodos** $id \in e.Ids \mid id \notin IdEntreguesLocal$ **faça**
(26) $IdNaoEntregues = IdNaoEntregues \cup \{(e.emissor, id)\}$
(27) **fim para**
(28) **fim enquanto**

(29) || Módulo: RecuperaMsg
(30) **enquanto** VERDADE **faça**
(31) **enquanto** $IdNaoEntregues \neq \{\}$ **faça**
(32) **paratodos** $(p, id) \in IdNaoEntregues$ **faça**
(33) requisita mensagem $m \mid m.id = id$ de p
(34) **fim para**
(35) **enquanto** δ_{R2} não expirar **faça**
(36) recebe m
(37) $ReqAtendidas = \{r = (emissor, id) \mid r \in IdNaoEntregues \wedge r.id = m.id\}$
(38) $IdNaoEntregues = IdNaoEntregues - ReqAtendidas$
(39) $MsgEntregues = MsgEntregues \cup \{m\}$
(40) entrega m para a aplicação a qual se destina
(41) **fim enquanto**
(42) **fim enquanto**
(43) **fim enquanto**

(44) || Módulo: RetransmiteMsg
(45) **enquanto** VERDADE **faça**
(46) recebe requisição $r = (emissor, id)$
(47) envia $m \mid m.id = r.id \wedge m \in MsgEntregues$ para $r.emissor$
(48) **fim enquanto**

Módulo para recuperação de mensagens (RecuperaMsg). Este módulo é responsável por requisitar, a algum processador, a retransmissão das mensagens cujos identificadores foram armazenados no conjunto $IdNaoEntregues$. Os elementos de $IdNaoEntregues_i$ são tuplas $(emissor, id)$ contendo o identificador de uma mensagem ainda não recebida por p_i (id) e o identificador de um processador que já entregou esta mensagem ($emissor$). A retransmissão de tal mensagem é requisitada a qualquer processadore que já tenha entregue a mesma. O tempo de espera pela retransmissão de uma mensagem é representado pela constante δ_{R2}. Tão logo sejam recebidas as mensagens requisitadas, o conteúdo dos conjuntos $IdNaoEntregues$ e $MsgEntregues$ será atualizado.

Módulo para retransmissão de mensagens (RetransmiteMsg). Este módulo é responsável pela retransmissão de mensagens perdidas. Tão logo o pedido de retransmissão de uma mensagem m seja recebido, esta mensagem é enviada ao processador requisitante.

A difusão de mensagens, por meio da chamada à primitiva $difunde(m)$, combinada ao mecanismo para recuperação de falhas descrito no protocolo 1, é confiável. Isto será provado quando da validação do serviço para difusão confiável, na seção 4.4.1.

4.3.2 Um Protocolo para Detecção de Falhas

O modelo de sistema apresentado na seção 4.2 assume um serviço para detecção de falhas mais restritivo do que aquele oferecido pelos detectores da classe $\Diamond S$. Tal serviço é implementado pelo protocolo 2. A idéia é que, periodicamente (a cada δ_E unidades de tempo), os processadores corretos possam difundir mensagens do tipo $EuEstouOperante$ informando sobre o estado corrente dos mesmos. Todo processador correto p_i irá suspeitar de um processador p_j se não receber mensagens $EuEstouOperante$ deste processador, δ_F unidades de tempo após a última unidade de tempo em que foi recebida uma mensagem $EuEstouOperante$ do mesmo. Entretanto, quando uma nova mensagem $EuEstouOperante$ for recebida de p_j, este processador não mais será considerado suspeito por p_i.

O protocolo 2 divide-se em três módulos e possui uma primitiva de acesso, a saber:

Protocolo 2 Protocolo para Detecção de Falhas Executado por p_i

(1) /* Variáveis Globais */
(2) $Psuspeitos = \{\}$

(3) /* Inicialização de variáveis locais */
(4) **paratodos** $p_j, j \neq i$ **faça**
(5) $\delta_r(p_j)$ = $valor_{relogiofisicolocal}$
(6) **fim para**

(7) || Módulo: EnviaMsgOperante
(8) **enquanto** VERDADE **faça**
(9) $m = (i, EuEstouOperante)$
(10) envia m $\forall p_j$ $(j \neq i)$
(11) **espera até** δ_E expirar
(12) **fim enquanto**

(13) || Módulo: DetectaFalha
(14) **enquanto** VERDADE **faça**
(15) **espera até** δ_C expirar
(16) $NovoPsuspeitos = \{\}$
(17) **paratodos** $p_j, j \neq i$ **faça**
(18) **se** $\delta_r(p_j) < valor_{relogiofisicolocal} - \delta_F$ **então**
(19) $NovoPsuspeitos = NovoPsuspeitos + \{p_j\}$
(20) **fim se**
(21) **fim para**
(22) $Psuspeitos = NovoPsuspeitos$
(23) **fim enquanto**

(24) || Módulo: RecebeMsgOperante
(25) **quando** receber mensagem $m = (emissor, EuEstouOperante)$
(26) $\delta_r(m.emissor) = valor_{relogiofisicolocal}$
(27) **fim quando**

(28) suspeitos()
(29) **início**
(30) retorna $Psuspeitos$
(31) **fim**

Primitiva $suspeitos()$**.** Retorna o conjunto $Psuspeitos$, contendo os processadores suspeitos de terem sofrido uma falha por parada.

Módulo para envio de mensagens $EuEstouOperante$ **(EnviaMsgOperante).** Este módulo é responsável pela difusão (não confiável) de mensagens $EuEstouOperante$ no grupo de processadores. Isto ocorre a cada δ_E unidades de tempo.

Módulo para detecção de falhas (DetectaFalha). Este módulo é responsável pela detecção de processadores que falharam por parada. A cada δ_C unidades de tempo, é verificado se existe algum processador que deve ser considerado suspeito e armazenado no conjunto $Psuspeitos$. Um processador p_j é considerado suspeito por outro processador p_i, quando passa mais do que δ_F unidades de tempo sem enviar mensagens $EuEstouOperante$ para p_i. Ou seja, o tempo transcorrido desde a última vez que p_i recebeu uma mensagem $EuEstouOperante$ proveniente de p_j (valor de $\delta_r(p_j)$), é superior a δ_F unidades de tempo (essa verificação é descrita pela linha 18). Em cada rodada de detecção, monta-se o conjunto $NovoPsupeitos$ e atualiza-se o conjunto $Psuspeitos$. Se na rodada x um processador p_j for considerado suspeito, este será inserido em $NovoPsuspeitos$ e, consequentemente, em $Psuspeitos$. Porém, na rodada $x + 1$, p_j pode não mais pertencer ao conjunto $Psuspeitos$ (caso mensagens $EuEstouOperante$ enviadas por p_j sejam recebidas por p_i no futuro).

Módulo para recebimento de mensagens $EuEstouOperante$ **(RecebeMsgOperante).** Quando for recebida uma mensagem $EuEstouOperante$ de algum processador p_j, atualiza-se o vetor δ_r, na posição j, com o valor do relógio local. Este vetor guarda, para cada processador, o tempo no qual a última mensagem $EuEstouOperante$ foi recebida do mesmo.

A validação do serviço para detecção de falhas, descrito pelo protocolo 2, será tratada na seção 4.4.2.

