**UNIVERSIDADE FEDERAL DE SANTA CATARINA**
**PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**Jeferson Luiz Rodrigues Souza**

# WSFTA: Uma Arquitetura para Tolerância a Faltas em Serviços Web

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

Prof. Frank Augusto Siqueira, Dr.
(Orientador)

Florianópolis, Março de 2008

# WSFTA: Uma Arquitetura para Tolerância a Faltas em Serviços Web

Jeferson Luiz Rodrigues Souza

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Sistemas Distribuídos, e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

________________________________________

Prof. Mário Antônio Ribeiro Dantas, Dr. (Coordenador)

Banca Examinadora

________________________________________

Prof. Frank Augusto Siqueira, Dr.

(Orientador)

________________________________________

Prof. Mário Antônio Ribeiro Dantas, Dr.

________________________________________

Prof. Lau Cheuk Lung, Dr.

________________________________________

Prof. Joni da Silva Fraga, Dr.

*"Nada lhe posso dar, nada mais do que já não exista em você mesmo. Não posso abrir-lhe outro mundo de imagens, além daquele que há em sua própria alma. Nada lhe posso dar, a não ser a oportunidade, o impulso, a chave para busca do trilhar de seus próprios caminhos. Eu o ajudarei a tornar visível o seu próprio mundo, e isso é tudo."*

*Hermann Hesse*

A Deus, minha namorada Denise e a todos os membros da
minha família que acreditaram nesta caminhada.

# Agradecimentos

A todos os membros da minha família, principalmente aos meus tios Paulo, Lígia e Maria de Fátima, minhas irmãs Luciana e Bianca e minha mãe Regina, por terem me fornecido uma base sólida para alcançar mais este estágio de conhecimento.

A Denise, minha namorada, por ter se tornado em tão pouco tempo o meu grande amor, sendo extremamente compreensiva nos diversos momentos dos quais estive ausente.

Aos amigos Vivian, Lucas (Poodle), Joelson (Tio), Sandro (Sandroca) e aos amigos do LaPeSD Alexandre Parra (Parra de Uva), Vinícius (Ligeirinho) e Christopher, pelas boas risadas, momentos de descontração e trocas de conhecimento, situações essas que foram de grande importância na melhoria contínua e visualização de novos horizontes dentro do contexto desta dissertação de mestrado.

Aos eternos amigos da época da graduação Gustavo (Camisoleta), Luiz, Osvaldo (Serginho), João Paulo (Jonnhy Codorna), Eder (Malvado), Igor (Madimbu), Lucas, Marina, Márcio (Pink), Max (Cérebro) e Anderson pelo incentivo, convivência e principalmente pelo comparecimento nas confraternizações repletas de histórias das situações cômicas do passado.

Ao meu orientador, Frank Siqueira, por estar sempre presente e disposto a contribuir de todas as formas, tendo se tornado um grande amigo ao longo desta caminhada. Ao professor, Mário Dantas, por compartilhar seu conhecimento e seu senso de humor inigualáveis.

Por fim, agradeço a todos os professores que contribuiram direta ou indiretamente para a elaboração deste trabalho.

# Sumário

**Sumário** vi

**Lista de Figuras** ix

**Lista de Tabelas** xi

**Lista de Algoritmos** xii

**Lista de Acrônimos** xiii

**Resumo** xiv

**Abstract** xv

**1 Introdução** 1

1.1 Objetivo do Trabalho . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.2 Justificativa . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.3 Metodologia . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.4 Organização do texto . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**2 Confiança no Funcionamento** 5

2.1 Conceituação Básica . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2 Falta, Erro e Falha . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.3 Meios para Obtenção de Confiança no Funcionamento . . . . . . . . . . . . . 8

2.4 Tolerância a Faltas em Sistemas Distribuídos . . . . . . . . . . . . . . . 8

2.4.1 Comunicação de Grupo . . . . . . . . . . . . . . . . . . . . . . . . 12

2.4.2 Replicação . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.4.3 Detecção de Falhas . . . . . . . . . . . . . . . . . . . . . . . . . . 19

2.4.4 Recuperação de Falhas . . . . . . . . . . 22
2.5 Considerações sobre o capítulo . . . . . . . . . . 23

**3 Serviços Web** **24**
3.1 Arquitetura de Serviços Web . . . . . . . . . . 25
3.2 Camada de Transporte . . . . . . . . . . 27
3.3 Camada de Troca de Mensagens (SOAP) . . . . . . . . . . 27
3.4 Camada de Descrição (WSDL) . . . . . . . . . . 29
3.5 Camada de Descoberta (UDDI) . . . . . . . . . . 31
3.6 WS-Reliable Messaging . . . . . . . . . . 31
3.7 Tolerância a Faltas em Serviços Web . . . . . . . . . . 33
3.8 Considerações sobre o capítulo . . . . . . . . . . 34

**4 Trabalhos Relacionados** **35**
4.1 A Middleware for Replicated Web Services . . . . . . . . . . 36
4.2 A Framework to Support Survivable Web Services . . . . . . . . . . 37
4.3 FTWEB: A Fault Tolerant Infrastructure for Web Services . . . . . . . . . . 39
4.4 Fault Tolerant Web Services . . . . . . . . . . 41
4.5 WS-Replication: A Framework for Highly Available Web Services . . . . . . . . . . 43
4.6 Análise Comparativa . . . . . . . . . . 46
4.7 Considerações sobre o capítulo . . . . . . . . . . 47