4.4 Validando os Serviços Extras do Modelo Assíncrono Temporizado Estendido

A validação dos protocolos para difusão confiável de mensagens e detecção de falhas, será enfocada nesta seção. O objetivo é mostrar que as propriedades descritas na seção 4.2.2 são garantidas pelos respectivos protocolos. Desta maneira será possível afirmar que os mesmos funcionam corretamente, assegurando as restrições impostas para o modelo assíncrono temporizado estendido proposto.

4.4.1 Validando o Protocolo para Difusão Confiável de Mensagens

A difusão confiável de mensagens caracteriza-se pelas propriedades de Acordo, Validade, Integridade e Terminação, descritas na seção 4.2.2. O protocolo de difusão proposto satisfaz essas propriedades, como será demonstrado a seguir, desde que $0 < \delta_{R1} < \beta - 3 * \Delta$ e $\beta > 3 * \Delta$.

Lema 4.1 *Toda mensagem m que foi entregue por um processador correto pertencente a um conjunto P_x, no tempo $t < TF_x - \delta_{R1} - 3 * \Delta$, será entregue por todos os outros processadores corretos pertencentes a P_x, no máximo, até TF_x.*

Prova. Suponha que um processador correto p_i, pertencente a P_x, entregou uma mensagem m no tempo $t < TF_x - \delta_{R1} - 3 * \Delta$. De acordo com o protocolo 1, periodicamente (a cada δ_{R1} unidades de tempo), todo processador correto envia o seu estado interno para os outros processadores. Sendo assim, se $t \geq TI_x$ e sabendo que durante o intervalo de estabilidade $I_x = [TI_x, TF_x]$ todos os processadores corretos pertencentes a P_x comunicam-se de forma síncrona, no máximo até $t + \delta_{R1} + \Delta$, todos os processadores p_j, $p_j \in P_x$, irão receber o estado interno de p_i indicando que p_i já entregou a mensagem m. Caso m não pertença a $MsgEntregues_j$ (isto é, m pertence ao conjunto $IdNaoEntregues_j$, linha 32 - protocolo 1), p_j requisita de p_i a retransmissão de m, e esta mensagem será entregue, no máximo, após $2 * \Delta$ unidades de tempo, ou seja, até $t + \delta_{R1} + 3 * \Delta < TF_x$ e, dessa forma, dentro do intervalo I_x.

Por outro lado, se $t < TI_x$, então, no máximo até $TI_x + \delta_{R1} + \Delta$, todo processador p_j, $p_j \in P_x$, irá receber o estado interno de p_i indicando que p_i já entregou a mensagem m. Caso m não pertença a $MsgEntregues_j$, p_j requisita de p_i a retransmissão de m, e esta mensagem será entregue, no máximo, após $2 * \Delta$ unidades de tempo, ou seja, até $TI_x + \delta_{R1} + 3 * \Delta$.

Como $\delta_{R1} < \beta - 3 * \Delta$ e $\beta < TF_x - TI_x$, então, $TI_x + \delta_{R1} + 3 * \Delta < TF_x$, e todo processador correto p_j pertencente a P_x, que não entregou uma mensagem m, irá fazê-lo dentro do intervalo de estabilidade I_x.

Lema 4.2 *Uma mensagem m que foi entregue por todos os processadores corretos pertencentes a P_x, será entregue por todos os processadores corretos pertencentes a P_y, $y > x$, no máximo até $TI_y + \delta_{R1} + 3 * \Delta$.*

Prova. Suponha que uma mensagem m foi entregue por todos os processadores corretos pertencentes a P_x no intervalo de estabilidade I_x, ou seja, no tempo máximo $t \leq TF_x$. Como $N = 2 * f + 1$ e qualquer conjunto estável de processadores P contém, pelo menos, $f + 1$ processadores corretos, existe, no mínimo, um processador correto p_i que pertence a P_x e a P_y, $y > x$. Suponha, ainda, que p_i entregou m no tempo $t_i \leq t \leq TF_x \leq TI_y$, então, todo processador correto $p_j \in P_y$ receberá o estado interno de p_i, no máximo até $TI_y + \delta_{R1} + \Delta$, indicando que p_i já entregou a mensagem m. Caso m não pertença ao conjunto $MsgEntregues_j$, p_j requisita de p_i a retransmissão de m e esta mensagem será entregue, no máximo, após $2 * \Delta$ unidades de tempo, isto é, em $TI_y + \delta_{R1} + 3 * \Delta$. Já que, por definição, $\delta_{R1} < \beta - 3 * \Delta$ e $\beta < TF_y - TI_y$, tem-se $TI_y + \delta_{R1} + 3 * \Delta < TF_y$ e todo processador correto p_j pertencente a P_y que ainda não entregou a mensagem m, irá fazê-lo até $TI_y + \delta_{R1} + 3 * \Delta$.

Lema 4.3 (Integridade) *Toda mensagem m entregue por um processador correto foi difudida por algum processador e é entregue apenas uma vez.*

Prova. A prova segue diretamente do protocolo 1. De acordo com este protocolo, apenas as mensagens passadas como parâmetro para a primitiva $difunde(m)$ são retransmitidas. Além disso, uma vez que uma mensagem m é entregue, esta é armazenada em $MsgEntregues$, então, qualquer cópia de m recebida no futuro será descartada.

Lema 4.4 (Acordo) *Toda mensagem m entregue por um processador correto, será entregue por todos os processadores corretos.*

Prova. Seja p_i um processador correto que entregou m e t o tempo real no qual p_i entregou esta mensagem. Então, existe um período de estabilidade $PE_x = (P_x, I_x)$, tal que, $t < TF_x - \delta_{R1} - 3 * \Delta$ e p_i pertence a P_x. Pelo lema 4.1, todos os processadores corretos pertencentes a P_x, entregarão m, no máximo, até TF_x. Por definição, para todo processador correto p_j, $p_j \notin P_x$, existirá um período de estabilidade $PE_y = (P_y, I_y)$, $y > x$, tal que, p_j

pertence a P_y. Pelo lema 4.2, p_j entregará m. Sendo assim, todos os processadores corretos entregarão a mensagem m.

Lema 4.5 (Validade) *Toda mensagem m cuja difusão foi iniciada por um processador correto, será entregue por todos os processadores corretos.*

Prova. Seja p_i um processador correto que iniciou a difusão de uma mensagem m. De acordo com o protocolo 1, p_i entrega m após difundí-la. Pelo lema 4.4, todos os processadores corretos entregarão a mensagem m.