**5 WSFTA - Uma Arquitetura para Tolerância a Faltas em Serviços Web** **49**
5.1 Descrição Geral da Arquitetura . . . . . . . . . . 50
5.2 Componente de Comunicação de Grupo - CG . . . . . . . . . . 55
5.3 Componente de Ordenação de Mensagens - MO . . . . . . . . . . 57
5.4 Detecção de Falha . . . . . . . . . . 58
5.4.1 Componente de Detecção de Falha de Réplica - RFD . . . . . . . . . . 59
5.4.2 Componente de Detecção de Falha de Grupo - GFD . . . . . . . . . . 60
5.5 Componentes de LOG (LOG) e Recuperação de Falhas (REC) . . . . . . . . . . 62
5.6 Manutenção e armazenamento de estado . . . . . . . . . . 63
5.7 Considerações sobre o capítulo . . . . . . . . . . 65

**6 Implementação, Experimentos e Análise de Resultados 67**

6.1 Comunicação de Grupo . . . . . . . . . . 68
6.2 Ordenação de Mensagens . . . . . . . . . . 72
6.3 Detecção de Falha . . . . . . . . . . 75
6.4 Armazenamento e recuperação de LOG . . . . . . . . . . 79
6.5 Recuperação de Falha . . . . . . . . . . 80
6.6 Experimentos . . . . . . . . . . 81
6.7 Análise dos Resultados . . . . . . . . . . 82
6.8 Comparação com Trabalhos Relacionados . . . . . . . . . . 87
6.9 Considerações sobre o capitulo . . . . . . . . . . 90

**7 Conclusões e Trabalhos Futuros 93**

7.1 Trabalhos Futuros . . . . . . . . . . 94
7.2 Considerações Finais . . . . . . . . . . 95

**Referências Bibliográficas 96**

**A Descrição em CSP dos componentes da WSFTA 101**

A.1 Componente de Proxy - WSFTA-Proxy . . . . . . . . . . 101
A.2 Componente de Comunicação de Grupo - CG . . . . . . . . . . 102
A.3 Componente de Ordenação de Mensagens - MO . . . . . . . . . . 104
A.4 Processo de detecção de falhas . . . . . . . . . . 106
A.5 Componente de LOG - LOG . . . . . . . . . . 108
A.6 Componente de Recuperação de Falhas - REC . . . . . . . . . . 109

**B Classes da WSFTA 112**

# Lista de Figuras


2.1 Árvore de conceitos da confiança no funcionamento [Avizienis et al., 2004] . . . . 6
2.2 Relação entre falta, erro e falha no modelo de 3 universos . . . . . . . . . . . . . . . 7
2.3 Representação conceitual da propagação de faltas em um SD . . . . . . . . . . . . 10
2.4 Classes de Faltas [Veríssimo and Rodrigues, 2001] . . . . . . . . . . . . . . . . . . . 11
2.5 Difusão FIFO [Veríssimo and Rodrigues, 2001] . . . . . . . . . . . . . . . . . . . . . . 14
2.6 Difusão Causal [Veríssimo and Rodrigues, 2001] . . . . . . . . . . . . . . . . . . . . . 14
2.7 Difusão Atômica [Veríssimo and Rodrigues, 2001] . . . . . . . . . . . . . . . . . . . . 16
2.8 Esquema de Funcionamento da Replicação Ativa . . . . . . . . . . . . . . . . . . . . 17
2.9 Esquema de Funcionamento da Replicação Passiva . . . . . . . . . . . . . . . . . . . 19
2.10 Esquema de Funcionamento da Replicação Semi-ativa . . . . . . . . . . . . . . . . 20

3.1 Modelo Conceitual da SOA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
3.2 Pilha de protocolos utilizados em Serviços Web . . . . . . . . . . . . . . . . . . . . . 27
3.3 Representação dos elementos de uma mensagem SOAP . . . . . . . . . . . . . . . 28
3.4 Organização de um documento WSDL . . . . . . . . . . . . . . . . . . . . . . . . . . 30
3.5 Contratos de entrega confiável de mensagens . . . . . . . . . . . . . . . . . . . . . . 32

4.1 Representação de uma réplica contendo o PWSS e o WSS [Ye and Shen, 2005] . . 37
4.2 Modelo de Sistema do SWS *Framework* [Li et al., 2005] . . . . . . . . . . . . . . . 38
4.3 Infra-estrutura FTWEB [Santos et al., 2005] . . . . . . . . . . . . . . . . . . . . . . . 40
4.4 Grupo de serviços com o FT-SOAP [Fang et al., 2007] . . . . . . . . . . . . . . . . . 42
4.5 WS-Replication *Framework* [Salas et al., 2006] . . . . . . . . . . . . . . . . . . . . . 44

5.1 Representação dos componentes da WSFTA . . . . . . . . . . . . . . . . . . . . . . . 51
5.2 Diferentes formações suportadas pela WSFTA . . . . . . . . . . . . . . . . . . . . . 53
5.3 Réplicas organizadas em dois grupos . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

5.4 Abstração oferecida pelo componente GC . . . . . . . . . . . . . . . . . . . . . . . . 56
5.5 Funcionamento conceitual do componente MO com réplicas distribuídas em dois grupos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.6 RFDs formando um anel virtual de monitoramento em grupo com líder . . . . . . . 59
5.7 Anel virtual de detecção de falha dos GFDs . . . . . . . . . . . . . . . . . . . . . . . 61
5.8 Serviço A invocando funcionalidades de outros serviços . . . . . . . . . . . . . . . 64

6.1 Principais classes do componente GC . . . . . . . . . . . . . . . . . . . . . . . . . 68
6.2 Principais classes do componente MO . . . . . . . . . . . . . . . . . . . . . . . . . 72
6.3 Principais classes dos componentes RFD e GFD . . . . . . . . . . . . . . . . . . . . 76
6.4 Diagrama de eventos entre Detector e Serviço . . . . . . . . . . . . . . . . . . . . . 78
6.5 Principais classes do componente LOG . . . . . . . . . . . . . . . . . . . . . . . . 79
6.6 Principais classes do componente REC . . . . . . . . . . . . . . . . . . . . . . . . . 80
6.7 Overhead da replicação semi-ativa quanto ao tamanho das mensagens . . . . . . . 83
6.8 Overhead da replicação semi-ativa quanto ao número de usuários simultâneos . . . 84
6.9 Vazão do serviço com a WSFTA utilizando replicação semi-ativa em diferentes configurações . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
6.10 Comparação do overhead do serviço utilizando a WSFTA com replicação passiva e semi-ativa . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
6.11 Variação do tempo de resposta em função da variação dinâmica do número de usuários simultâneos . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86

B.1 Pacotes com as classes de recuperação e log . . . . . . . . . . . . . . . . . . . . . . 112
B.2 Pacotes com as classes para comunicação de grupo e detecção de falha . . . . . . . 113
B.3 Pacotes com as classes de ordenação, notificação de eventos e *handlers* . . . . . . 114

# Lista de Tabelas

4.1 Comparativo entre as diferentes infra-estruturas e *frameworks* de TF para Serviços Web . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

6.1 Comparativo entre as diferentes infra-estruturas e *frameworks* de TF para Serviços Web . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89

# Lista de Algoritmos


6.1 Enviando mensagem para um grupo de réplicas . . . . . . . . . . . . . . . . . . . . 69

6.2 Recebendo mensagens de outras réplicas do grupo . . . . . . . . . . . . . . . . . . 70

6.3 Verifica e devolve resposta(s) de mensagem(ns) enviada(s) anteriormente . . . . . 71

6.4 Adicionando mensagem para ordenação . . . . . . . . . . . . . . . . . . . . . . . 91

6.5 Define ordem de execução das mensagens . . . . . . . . . . . . . . . . . . . . . . 92

# Lista de Acrônimos

**CORBA** *Common Object Request Broker Architecture*

**CSP** *Communicating Sequential Processes*

**DF** *Detector de Falhas*

**FIFO** *Fisrt-In-First-Out*

**GCM** *Group Communication Manager*

**LAN** *Local Area Network*

**MH** *Message Handler*

**PWSS** *Proxy Web Service Site*

**SD** *Sistema Distribuído*

**SOA** *Service Oriented Architecture*

**SWS-IGC** *SWS Inter-Group Communication Protocol*

**SWS-MO** *SWS Message Ordering Protocol*

**TF** *Tolerância a Faltas*

**UDDI** *Universal Description, Discovery and Integration*

**WAN** *Wide Area Network*

**WSDL** *Web Services Description Language*

**WSFTA** *Web Services Fault Tolerance Architecture*

**WSS** *Web Service Site*

**XML** *eXtensible Markup Language*

# Resumo

A grande vantagem na utilização de Serviços Web para concepção de sistemas distribuídos é a interoperabilidade que essa tecnologia proporciona [Ayala et al., 2002]. Essa interoperabilidade facilita de forma significativa o desenvolvimento de aplicações distribuídas, já que problemas tais como a diversidade de hardwares e softwares são solucionados pelos protocolos utilizados por esses serviços [Newcomer, 2002]. Nesse sentido, Serviços Web têm sido amplamente utilizados para solucionar problemas de interoperabilidade entre aplicações e/ou tecnologias. Porém, as especificações e padrões definidos para Serviços Web não solucionam problemas relativos a tolerância a faltas dos serviços.

A forma mais tradicional para tolerar faltas é a utilização de esquemas de replicação, provendo continuidade e disponibilidade aos serviços [Veríssimo and Rodrigues, 2001, Li et al., 2005]. Baseado nessa premissa, esse trabalho apresenta uma proposta de arquitetura de *software* para tolerância a faltas em Serviços Web. Essa arquitetura, chamada de WSFTA (*Web Services Fault Tolerant Architecture*), descreve de forma abstrata seus componentes e princípio de funcionamento, dividindo as responsabilidades e funcionalidades, presentes na arquitetura, entre esses componentes. Adicionalmente, este trabalho apresenta a especificação formal desses componentes, que possibilita a utilização de diferentes técnicas para comunicação de grupo, ordenação de mensagens, detecção e recuperação de falhas, proporcionando a flexibilidade necessária para que a arquitetura possa ser configurada para tratar as diversas classes de faltas presentes na literatura. Além disso, esse trabalho ainda apresenta uma implementação de referência e experimentos que possibilitam a comprovação de sua potencialidade de utilização no fornecimento de tolerância a faltas para Serviços Web.

**Palavras chave:** Tolerância a faltas, Serviços Web, Replicação.

# Abstract

The advantage in use Web Services for design of distributed systems is the interoperability provided for this technology [Ayala et al., 2002]. This interoperability facilitates the development of distributed applications, already that problems such as diversity of hardware and software are solved by the protocols used by these services [Newcomer, 2002]. In this way, Web Services have been widely used to solve problems of interoperability among applications and/or technologies. However, the specifications and standards defined for Web Services do not address solutions for problems related to fault tolerance of services.

The replication schemes are the most traditional way to tolerate faults, providing continuity and availability to the services [Veríssimo and Rodrigues, 2001, Li et al., 2005]. Based on this sentence, this work proposes a software architecture for fault tolerance in Web Services. This software architecture, called WSFTA (Web Services Fault Tolerant Architecture), describes your components an abstract form, dividing responsibilities and features, presents in this architecture, between these components. Additionally, this work present the formal specification of these components, allowing the use different techniques of group communication, message order, failure recovery and detection, provide the capability of configuration this architecture for support the different classes of faults presents in the literature. Finally, this work presents an reference implementation and experiments that enable the proof of its potential for use in the provision of fault tolerance for Web Services.