Lema 4.6 (Terminação) *Uma mensagem m que foi difundida por um processador correto no tempo t, será entregue por todos os processadores corretos, no máximo, até $t + \Sigma$, onde $\Sigma \leq 2 * (\Upsilon - \beta) + \delta_{R1} + 4 * \Delta$*

Prova. Suponha que um processador p_i correto inicia a difusão de uma mensagem m no tempo t. A fim de determinar o atraso para a entrega da mensagem m, considera-se três casos possíveis:

1. existe um período de estabilidade PE_x tal que $p_i \in PE_x$ e $TI_x \leq t \leq TF_x - \Delta$: nesse caso, já que a comunicação é síncrona durante um período de estabilidade (atraso na comunicação entre processadores obedece ao limite Δ, estabelecido), de acordo com o protocolo 1, todos os processadores corretos pertencentes a P_x entregarão m antes do tempo TF_x. Além disso, pelo lema 4.2, para todo processador correto p_j, $p_j \notin P_x$, existe um tempo t_j no qual p_j entregará m, tal que, $tj \leq TI_y + \delta_{R1} + 3 * \Delta$ e $y > x$. Sendo assim, o atraso para a entrega de m é dado por $t_j - t < TI_y + \delta_{R1} + 3 * \Delta - TI_x$. Note que, como p_j é correto e $p_j \notin P_x$, na pior das hipóteses $p_j \in P_{x-1}$ e $TF_{x-1} = TI_x$. Dessa forma, tem-se $TI_y - TI_{x-1} \leq \Upsilon$ e $TI_x - TI_{x-1} > \beta$, portanto, $TI_y - TI_x < \Upsilon - \beta$. Então, $\Sigma_{(1)} = t_j - t < \Upsilon - \beta + \delta_{R1} + 3 * \Delta$.

2. não existe um período de estabilidade PE_x tal que $p_i \in P_x$ e $TI_x \leq t \leq TF_x$[7]: nesse caso, como p_i é correto, existem dois períodos de estabilidade PE_z e PE_y, $z < y$, tal que, $p_i \in P_z$ e $p_i \in P_y$, além disso, $TF_z < t < TI_y$. Sabendo que $TI_y - TF_z < \Upsilon - \beta$ e $t > TF_z$, tem-se $TI_y - t < \Upsilon - \beta$. Dessa forma, o atraso para a entrega de m é dado por $\Sigma_{(2)} = TI_y - t + \Sigma_{(1)} < 2 * (\Upsilon - \beta) + \delta_{R1} + 3 * \Delta$.

[7]Não existe diferença se o sistema está estável ou não no tempo t. Isso acontece porque um processador correto que não faz parte de um conjunto de processadores estáveis P, no tempo t, tem a mesma visão do sistema nos dois casos.

3. existe um período de estabilidade PE_x, tal que, $p_i \in PE_x$ e $TF_x - \Delta < t \leq TF_x$: nesse caso, não é possível garantir que m será entregue por todos os processadores corretos pertencentes a P_x antes do tempo TF_x, pois esta mensagem pode levar Δ unidades de tempo para ser transmitida e $t + \Delta > TF_x$. Na realidade, até o tempo TF_x é possível que apenas p_i tenha entregue m, entretanto, como p_i é correto, existe PE_y, tal que $p_i \in P_y$. Como $t+\Delta > TF_x$, tem-se $TI_y - t < \Upsilon - \beta + \Delta$. Dessa forma, o atraso para a entrega de m é dado por $\Sigma_{(3)} = TI_y - t + \Sigma_{(1)} < 2*(\Upsilon - \beta) + \delta_{R1} + 4*\Delta$.

O maior valor entre os atrasos $\Sigma_{(1)}$, $\Sigma_{(2)}$ e $\Sigma_{(3)}$ define o atraso máximo para a entrega de uma mensagem m, pelo protocolo 1. Então, $\Sigma = 2 * (\Upsilon - \beta) + \delta_{R1} + 4 * \Delta$.

Teorema 4.7 *O protocolo 1 implementa o serviço para difusão confiável de mensagens requerido.*

Prova. As propriedades de Integridade, Acordo, Validade e Terminação do serviço para difusão confiável seguem dos lemas 4.3, 4.4, 4.5 e 4.6, respectivamente.

4.4.2 Validando o Protocolo para Detecção de Falhas

Em relação ao protocolo para detecção de falhas, deve-se garantir as propriedades de Completude e Precisão do serviço para detecção de falhas apresentado na seção 4.2.2.

Lema 4.8 (Completude) *Se um processador falhou por parada no tempo t_p, então, esta falha será detectada por todos os processadores corretos no tempo $t_p + \Omega$, onde $\Omega = \Delta + \delta_F + \delta_C$.*

Prova. Seja t_p o tempo real em que um processador p_f qualquer falhou. Como p_f não enviará mensagens $EuEstouOperante$ após ter sofrido uma falha no tempo t_p, o valor de $\delta_r(p_f)$ mantido por cada processador correto p_i será, no máximo, $t_p + \Delta$. De acordo com o protocolo 2, no pior caso, p_i irá checar se p_f falhou em $t_p + \Delta + \delta_F$ e, portanto, não irá suspeitar deste processador neste momento. Entretanto, no máximo, até $t_p + \Delta + \delta_F + \delta_C$ uma nova checagem será realizada e fará p_i suspeitar de p_f. Nesse caso, p_f será adicionado a todos os conjuntos $Psupeitos$ formados por p_i após $TI_x + \Delta + \delta_F + \delta_C$, isto é, até $t_p + \Omega$, onde $\Omega = \Delta + \delta_F + \delta_C$.

Lema 4.9 (Precisão) *Para todo instante de tempo t, existe um tempo t', $t' > t$, após o qual um conjunto majoritário de processadores corretos não irá suspeitar de algum processador correto p por um período equivalente a Θ unidades de tempo e $t' - t < \Gamma$, onde $\Theta = \beta - \Delta - \delta_E - \delta_C$ e $\Gamma = \Upsilon + \Delta + \delta_E + \delta_C$.*

Proof. Seja PE_x o primeiro período de estabilidade tal que, $t \leq TI_x$ e $p \in P_x$. Além disso, assuma $t' = TI_x + \Delta + \delta_E + \delta_C$. De acordo com o protocolo 2, no máximo, até $TI_x + \delta_E + \Delta$ todo processador correto pertencente a P_x irá receber uma mensagem $EuEstouOperante$ de p. No pior caso, um processador correto p_i irá checar se p falhou antes do tempo $TI_x + \Delta + \delta_E$, podendo considerá-lo suspeito neste momento. Entretanto, no máximo, até $TI_x + \Delta + \delta_E + \delta_C$ uma nova checagem será realizada e fará p_i retirar o processador p da sua lista de suspeitos. Então, após $TI_x + \Delta + \delta_E + \delta_C$ e, no máximo, até TF_x, todos os processadores corretos pertencentes a P_x não irão suspeitar de p. Como P_x é constituído por uma maioria de processadores corretos; $TF_x - TI_x > \beta$ e $TI_x - t < \Upsilon$, este lema é válido para $\Gamma = \beta - \Delta - \delta_E - \delta_C$ e $\Theta = \Upsilon + \Delta + \delta_E + \delta_C$.

Teorema 4.10 *O protocolo 2 implementa o serviço para detecção de falhas requerido.*

Prova. As propriedades de Completude e Precisão do serviço para detecção de falhas seguem dos lemas 4.8 e 4.9, respectivamente.

Capítulo 5

Conclusões

Neste trabalho foram propostos dois serviços de processamento tolerante a faltas para sistemas distribuídos de prateleira (assíncronos). Os serviços em questão apresentam a semântica de falha por parada e incorporam mecanismos para tolerância a faltas físicas do hardware (processador) e faltas de concepção do software (aplicação), dessa forma, podem facilitar a implementação das aplicações com requisitos de confiança no funcionamento (mais especificamente, requisitos de confiabilidade e disponibilidade) que executam nos sistemas a partir dos quais os serviços serão disponibilizados.