**Keywords:** Fault Tolerance, Web Services, Replication.

# Capítulo 1

# Introdução

Desde o surgimento dos sistemas distribuídos, softwares passaram a utilizar a infra-estrutura de rede para trocar informações entre si, formando um conjunto integrado de sistemas. Analogamente, pode-se dizer que um conjunto de pessoas, agrupadas em uma equipe que tem a responsabilidade de solucionar um determinado problema, classificasse como um sistema distribuído. Nessa equipe, cada pessoa é responsável por um conjunto de tarefas, sendo que essas tarefas podem ser executadas paralelamente. Qualquer problema que atinja uma dessas pessoas, pode ou não, atrapalhar a execução de suas tarefas. Em outras palavras, a origem desse problema (falta) pode ocasionar erros durante a execução das tarefas, sendo que a manifestação desses erros pode levar uma pessoa a não concluir suas atividades (falha), comprometendo assim toda a equipe, caso não existam outras pessoas que consigam executar as mesmas tarefas no lugar da pessoa que falhou. É assim, similar à situação descrita na analogia anterior, o contexto de aplicação de técnicas de uma área chamada de tolerância a faltas.

Tolerar faltas permite que um sistema, seja ele distribuído ou não, possa continuar desempenhando as suas atividades normalmente mesmo na presença de faltas. Uma falta é a origem do problema, sendo a partir dela que um sistema pode ou não cometer erros e conseqüentemente pode ou não falhar na execução de suas atividades.

Existem diversas tecnologias que permitem a construção de sistemas distribuídos responsáveis pela troca de informações entre sistemas. Exemplos dessas tecnologias são o JavaRMI [Grosso, 2001], DCOM [Rubin and Brain, 1998] e o CORBA [Orfali and Harkey, 1998], tendo cada uma dessas tecnologias especificações e padrões próprios para a troca de informações. Com o crescimento e a evolução dos sistemas baseados nas tecnologias Web, que aproveitam da facilidade de utilização do protocolo HTTP, surgiu uma nova forma de concepção de sistemas distribuídos.

Essa nova forma define um novo paradigma que tem como principal característica fornecer interoperabilidade entre aplicações independentemente da arquitetura de hardware, sistema operacional e linguagem de programação nos quais esses sistemas foram construídos e estão sendo executados. Esse novo paradigma é chamado de arquitetura orientada a serviços (SOA) [Erl, 2005].

Um dos objetivos da SOA é permitir a colaboração e troca de informações entre aplicações, em outras palavras, a SOA permite a interoperabilidade entre aplicações. A tecnologia que permite concretizar a construção de software que utilize esse paradigma é chamada de Serviço Web. Um Serviço Web pode ser definido como sendo qualquer serviço, disponível na Internet, que se comunique utilizando protocolos padrões, baseados em XML (*eXtensible Markup Language*), e não seja limitado a uma plataforma de hardware, sistema operacional e linguagem de programação [Cerami, 2002].

Algumas aplicações, tais como um ERP (*Enterprise Resource Planning*), sistemas de comércio eletrônico, entre outros, necessitam que seu serviços, além de serem interoperáveis, possuam características de confiabilidade [Pullum, 2001]. Essas características destinam atenção especial a atributos não funcionais da aplicação, sendo um exemplo desses atributos a comunicação com outras aplicações. A comunicação com outras aplicações é crucial para o bom andamento e gerenciamento de uma organização que utiliza esses tipos de sistema. Logo, a falha de um Serviço Web que possibilita esta comunicação, se torna bastante crítica. Sabe-se que cada aplicação necessita de um determinado grau de confiabilidade, porém, para todos os casos, é importante que o serviço consiga tolerar faltas, evitando assim a parada completa da aplicação por conseqüência da manifestação destas. Além disso, não existem padrões e especificações para Serviços Web que endereçam o tratamento de falhas que possam ocorrer nesses serviços. É por esse motivo que este trabalho foi elaborado.

## 1.1 Objetivo do Trabalho

O objetivo deste trabalho é fornecer uma arquitetura de tolerância a faltas para Serviços Web que possua baixo acoplamento entre consumidor e provedor de serviço, proporcionando transparência, tratamento e recuperação de falhas ocorridas durante a utilização do serviço. Essa arquitetura toma como base os mesmos princípios presentes na SOA, ou seja, ser uma arquitetura independente de plataforma de hardware, sistema operacional e linguagem de programação utilizada. A modelagem formal de seus componentes permite a formação de uma base de suporte para o tratamento das diversas classes de faltas presentes na literatura, possibilitando a utiliza-

ção de diferentes técnicas de replicação, algoritmos de ordenação de mensagens, protocolos de comunicação de grupo, entre outras técnicas.

## 1.2 Justificativa

Serviços que são executados sobre uma infra-estrutura de rede podem sofrer com os problemas inerentes a essa infra-estrutura. Esses problemas podem causar uma indisponibilidade do serviço, podendo ser resumidamente de três tipos: problemas no serviço, na plataforma de execução (hardware e software) e na rede de comunicação. Os Serviços Web, por serem serviços executados sobre uma infra-estrutura de rede, sofrem destes mesmos problemas. Para que esses problemas não comprometam o funcionamento do serviço, são utilizadas técnicas tais como replicação de estado, detecção de falha, comunicação de grupo, entre outras técnicas, que permitam que esse serviço continue funcionando mesmo na presença de faltas. Já que uma padronização em torno dos aspectos relativos a TF para Serviços Web ainda não foi formulada, essa arquitetura tem o objetivo de fornecer TF para Serviços Web.

Existem outros trabalhos, além deste, que buscam esse mesmo objetivo [Ye and Shen, 2005, Li et al., 2005, Santos et al., 2005, Fang et al., 2007, Salas et al., 2006]. Apesar de propiciarem tolerância a faltas para Serviços Web, esses trabalhos apresentam restrições como a incompatibilidade com consumidores legados, exigência de alterações nos padrões existentes e o forte acoplamento com a tecnologia na qual foram desenvolvidos. Sendo assim, a justificativa para a especificação dessa arquitetura é o fornecimento de tolerância a faltas para Serviços Web eliminando tais restrições.

## 1.3 Metodologia

Para atingir os objetivos propostos, primeiramente foi necessário realizar uma revisão da literatura sobre tolerância a faltas, Serviços Web e seus assuntos relacionados. Em seguida foram estudados os diversos trabalhos relacionados, permitindo a análise de suas principais características e deficiências. Com base nesses estudos, foi realizada uma definição dos componentes presentes na arquitetura proposta, especificando seus comportamentos com auxílio de *CSP* (*Communicating Sequential Process*) [Hoare, 2004]. Esses componentes foram projetados e modelados na forma de classes Java [SUN, 2006] que constituem o protótipo que serve como implementação de referência dessa arquitetura. Por fim, foram realizados testes para avaliar o desempenho da implementação

de referência da arquitetura proposta.

## 1.4 Organização do texto

Para entendimento geral da arquitetura, este trabalho está organizado da seguinte forma: O capítulo 2 aborda conceitos relacionados a tolerância a faltas e às diferentes técnicas que permitem a obtenção dessa característica. O capítulo 3 descreve os conceitos relacionados a Serviços Web, objetivando o conhecimento de suas principais especificações e protocolos. No capítulo 4 são descritos os trabalhos relacionados a esta proposta, fornecendo no final do capítulo uma análise comparativa entre esses trabalhos.

Já no capítulo 5, enfim é descrita a arquitetura proposta neste trabalho, seus principais componentes e comportamentos. Na seqüência, o capítulo 6 descreve a implementação de referência da arquitetura, alguns experimentos e uma análise dos resultados desses experimentos. Por fim, no capítulo 7 são expressadas as conclusões obtidas e sugestões de trabalhos futuros sobre o tema.

# Capítulo 2

# Confiança no Funcionamento

Com a constante evolução tecnológica ocorrida nas últimas décadas, inúmeras aplicações computacionais têm sido projetadas para solucionar os diversos problemas, tais como processamento de sinais, mineração de dados, gerência de suprimentos, entre outros. Essas aplicações são desenvolvidas em diferentes plataformas de hardware, desde de dispositivos móveis, tais como celulares e *palmtops*, a equipamentos de grande porte como os utilizados em redes telefônicas e na geração e distribuição de energia. Um dos objetivos da concepção de sistemas computacionais é contribuir para resolução desses problemas da melhor maneira possível. Para isso, esses sistemas precisam estar funcionando de forma a gerar resultados corretos e confiáveis. Sistemas não confiáveis podem produzir resultados imprecisos, que venham a prejudicar a resolução desses problemas. Logo, é de extrema importância que esses sistemas sejam projetados e construídos com atributos e funcionalidades que assegurem o mínimo necessário de confiança no funcionamento.

Sendo assim, esse capítulo tem a finalidade de descrever os principais conceitos referentes a confiança no funcionamento. Primeiramente serão abordados uma conceituação básica que define o que é confiança no funcionamento, quais suas principais características e qual sua relação com tolerância a faltas. Em seguida são apresentados conceitos de tolerância a faltas em sistemas distribuídos, juntamente com as principais técnicas empregadas para prover fornecimento dessa tolerância a faltas.

## 2.1 Conceituação Básica

Confiança no funcionamento, do inglês *dependability*, pode ser definida como a propriedade de um sistema computacional fornecer serviços que podem ser justificadamente confiáveis

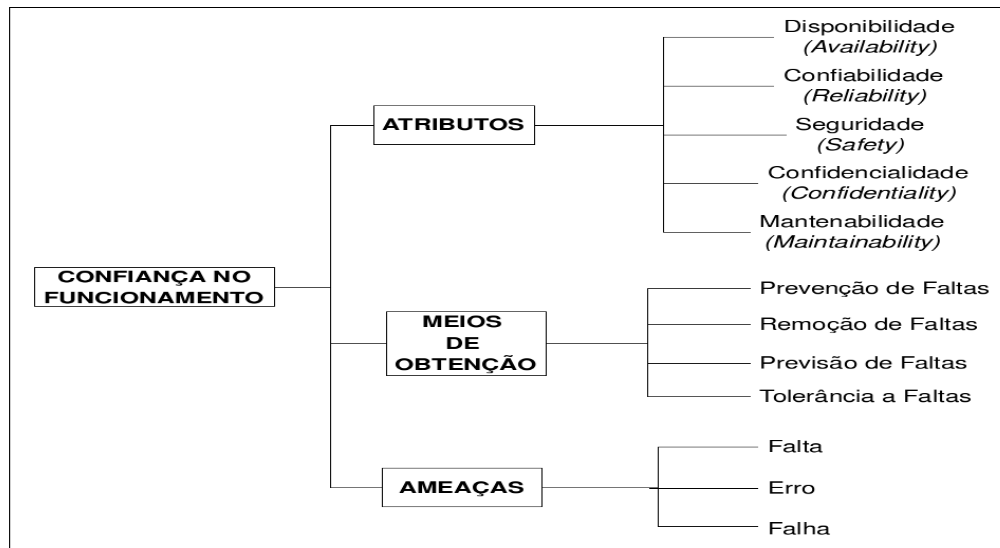



**Figura 2.1:** Árvore de conceitos da confiança no funcionamento [Avizienis et al., 2004]

[Laprie, 1995]. A figura 2.1 exibe a árvore dos conceitos relacionados a essa propriedade. Pode-se notar na figura que seus atributos podem ser obtidos com a aplicação de uma ou mais técnicas computacionais, classificadas como meios de obtenção de confiança no funcionamento de uma aplicação. Com base na literatura pode-se definir que [Shooman, 2002]:

- **Disponibilidade**: permite que serviços corretos fornecidos por um sistema estejam sempre disponíveis para utilização;
- **Confiabilidade**: define a continuidade do fornecimento correto dos serviços do sistema;
- **Seguridade (*safety*)**: ausência de conseqüências catastróficas para o ambiente e os usuários do sistema;
- **Confidencialidade**: garante que a informação fornecida pelo sistema não estará disponível para entidades não autorizadas;
- **Integridade**: garante que informações fornecidas pelo sistema são confiáveis e corretas;
- **Mantenabilidade**: capacidade de suportar reparos e modificações que auxiliam o fornecimento de todos os outros atributos;

Dependendo do sistema, diferentes graus de confiança no funcionamento podem ser aplicados e fornecidos [Laprie, 1995]. Em outras palavras, o fornecimento de um ou mais atributos de

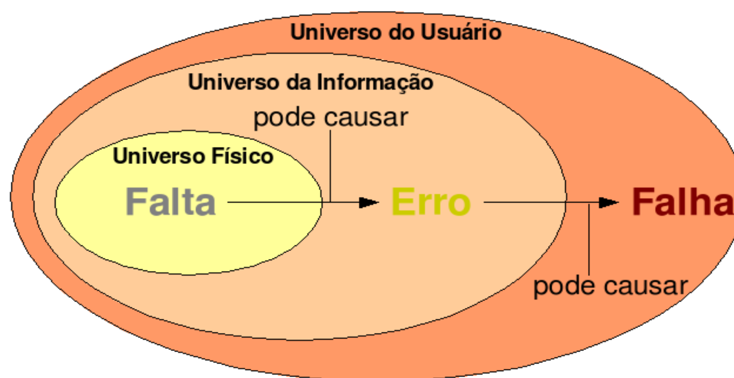



**Figura 2.2:** Relação entre falta, erro e falha no modelo de 3 universos

confiança no funcionamento permite que um sistema alcance um determinado grau dessa propriedade. Ainda na figura 2.1 é possível observar que existem ameaças que devem ser tratadas. Essas ameaças definem três conceitos importantes: falta, erro e falha. O entendimento desses conceitos permite um melhor diagnóstico das causas que levam, por exemplo, um sistema a falhar.

## 2.2 Falta, Erro e Falha

O fato de um sistema computacional possuir atributos, por exemplo, de disponibilidade e integridade, não significa que ele está livre de faltas que envolvam esses atributos. Um sistema computacional, mesmo muito bem projetado, é suscetível a faltas. A **falta** é a hipótese ou causa de um erro no sistema. Uma falta pode ser causada por um problema físico ou algorítmico do sistema computacional. Um **erro** pode ser definido como o estado de um sistema que pode levar à ocorrência de uma falha. Já a **falha** pode ser definida como sendo uma anormalidade ocorrida na execução do sistema, que implica na obtenção de resultados diferentes dos esperados e definidos pela especificação do serviço. A falha é a manifestação de um erro. Além disso, pode-se dizer que as falhas são problemas perceptíveis aos usuários desse sistema [Koren and Krishna, 2007]. É bom salientar que o termo usuário diz respeito a entidades que interagem com o sistema. Essas entidades podem ser tanto pessoas como outros sistemas.

A figura 2.2 exemplifica a relação entre os conceitos de falta, erro e falha. Quando um sistema computacional possui uma falta (ex: problema em uma região de memória) que não se manifestou na presença de um erro, pode-se dizer que essa falta é uma **falta latente** no sistema. O período de tempo que compreende desde a ocorrência da falta até a manifestação de um erro

derivado dessa falta é chamada de **latência de falta**. Em outras palavras, enquanto uma falta não se manifestar na forma de um erro, é praticamente impossível detectá-la. De forma similar, a mesmo definição pode ser aplicada ao erro com relação a sua manifestação ou não na forma de uma falha [Coulouris et al., 2005].

Para auxiliar na detecção, tratamento e remoção dessas ameaças, existem algumas técnicas, como por exemplo remoção de faltas e tolerância a faltas, que são utilizadas para obtenção de confiança no funcionamento em sistemas computacionais, aumentando assim a capacidade do sistema de fornecer serviços justificadamente confiáveis.

## 2.3 Meios para Obtenção de Confiança no Funcionamento

De acordo com Avizienis [Avizienis et al., 2004], diversas técnicas tem sido desenvolvidas com objetivo de alcançar atributos necessários ao fornecimento de confiança no funcionamento a sistemas computacionais. Considerando o objetivo principal de cada técnica, é possível realizar uma divisão em quatro grandes categorias, aqui denominadas como meios para obtenção de confiança no funcionamento. Esses meios/categorias são a prevenção de faltas, a remoção de faltas, a previsão de faltas e a tolerância a faltas [Avizienis et al., 2004].

A **prevenção de faltas** tem o objetivo de prevenir a ocorrência ou introdução de faltas no sistema. A **remoção de faltas** busca a redução da ocorrência de faltas no sistema, removendo-as antes que as mesmas se manifestem como erros. A **previsão de faltas** procura estimar a presença de faltas no sistema, suas incidências futuras e conseqüências decorrentes de suas manifestações. Por fim, **a tolerância a faltas**, tratada em maiores detalhes na seção 2.4, se preocupa em tolerar as faltas presentes no sistema, permitindo o fornecimento do serviço de forma correta e confiável mesmo com a manifestação dessas faltas.