Cada serviço utiliza uma abordagem diferente para tolerar as faltas da aplicação, são estas: blocos de recuperação e programação em N-versões. Entretanto, ambos os serviços são providos a partir de um grupo de processadores replicados de acordo com a estratégia da replicação ativa. No caso, cada bloco de recuperação ou conjunto de N-versões que implementa uma aplicação é replicado em diferentes processadores do sistema. A maneira como os mecanismos para tolerância a faltas foram combinados confere aos serviços de processamento atributos de confiabilidade e disponibilidade.

O estudo comparativo sobre os serviços de processamento propostos permitiu uma análise simplificada dos mesmos. Através desta análise foi possível conhecer os custos associados a cada serviço, no que se refere ao grau de tolerância a faltas provido, fornecendo subsídios adicionais ao projetista da aplicação para a escolha do serviço mais adequado. O estudo foi feito em termos das probabilidades sobre o correto funcionamento e operacionalidade dos serviços.

As probabilidades sobre o correto funcionamento e operacionalidade dos serviços foram expressas pelas constantes PC e PO, respectivamente. Os valores associados a estas prob-

abilidades sofrem influência das suposições feitas acerca do comportamento dos processadores e dos mecanismos para tolerância a faltas do software, como também, do nível de redundância utilizado.

O valor de PC associado ao serviço de processamento baseado em blocos de recuperação (PC_{br}) depende da probabilidade do processador falhar apenas por parada e do teste de aceitação ser confiável. No caso do serviço de processamento baseado em programação em N-versões, esta medida depende da probabilidade do processador falhar apenas por parada e da suposição de falha associada ao conjunto de N-versões ser satisfeita, isto é válido para ambos os métodos de validação utilizados, votação majoritária ou comparação. Então, o valor de PC associado aos serviços de processamento baseados em programação em N-versões com votação majoritária (PC_{pnvV}) ou comparação (PC_{pnvC}) são equivalentes.

Por outro lado, o valor de PO para o serviço baseado em blocos de recuperação (PO_{br}) depende da probabilidade de pelo menos um processador e um dos módulos redundantes que compõem um bloco de recuperação não falharem. Já para o serviço baseado em programação em N-versões com votação majoritária, esta medida (PO_{pnvV}) depende apenas da probabilidade de pelo menos um processador não falhar. Se o método de validação for por comparação, esta medida (PO_{pnvC}) depende também da probabilidade de nenhuma das N-versões de um conjunto falharem.

Os valores de PC e PO associados a cada serviço podem ser maximizados, como apresentado a seguir:

- PC_{br}: simplificar o teste de aceitação a fim de evitar faltas de concepção e, dessa forma, aumentar a confiabilidade do teste (considere que a probabilidade do processador falhar apenas por parada é muito alta).

- PC_{pnvV}, PC_{pnvC}: aumentar o número de versões disponíveis para a aplicação.

- PO_{br}: aumentar o número de processadores disponíveis ou o número de módulos redundantes que compõem um bloco de recuperação, como também, evitar faltas físicas, diminuindo a probabilidade do processador falhar.

- PO_{pnvV}: aumentar o número de processadores disponíveis ou evitar faltas físicas, diminuindo a probabilidade do processador falhar

- PO_{pnvC}: utilizar métodos de controle de qualidade no processo de desenvolvimento do software, a fim de tornar as versões mais confiáveis (livres de faltas de concepção).

É interessante observar que os valores de PC e PO representam, respectivamente, os níveis de segurança na semântica de falha escolhida e confiabilidade/disponibilidade dos serviços. Portanto, maximizando estes valores é possível prover serviços de processamento com um maior grau de tolerância a faltas.

Os mecanismos disponibilizados através dos serviços de processamento a fim de tolerar faltas dos processadores e das aplicações, podem ser implementados no nível da linguagem de programação ou do sistema operacional. A estratégia de implementação sugerida para os mecanismos baseados nas abordagens dos blocos de recuperação e programação em N-versões apoia-se numa linguagem de programação orientada a objetos (Java) e consiste, essencialmente, em prover os referidos mecanismos através de classes e interfaces específicas. Esta estratégia também é válida para o mecanismo de replicação ativa, nesse caso, a preocupação é definir classes e interfaces que suportem a implementação de objetos (servidores da aplicação) replicados. Ainda considerando a implementação do mecanismo de replicação ativa, foi sugerida uma estratégia que descreve uma solução no nível do sistema operacional. A idéia é estender o serviço de processamento de um sistema operacional introduzindo mecanismos para criação de nodos replicados (conjunto de processadores que executam processamento replicado), e adicionar protocolos de gerência da redundância no serviço de comunicação do mesmo sistema operacional.

Para validar os serviços de processamento propostos deve-se, essencialmente, garantir a correta execução do processamento replicado através da estratégia de replicação ativa. Isto requer um protocolo para ordenação de mensagens, o qual foi projetado e validado considerando um modelo de sistema assíncrono temporizado estendido. Este modelo incorpora dois serviços complementares, são estes: um serviço para detecção de falhas (não confiável) e um serviço para difusão confiável de mensagens. Tem-se então, o uso de uma abordagem híbrida que permitiu a definição de uma solução para ordenação de mensagens simples (fácil de validar) e prática (implementável em sistemas distribuídos de prateleira).

A solução proposta para prover tolerância a faltas em sistemas distribuídos é, de fato, relevante. Isto porque, a solução foi projetada no contexto de sistemas distribuídos de prateleira e pode ser utilizada pela aplicação como um serviço. Além disso, tal solução incorpora

mecanismos para tolerância a faltas de hardware e software, sendo assim, mais genérica. Outro fator importante refere- se ao tipo de serviço considerado, no caso, serviços de processamento, isto permite que os mecanismos para tolerância a faltas sejam providos como serviços de baixo nível do sistema. É ainda interessante ressaltar o valor das discussões sobre os níveis de confiabilidade e disponibilidade que podem ser obtidos através dos serviços de processamento propostos, como também, sobre as estratégias de implementação sugeridas.

A contribuição mais importante deste trabalho refere-se à abordagem híbrida proposta. Note que, tal abordagem foi definida com o objetivo de facilitar a validação de um protocolo para ordenação de mensagens, ou seja, um contexto específico. Isto significa que o modelo de sistema assíncrono temporizado estendido proposto pode não apresentar os requisitos necessários para resolver outros problemas, porém, a idéia de usar uma abordagem híbrida no processo de validação de uma solução é válida, independente do problema considerado. De fato, esta abordagem aplica-se a qualquer protocolo distribuído projetado para sistemas assíncronos.

Bibliografia

[AVIZIENIS, 1976] Avizienis, A., 'Fault-tolerant systems.' *IEEE Transaction on Computers*, volume C-25, num. 12, pp. 1491–1501, dezembro 1976.

[AVIZIENIS, 1985] Avizienis, A., 'The n-version approach to fault tolerant software.' *IEEE Transaction on Software Engineering*, volume 11, num. 12, pp. 1491–1501, dezembro 1985.

[AVIZIENIS ET AL., 1988] Avizienis, A., Lyu, M., Schutz, W., 'In search of effective diversity: A six language study of fault tolerant flight control software.' *Proceedings of the 18th International Symposium on Fault- Tolerant Computing Systems (FTCS'18)*, Tokyo, Japan, junho 1988.