## 2.4 Tolerância a Faltas em Sistemas Distribuídos

Antes de iniciar a explanação com conceitos de tolerância a faltas em sistemas distribuídos, é necessário definir e conceituar o que é um sistema distribuído (SD). Um SD pode ser definido como a composição de componentes, presentes em um infraestrutura de rede, que se comunicam através de um conjunto de protocolos pré-definidos, executando atividades que tenham como finalidade o alcance de um objetivo comum [Coulouris et al., 2005, Veríssimo and Rodrigues, 2001]. Outra definição sugere que um SD pode ser definido como a união de hardware distribuído, controle

distribuído e dados distribuídos, sendo tudo isso transparente para o usuário [Wu, 1999]. Pode-se perceber que ambas as definições se encontram em termos da distribuição lógica e/ou física dos componentes de um SD. Logo, pode-se dizer que um SD é um conjunto de componentes distribuídos cujo controle e forma de armazenamento dos dados é realizada também de forma distribuída, sendo sua composição totalmente transparente ao usuário.

A maior motivação para a construção de sistemas distribuídos está na facilidade de compartilhamento, distribuição e disponibilidade de serviços e recursos que esse tipo de sistema tem a capacidade de prover. Porém, um SD ainda possui algumas problemáticas derivadas principalmente da independência de localidade de seus componentes. Essas problemáticas são [Coulouris et al., 2005]:

- Heterogeneidade: Garantia de independência de hardware e software dos vários componentes do sistema distribuído;
- Abertura: Capacidade de integração com outros sistemas sem ter necessidade de realizar uma alteração significativa na estrutura da aplicação;
- Escalabilidade: Suporte á adição de mais recursos e usuários sem que o sistema tenha uma perda de desempenho significativa;
- Segurança: Garantia de que informações do sistema somente poderão ser acessadas por usuários autorizados;
- Tratamento de falhas: Torna transparente o tratamento de falhas, garantindo assim, o bom funcionamento do sistema independentemente da ocorrência ou não dessas falhas;
- Concorrência: Garantir de que os recursos poderão ser acessados por diversos usuários ao mesmo tempo.
- Transparência: Garantia de que o serviço será executado independente do acesso, localização, concorrência, entre outros aspectos.

Alguns exemplos de SD são sistemas de arquivos distribuídos de larga escala, intranets e sistemas bancários [Veríssimo and Rodrigues, 2001, Coulouris et al., 2005]. Em complemento, ainda pode-se dizer que sistemas distribuídos normalmente são acompanhados lado a lado por disciplinas tais como tolerância a faltas, tempo-real, segurança e sistemas de gestão [Veríssimo and Rodrigues, 2001]. Técnicas de tolerância a faltas auxiliam na resolução e atenuação de grande parte das problemáticas citadas anteriormente.

Tolerância a Faltas (TF) pode ser definida como a propriedade que um sistema computacional possui de fornecer seus serviços corretamente mesmo na presença de faltas [Avizienis et al., 2004]. Como em um SD podem existir diversos e diferentes componentes que são parte integral desse

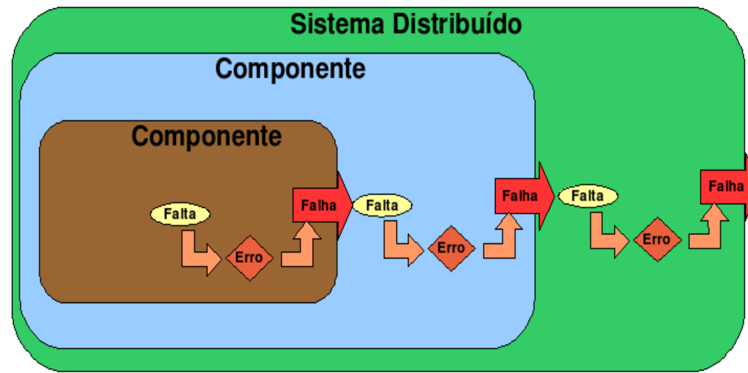



**Figura 2.3:** Representação conceitual da propagação de faltas em um SD

sistema, podendo estar localizados ao longo de uma rede, tolerar faltas em sistemas distribuídos se torna uma tarefa mais complexa de ser realizada que em sistemas centralizados. Para que um SD tenha uma falha que afete seus usuários, é necessário que as falhas ocorridas nos componentes que fazem parte da sua constituição interna sejam propagadas.

Relembrando os conceitos de falta, erro e falha introduzidos na seção 2.2, a propagação de faltas é o cascateamento de faltas, erros e falhas, onde as faltas manifestadas nos componentes do sistema são propagadas até o momento que se tornam falhas perceptíveis aos usuários desse sistema. Para auxilar no entendimento do processo de propagação de faltas, a figura 2.3 exibe uma representação conceitual dessa propagação em um SD.

Tolerância a faltas em sistemas distribuídos pode ser definida como a utilização de técnicas que inibam a propagação de faltas que podem ocorrer nos componentes pertencentes a esses sistemas. Diante de possíveis faltas de comunicação, processamento, infra-estrutura de hardware, entre outros aspectos, é necessário um bom entendimento do modelo de faltas de aplicações distribuídas [Veríssimo and Rodrigues, 2001]. Esse modelo de faltas define como e quais faltas podem ocorrer em um SD, além de prover o entendimento das conseqüências que essas faltas possam causar.