[BARATLOO ET AL., 1998] Baratloo, A., Chung, P.E., Huang, Y., Rangarajan, S., Yajnik, S., 'Filterfresh: Hot replication of java rmi server objects.' *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems (COOTS-98)*, pp. 65–78, Berkeley, USA, abril 1998.

[BERMAN ET AL., 1989] Berman, P., Garay, J. A., Perry, K. J., 'Towards optimal distributed consensus.' *Proceedings of the 30th Symposium on Foundations of Computer Science (FOCS'89)*, pp. 410–415, Washington, USA, outubro 1989.

[BIRMAN, 1985] Birman, K.P., *Replication and Fault-Tolerance in the ISIS System*. Technical Report TR85-668, Cornell University, Computer Science Department, março 1985.

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

[BRASILEIRO ET AL., 1997] Brasileiro, F. V., Gallindo, E. de L., Vasconcelos, S. R. A.,

Catão, V. S., 'On the design of the seljuk-amoeba operating environmental.' *Journal of the Brazilian Computer Society*, volume 4, num. 2, pp. 49–61, novembro 1997.

[CHANDRA ET AL., 1996a] Chandra, T., Hadzilacos, V., Toueg, S., Charron-Bost, B., 'On the impossibility of group membership.' *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC'96)*, pp. 322–330, Philadelphia, maio 1996.

[CHANDRA ET AL., 1996b] Chandra, T. D., Hadzilacos, V., Toueg, S., 'The weakest failure detector for solving consensus.' *Journal of the ACM*, volume 43, num. 4, pp. 685–722, julho 1996.

[CHANDRA & TOUEG, 1990] Chandra, T. D., Toueg, S., 'Time and message efficient reliable broadcast.' *Proceedings of the 4th International Workshop on Distributed Algorithms*, pp. 289–300, New York, USA, setembro 1990.

[CHANDRA & TOUEG, 1996] Chandra, T. D., Toueg, S., 'Unreliable failure detector for reliable distributed systems.' *Journal of the ACM*, volume 43, num. 2, pp. 225–267, março 1996.

[CHANG & MAXEMCHUK, 1984] Chang, J. M., Maxemchuk, N. F., 'Reliable broadcast protocols.' *ACM Transactions on Computer Systems*, volume 2, num. 3, pp. 251–273, agosto 1984.

[CRISTIAN, 1991] Cristian, F., 'Understanding fault-tolerant distributed systems.' *Communications of the ACM*, volume 34, num. 2, pp. 56–78, fevereiro 1991.

[CRISTIAN & FETZER, 1999] Cristian, F., Fetzer, C., 'The timed asynchronous distributed system model.' *IEEE Transactions on Parallel and Distributed Systems*, volume 10, num. 6, pp. 642–657, junho 1999.

[DOLEV ET AL., 1987] Dolev, D., Dwork, C., Stockmeyer, L., 'On the minimal synchronism needed for distributed consensus.' *Journal of the ACM*, volume 34, num. 1, pp. 77–97, janeiro 1987.

[DOLEV ET AL., 1988] Dolev, D., Lynch, N., Stockmeyer, L., 'Consensus in the presence of approximate partial synchrony.' *Journal of the ACM*, volume 35, num. 2, pp. 288–323, abril 1988.

[EZHILCHELVAN & SHRIVASTAVA, 1986] Ezhilchelvan, P. D., Shrivastava, S. K., 'A characterisation of faults in systems.' *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, pp. 215–222, Los Angeles, USA, 1986.

[FISCHER ET AL., 1985] Fischer, M. J., Lynch, N. A., Paterson, M. S., 'Impossibility of distributed consensus with one faulty process.' *Journal of the ACM*, volume 32, num. 2, pp. 374–382, abril 1985.

[GALLINDO, 1998] Gallindo, E. de L., *Processamento Confiável no Ambiente Operacional Seljuk- Amoeba*. Dissertação de mestrado, COPIN - Universidade Federal da Paraíba, Campina Grande, junho 1998.

[GERRAOUI, 1995] Gerraoui, R., 'Revisiting the relationship between non-blocking atomic commitment and consensus.' *Proceedings of the 9th International Workshop on Distributed Algorithms*, pp. 87–100, Le Mont-Saint-Michel, France, setembro 1995.

[GRAY & SIEWIOREK, 1991] Gray, J., Siewiorek, D. P., 'High-availability computer systems.' *IEEE Computer*, volume 24, num. 9, pp. 39–48, setembro 1991.

[GREVE ET AL., 1999] Greve, F., Hurfin, M., Narzul, J.L., Tronel, F., Raynal, M., *A Group Communication System to Support Fault Tolerance*. Technical Report R221-A, Institut National de Recherche en Informatique et en Automatique, março 1999.

[HUANG & KINTALA, 1993] Huang, Y., Kintala, C., 'The design and implementation of a reliable distributed operating system.' *Proceedings of the 23rd International Symposium on Fault Tolerance Computing (FTCS-23)*, pp. 2–9, Toulouse, France, junho 1993.

[HURFIN ET AL., 1998] Hurfin, M., Mostefaoui, A., Raynal, M., 'Consensus in asynchronous systems where processes can crash and recover.' *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, pp. 86–91, West Lafayette, USA, outubro 1998.

[JALOTE, 1994] Jalote, P., *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[KNIGHT & LEVESON, 1986] Knight, J. C., Leveson, N. G., 'An experimental evaluation of the assumption of independence in multiversion programming.' *IEEE Transactions on Software Engineering*, volume SE-12, num. 1, pp. 96–109, janeiro 1986.

[KNIGHT & LEVESON, 1989] Knight, J.C., Leveson, N.G.Ammann, P., *Testing Software Using Multiple Versions*. Technical Report 89029N, Software Production Consortium, junho 1989.

[LAMPORT ET AL., 1982] Lamport, L., Shostak, R., Pease, M., 'The Byzantine general problem.' *ACM Transactions on Programming Languages and Systems*, volume 4, num. 3, pp. 382–401, Jul 1982.

[LAPRIE, 1989] Laprie, J.C., 'Dependability: a unifying concept for computing and fault tolerance.' T. Anderson, editor, *Distributed Systems*, BSP Professional Books, 1989.

[LEE & ANDERSON, 1990] Lee, P. A., Anderson, T., *Fault-Tolerance: Principles and Practice - Second Edition*. Spring Verlag, 1990.

[LEMOS & VERÍSSIMO, 1991] Lemos, R. de, Veríssimo, P., 'Confiança no funcionamento - proposta para uma terminologia em português.' dezembro 1991, comunicação pessoal.

[LEVESON & SHIMEALL, 1988] Leveson, N.G., Shimeall, T.J., 'An empirical exploration of five software fault detection methods.' *IFAC SAFECOMP'88*, Fulda, Germany, novembro 1988.

[LYU, 1995] Lyu, M., *Software Fault Tolerance*. Wiley Press, 1995.

[LYU & HE, 1993] Lyu, M., He, Y., 'Improving the n-version programming process through the evolution of a design paradigm.' *IEEE Transactions on Reliability*, volume 42, num. 2, pp. 179–189, junho 1993.

[MICROSYTEMS, 1997] Microsytems, Sun, 'Remote method invocation specification.' http://java.sun.com/products/jdk/1.2/docs/guide/rmi/index.html, 1997.

[MISHRA & SCHLICHTING, 1992] Mishra, S., Schlichting, R.D., *Abstractions for Constructing Dependable Distributed Systems*. Technical Report TR92-19, University of Arizona, agosto 1992.

[MULLENDER ET AL., 1990] Mullender, S. J., Rossum, G. van, Tanenbaum, R. van R., Staven, H. van, 'Amoeba: A distributed operating system for the 1990's.' *IEEE Computer*, volume 23, num. 5, pp. 44–53, maio 1990.

[POWELL, 1992] Powell, D., *Delta-4 - A Generic Architecture for Dependable Distributed Computing*. Spring Verlag, 1992.

[RANDELL, 1975] Randell, B., 'System structure for software fault tolerance.' *IEEE Transactions on Software Engineering*, volume 1, num. 2, pp. 220–232, junho 1975.

[RANDELL & XU, 1997] Randell, B., XU, J., 'The evolution of the recorvey block concept.' http://www.cs.ncl.ac.uk/events/ anniversaries/40th/webbook/dependability /recblocks/rec_blocks.html, 1997.

[SCHNEIDER, 1990] Schneider, F. B., 'Implementing fault tolerant services using the state machine approach: a tutorial.' *ACM Computing Surveys*, volume 22, num. 4, pp. 299–319, dezembro 1990.

[SHRIVASTAVA ET AL., 1991] Shrivastava, S. K., Dixon, G. N., Parrington, G. D., 'An overview of the arjuna distributed programming system.' *IEEE Software*, volume 8, num. 1, pp. 66–73, janeiro 1991.

[VASCONCELOS, 1997] Vasconcelos, S.R.A., *Provendo Serviços para Tolerância a Faltas em Sistemas Distribuídos*. Dissertação de mestrado, COPIN - Universidade Federal da Paraíba, Campina Grande, julho 1997.

[VERISSIMO & ALMEIDA, 1995] Verissimo, P., Almeida, C., 'Quasi-synchronism: a step away from the traditional fault- tolerant real-time system models.' *Bulletin of the Technical Committee on Operating Systems and Application Environments (TCOS)*, volume 7, num. 4, pp. 35–39, dezembro 1995.

Apêndice A

O Problema do Consenso em Sistemas Assíncronos

A.1 Introdução

O problema do consenso não possui solução determinística em sistemas assíncronos, nem mesmo quando apenas um único processador falha por parada. Isto é conhecido como o resultado de impossibilidade FLP [FISCHER ET AL., 1985]. Este resultado não se aplica aos sistemas síncronos. Porém, a maioria dos sistemas existentes são, de fato, assíncronos, porque tais sistemas estão sujeitos a determinadas situações (sobrecarga de processamento, por exemplo) que provocam a perda de sincronismo, impedindo que as restrições especificadas para o modelo de sistema síncrono sejam garantidas da forma esperada[1].

Considerando a aplicabilidade do consenso, principalmente no sentido de resolver problemas de acordo, e sabendo da importância de desenvolver soluções para o modelo de sistema assíncrono, neste Apêndice apresenta-se um estudo sobre o problema do consenso em sistemas assíncronos. Como parte deste estudo, tem-se a descrição do modelo assíncrono e explicação do resultado de impossibilidade FLP, além disso, discute-se a abordagem dos detectores de falhas não confiáveis cujo objetivo é impor as restrições ao modelo assíncrono, possibilitando soluções para o problema do consenso. Finalmente, são descritos um protocolo de consenso para o modelo assíncrono com detectores de falhas não confiáveis e um protocolo de difusão atômica baseado em consenso, mais especificamente, que utiliza o protocolo de consenso descrito.

[1] O modelo de sistema síncrono caracteriza-se pela certeza na comunicação. Isto significa que, os atrasos na transmissão de mensagens e escalonamento de tarefas devem ser limitados e conhecidos em sistemas baseados neste modelo.

A.2 O Modelo de Sistema Assíncrono

O modelo de sistema assíncrono, definido formalmente em [FISCHER ET AL., 1985], não incorpora nenhuma noção de tempo, por essa razão, é visto como um modelo livre de qualquer restrição de sincronismo.

Um sistema assíncrono não requer o uso de relógios, pois, os serviços providos por este sistema não são temporizados, ou seja, a especificação do serviço descreve a saída produzida como resultado da computação realizada sobre cada entrada de dados (o quê) e a seqüência de passos executados a fim de obter tais resultados (como), mas, não estabelece o tempo gasto para que cada resultado seja produzido (quando). Da mesma forma, não existe restrições quanto aos atrasos na transmissão de mensagens e escalonamento de tarefas. Em relação à semântica de falha dos componentes do sistema, os processos e processadores podem falhar, apenas, por parada e assume-se que a comunicação entre os processadores é confiável, isto é, o serviço de comunicação oferecido não perde, como também, não duplica ou falsifica mensagens.

Por outro lado, num sistema assíncrono não é possível determinar se um processador falhou ou, apenas, está muito lento. Este fato impede que alguns problemas sejam resolvidos com soluções determinísticas sobre os sistemas em questão, dentre os quais: problemas de consenso, eleição de líder e gerenciamento de grupo [FISCHER ET AL., 1985], [CHANDRA ET AL., 1996a].

Considerando o problema do consenso, observa-se que é impossível resolver tal problema em sistemas assíncronos, até mesmo quando um único processador falha por parada. Este resultado foi comprovado por Fischer, Lynch e Paterson, sendo conhecido como o resultado de impossibilidade FLP [FISCHER ET AL., 1985].

A.2.1 O Resultado de Impossibilidade FLP

No problema do consenso, cada processador propõe um valor aos outros e todos os processadores corretos decidem um valor comum que será um dos valores propostos. Formalmente, o consenso é definido em termos de algumas propriedades, as quais devem ser garantidas pelas primitivas que implementam a solução para tal problema. Estas propriedades são as seguintes [CHANDRA & TOUEG, 1996]:

Terminação. Todos os processadores corretos, em algum momento, decidem um valor para o consenso.

Validade. Se um processador decidir o valor x, então, este valor foi proposto por algum processador.

Acordo. Quaisquer dois processadores corretos nunca decidem valores diferentes.

O resultado de impossibilidade FLP para o problema do consenso pode ser explicado, informalmente, através do exemplo apresentado a seguir [GREVE ET AL., 1999].

Suponha um sistema assíncrono composto por dois processadores P_1 e P_2 com valores iniciais v_1 e v_2, respectivamente. Para obter consenso nesse sistema, os processadores precisam trocar, pelo menos, uma mensagem, o que pode ser feito de duas maneiras: P_1 envia v_1 para P_2 ou P_2 envia v_2 para P_1. Em ambos os casos é possível observar o resultado de impossibilidade para o consenso, então, assume-se a segundo caso e considera-se que apenas P_2 pode falhar. O processador P_1 não pode tomar nenhuma decisão enquanto não receber o valor v_2 proveniente de P_2, caso contrário, a propriedade de acordo será violada. Sendo assim, P_1 pode esperar um longo tempo pelo valor v_2, isto acontece por duas razões igualmente possíveis:

1. P_2 falhou antes de enviar v_2 ou

2. P_2 está lento, mas, permanece funcionando corretamente.

Devido ao assincronismo característico do sistema, P_1 não pode determinar qual das razões, 1 ou 2, justifica o fato ocorrido. Seja qual for a escolha de P_1, confiar em P_2 e esperar pelo valor v_2 para tomar qualquer decisão (2) ou desconfiar de P_2 e decidir sobre o valor v_1 (1), esta pode ser errada, dependendo do que, realmente, causou o atraso no recebimento do valor v_1. Portanto, é impossível resolver o problema do consenso, com soluções determinísticas, sobre o modelo de sistema assíncrono, sem definir suposições extras para este modelo.

Motivados pelo fato de muitos sistemas reais serem assíncronos e sabendo dos resultados de impossibilidade para o consenso sobre este modelo de sistema, muitos pesquisadores têm trabalhado no sentido de descobrir um conjunto mínimo de propriedades que, quando satisfeitas por um sistema assíncrono, possibilite a resolução do problema do consenso com uma

solução determinística. Dentre os resultados já obtidos, pode-se citar: mínimo sincronismo [DOLEV ET AL., 1987], sincronismo parcial [DOLEV ET AL., 1988] e detectores de falhas não confiáveis [CHANDRA & TOUEG, 1996].

A abordagem dos detectores de falhas não confiáveis é considerada bastante atraente, o que tem levado à definição de várias classes de detectores e protocolos de consenso específicos para cada classe. Os principais aspectos relacionados a esta abordagem serão discutidos na próxima seção.

A.3 O Modelo de Sistema Assíncrono com Detectores de Falhas Não Confiáveis

A abordagem dos detectores de falhas não confiáveis foi proposta por Chandra e Toueg e apresentada em [CHANDRA & TOUEG, 1996]. A idéia dos autores era estender o modelo de sistema assíncrono com um mecanismo para detecção de falhas, passível de erros, mas, suficientemente confiável no sentido de fornecer informação sobre falhas ocorridas no sistema, permitindo resolver o problema do consenso.

Um detector de falhas pode ser visto como um conjunto de módulos detectores de falhas, um por processador. Os módulos associados a cada processador mantêm uma lista de processadores considerados suspeitos de terem sofrido uma falha por parada. Estes módulos podem cometer erros, seja suspeitando dos processadores corretos ou não suspeitando dos processadores que, realmente, falharam. Isto acontece porque, num sistema assíncrono não é possível assegurar se um processador falhou por parada ou está lento, mas permanece funcionando corretamente. Embora um módulo detector de falhas possa adicionar processadores corretos a sua lista de suspeitos, estes também podem ser removidos posteriormente, se o módulo reconhecer o erro cometido. Então, cada módulo pode adicionar e remover processadores de sua lista de suspeitos, repetidas vezes. Além disso, módulos associados a processadores diferentes podem ter listas de suspeitos distintas.

Chandra e Toueg introduziram várias classes de detectores de falhas. Cada classe é definida em função de duas propriedades abstratas[2]: uma propriedade de completude (*completeness*) e uma propriedade de precisão (*accuracy*). Estas propriedades garantem, respectiva-

[2]Uma propriedade abstrata não depende de nenhuma implementação particular, seja um mecanismo em hardware ou software.

mente, a vivacidade e a segurança dos detectores de falhas. As propriedades de completude estabelecem que um detector de falhas, a partir de um certo momento, deve suspeitar de todos os processadores que falharam por parada. Sendo assim, esta propriedade assegura a detecção das falhas reais. Já as propriedades de precisão estabelecem que, se o detector de falhas suspeita de algum processador, este processador realmente falhou. Isto permite restringir o número de erros cometidos pelo detector de falhas.

As classes de detectores de falhas introduzidas por Chandra e Toueg, diferem em relação ao nível das restrições impostas pelas suas respectivas propriedades de completude e precisão. Ou seja, a diferença está na quantidade de informação sobre falhas nos processadores do sistema que os detectores de cada classe podem prover. Nesse caso, a classe de detectores de falhas menos restritiva é identificada como $\Diamond W$ e apresenta as seguintes propriedades [CHANDRA & TOUEG, 1996]:

Completude. Existe um tempo, a partir do qual, alguns processadores corretos irão sempre suspeitar dos processadores que falharam por parada.

Precisão. Existe um tempo, a partir do qual, todos os processadores corretos não irão suspeitar de algum processador correto.

Qualquer detector de falhas A que satisfaça as propriedades da classe $\Diamond W$, provê informação suficiente sobre falhas para resolver o problema do consenso. De fato, os detectores desta classe são os menos restritivos que podem ser utilizados em protocolos de consenso projetados sobre sistemas assíncronos, como é demonstrado em [CHANDRA ET AL., 1996b]. Mais precisamente, se um detector A' pode ser utilizado para resolver o problema do consenso, então, existe um algoritmo que transforma A' em A [3].

A.4 O Protocolo de Consenso CT

Na literatura podem ser encontrados diversos protocolos de consenso baseados em detectores de falhas não confiáveis. A maioria destes protocolos utilizam detectores da classe $\Diamond S$ [4] cujas

[3] Um algoritmo que transforma um detector de falhas D' em outro detector de falhas D é chamado algoritmo de redução. Alguns exemplos de algoritmos deste tipo são apresentados em [CHANDRA & TOUEG, 1996] e [CHANDRA ET AL., 1996b].

[4] As classes $\Diamond W$ e $\Diamond S$, apesar de apresentarem propriedades de completude diferentes, são equivalentes. De fato, existe um algoritmo de redução que transforma qualquer detector da classe $\Diamond S$ em outro da classe $\Diamond W$ [CHANDRA & TOUEG, 1996]. Sendo assim, os detectores de falhas da classe $\Diamond S$ são tão restritivos quanto aqueles da classe $\Diamond W$.

propriedades são descritas abaixo:

Completude. Existe um tempo, a partir do qual, todos os processadores corretos irão sempre suspeitar dos processadores que falharam por parada.

Precisão. Existe um tempo, a partir do qual, todos os processadores corretos não irão suspeitar de algum processador correto.

O primeiro protocolo de consenso baseado em detectores de falhas da classe $\Diamond S$ foi definido por Chandra e Toueg [CHANDRA & TOUEG, 1996]. De acordo com este protocolo, pelo menos, a maioria dos processadores envolvidos no consenso devem estar corretos, ou seja, tem-se $f < n/2$, onde n é o número de processadores total do sistema e f é o número de processadores que podem falhar.

O protocolo de consenso CT utiliza o paradigma do coordenador rotativo (*rotating coordinator paradigm*) [CHANDRA & TOUEG, 1990], [BERMAN ET AL., 1989], e executa em rodadas assíncronas consecutivas, onde cada rodada é gerenciada por um processador previamente escolhido (coordenador). Dessa forma, todos os processadores sabem que o coordenador de uma rodada r, será o processador $c = \lfloor (r \mod n) + 1 \rfloor$. As primitivas de acesso ao protocolo são $propose(-)$ e $decide(-)$, através das quais um novo consenso é iniciado e finalizado, respectivamente.

De acordo com o referido protocolo, sempre que um processador tornar-se o coordenador da rodada, ele tentará resolver o consenso, propondo um valor consistente para apreciação dos outros processadores. Se o coordenador estiver correto e não for considerado suspeito por nenhum processador, será obtido consenso sobre o valor proposto pelo mesmo. Este valor é determinado a partir dos valores propostos pelos processadores envolvidos no consenso. É interessante ressaltar que cada processador gerencia, dentre outras, duas variáveis locais, quais sejam: $estimate$ e ts cujo conteúdo corresponde, respectivamente, ao último valor de consenso aceito pelo processador e ao identificador (r) da rodada em que o referido valor foi aceito. Estas variáveis são atualizadas à medida que o protocolo de consenso progride na execução e converge para a decisão final do consenso. Cada rodada do protocolo é dividida em quatro fases, a saber:

Fase 1. Na primeira fase, cada processador envia para o coordenador a sua proposta de consenso, ou seja, uma mensagem contendo o valor da variável $estimate$. Esta mensagem

recebe como identificador o valor da variável ts.

Fase 2. Na segunda fase, o coordenador reúne as mensagens (propostas de consenso) recebidas de, pelo menos, $\lfloor (n+1)/2 \rfloor$ processadores e seleciona aquela com maior identificador. Esta mensagem será difundida para todos os outros processadores como a proposta de consenso do coordenador.

Fase 3. Na terceira fase, cada processador espera pelo recebimento da proposta de consenso do coordenador, podendo acontecer uma das seguintes situações: 1) o processador recebe a proposta de consenso do coordenador e envia para o mesmo uma mensagem de reconhecimento positivo, indicando que a proposta recebida será adotada como o último valor de consenso aceito pelo processador (atualiza as variáveis $estimate$ e ts); ou 2) o processador suspeita que o coordenador falhou por parada antes de receber sua respectiva proposta de consenso, nesse caso, o processador envia para o coordenador uma mensagem de reconhecimento negativo. Ao final desta fase, ocorrendo (1) ou (2), os processadores passam para a próxima rodada do protocolo.

Fase 4. Na quarta fase, o coordenador aguarda o recebimento de, pelo menos, $\lfloor (n+1)/2) \rfloor$ mensagens de reconhecimento (positivo ao negativo), previamente enviadas pelos processadores. Se todas as mensagens recebidas forem de reconhecimento positivo, então, o coordenador assume que o valor proposto pelo mesmo é consistente e deve ser tomado como a decisão final do consenso. Para tal, o coordenador encapsula este valor numa mensagem e faz a difusão confiável (*reliable broadcast*) da mesma, a fim de que todos os processadores decidam sobre o valor em questão. Por outro lado, se o coordenador receber alguma mensagem de reconhecimento negativo, considera-se que a rodada corrente falhou, nesse caso, o coordenador iniciará uma nova rodada do protocolo de consenso.

O protocolo CT utiliza um esquema centralizado, onde, todas as mensagens trocadas entre os processadores durante uma rodada são provenientes ou destinadas ao processador coordenador. O número de mensagens trocadas durante a execução do protocolo vai depender do número de ocorrências reais (ou suspeitas) de falhas. Para que seja obtido consenso na primeira rodada do protocolo (melhor caso), o coordenador desta rodada não deve falhar ou

ser considerado suspeito por nenhum outro processador, nesse caso, o número de mensagens trocadas será $\lfloor 3 * (n-1) \rfloor$, além da mensagem contendo o valor final do consenso, difundida pelo coordenador. De fato, não é possível estimar o número de mensagens necessárias para obter consenso utilizando o protocolo CT, pois é imprevisível o tempo a partir do qual um processador correto não será mais considerado suspeito. Além disso, é impossível estimar o tempo de execução do protocolo pois este executa em rodadas assíncronas, isto é, não se sabe em qual instante uma rodada será iniciada ou quanto tempo a mesma vai durar.

A.4.1 Utilizando o Protocolo de Consenso CT para Resolver o Problema da Difusão Atômica de Mensagens

O consenso é tido como uma abstração para problemas de acordo, tal como o problema da difusão atômica de mensagens. Nesse caso, os processadores devem chegar a uma decisão comum sobre a ordem na qual um conjunto de mensagens será entregue. De modo geral, considera-se o consenso como sendo um bloco de construção (*building block*) sobre o qual soluções para problemas de acordo podem ser projetadas [HURFIN ET AL., 1998].

Chandra e Toueg propuseram e validaram uma solução baseada em consenso para difusão atômica de mensagens que está descrita em [CHANDRA & TOUEG, 1996]. Esta solução foi definida sobre o modelo assíncrono com detectores de falhas não confiáveis e utiliza o protocolo de consenso CT. Além disso é requerido um serviço para difusão confiável de mensagens, implementado através do protocolo apresentado no artigo referenciado.

O protocolo que descreve a solução de Chandra e Toueg para difusão atômica de mensagens, utiliza repetidas execuções de consenso e estas podem acontecer de forma concorrente. Nesse caso, para evitar ambigüidades sobre o valor decidido em cada consenso, os mesmos são identificados com um contador k. Sendo assim, a k-ésima execução do consenso deve decidir o k-ésimo conjunto de mensagens que será entregue naquele momento. O serviço para difusão confiável de mensagens será utilizado com a finalidade de garantir que todos os processadores corretos envolvidos no procedimento de difusão atômica, manipulam o mesmo conjunto de mensagens.

As primitivas do protocolo de consenso são $propose(k, -)$ e $decide(k, -)$. Note que, os parâmetros de entrada destas primitivas diferem daqueles definidos para as primitivas do protocolo de consenso CT. A diferença está apenas no valor do contador k que representa

o identificador do consenso sendo iniciado e finalizado. Já as primitivas do protocolo de difusão atômica são $A\_broadcast(-)$ e $A\_deliver(-)$, responsáveis, respectivamente, por entregar e difundir as mensagens submetidas ao protocolo em questão.

A difusão atômica de uma mensagem m, $A\_broadcast(m)$, na verdade, é feita através da primitiva correspondente, definida para o protocolo de difusão confiável utilizado. Por outro lado, a entrega de um conjunto de mensagens difundidas M, $A\_deliver(M)$, ocorre da seguinte forma. Quando as mensagens difundidas chegam ao seu destinatário, estas são entregues através do protocolo de difusão confiável e armazenadas num determinado conjunto. Havendo mensagens neste conjunto ainda não entregues pelo protocolo de difusão atômica, inicia-se uma nova execução do protocolo de consenso, passando como parâmetros o valor do contador k e as referidas mensagens. Estas mensagens constituem o valor proposto pelo processador que está iniciando o consenso. Ao término da execução de cada consenso, o conjunto de mensagens retornado como decisão final será entregue atomicamente numa ordem pré definida.

O protocolo de difusão atômica em questão satisfaz as seguintes propriedades (como comprovado em [CHANDRA & TOUEG, 1996]):

Acordo. Se um processador correto entrega uma mensagem m, então, todos os processadores corretos, em algum momento, também entregam esta mensagem.

Validade. Se um processador correto difunde uma mensagem m, então, todos os processadores corretos, em algum momento, entregam esta mensagem.

Integridade. Para qualquer mensagem m, todo processador entrega m uma única vez e, apenas, se esta mensagem foi difundida por algum processador.

Ordem. Se um processador correto entrega duas mensagens m e m', nesta ordem, então, todos os processadores corretos entregam estas mensagens na mesma ordem.