**Faltas de crash** são faltas que deixam o sistema permanentemente indisponível. Enquanto o sistema não retornar ao seu estado de operação normal, o serviço que deveria ser fornecido pelo sistema não o será. **Faltas por omissão** ocorrem quando o sistema eventualmente não executa uma determinada ação. A ocorrência de faltas por omissão pode ser ocasionada por diferentes aspectos. Na maioria dos casos, as faltas por omissão ocorrem devido à falha de um dos componentes responsáveis pela execução da ação. **Faltas por temporização** ocorrem quando o sistema não responde às requisições de seus usuários no intervalo de tempo previsto, ou seja, respondendo em

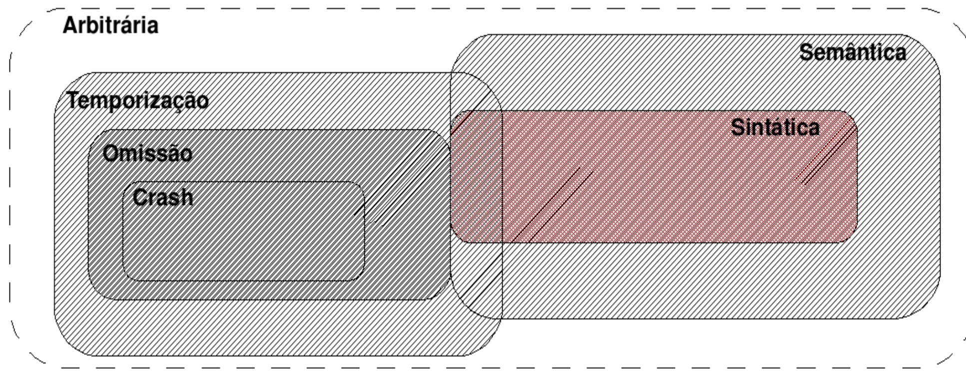



**Figura 2.4:** Classes de Faltas [Veríssimo and Rodrigues, 2001]

um período anterior ou posterior a esse intervalo. Essas faltas são principalmente ou quase que exclusivamente aplicadas a sistemas distribuídos síncronos nos quais são definidos limites de tempo para envio e recebimento de informação. Em sistemas distribuídos assíncronos, um processo pode demorar a transmitir a resposta de uma mensagem por estar sobrecarregado. Isso pode ocasionar falhas se for utilizada uma abordagem temporal nesse tipo de sistema, pois não há garantia que a mensagem será entregue em um intervalo de tempo estipulado previamente. **Faltas sintáticas** são faltas que ocorrem quando o sistema retorna uma resposta de valor incorreto que não faz parte do conjunto de valores possíveis para essa resposta. Já **faltas semânticas** ocorrem quando a resposta tem um valor incorreto para o contexto dessa resposta. Além disso, **faltas arbitrárias** são faltas que englobam todas as outras faltas que foram citadas anteriormente existindo um membro faltoso no sistema em não conformidade com sua especificação. Por fim, **faltas bizantinas**, classificadas como um subconjunto das faltas arbitrárias, são faltas de origem maliciosa que produzem saídas arbitrárias que diferem da saída esperada dentro de um contexto pré-estabelecido. Um exemplo de falta bizantina pode ser visualizado no caso de um sistema enviar diferentes saídas para diferentes sistemas com os quais interage. A figura 2.4 exibe o relacionamento entre as faltas.

Sistemas distribuídos possuem componentes distribuídos em diferentes locais, necessitando se manter em operação mesmo na presença de faltas nesses componentes. Para auxiliar nessa tarefa, técnicas computacionais são empregadas com o objetivo de tornar o sistema mais robusto, podendo ser classificado como um sistema tolerante a faltas. Entre essas técnicas destacam-se a comunicação de grupo, a replicação, a detecção e a recuperação de falhas.

### 2.4.1 Comunicação de Grupo

Distribuir componentes ao longo de uma rede, sem meios de comunicação entre esses componentes, não é suficiente para garantir, por exemplo, a disponibilidade do serviço oferecido pelo sistema. A comunicação entre esses componentes tem um papel fundamental para atingir a disponibilidade desejada. Quando componentes são reunidos para trocar informações com o intuito de atingir um objetivo comum, essa união pode ser denominada de grupo. Um grupo é compreendido como um conjunto de membros cooperantes, reunidos de alguma maneira definida pelo usuário ou pelo sistema, podendo ser endereçado como sendo uma unidade lógica única [Tanenbaum, 1995]. Cada membro do grupo, é uma unidade totalmente independente, não possuindo acesso direto aos dados presentes em outros membros. A única forma de acesso aos dados presentes em outros membros é através da troca de mensagens.

A forma de constituição de um grupo define quais serão as técnicas mais apropriadas a serem utilizadas na manutenção dos membros desse grupo. Essa constituição pode formar dois tipos de grupos [Babaoglu and Schiper, 1995]: **grupos estáticos** ou **grupos dinâmicos**. Em **grupos estáticos**, o conjunto de membros pertencentes ao grupo denominado *membership* [Wu, 1999], não muda durante todo o período de vida do grupo. Mesmo em caso de falha de um de seus membros, o *membership* permanece o mesmo. Isso significa que o membro que falhou continua sendo participante do grupo podendo, posteriormente, se recuperar dessa falha e retornar a receber as mensagens enviadas a esse grupo. Já em **grupos dinâmicos**, membros podem ser adicionados ou removidos do grupo durante todo o ciclo de vida do grupo. Para isso, devem ser utilizados mecanismos de manutenção do *membership*, a fim de manter no grupo somente membros aptos a receber mensagens.

Observando a forma de relacionamento entre os membros de um grupo, surge uma forma complementar de classificação. Essa classificação divide os grupos também em dois tipos: **grupos não hierárquicos** e **grupos hierárquicos**. Em **grupos não hierárquicos**, todos os membros do grupo tem responsabilidades e poder de decisão semelhantes. Logo, todas as decisões do grupo são obtidas através de uma votação. Além disso, a falta de hierarquização traz uma vantagem para esse tipo de grupo: não possuir membros que sejam pontos de vulnerabilidade do grupo. Porém, técnicas para alcance de consenso têm um grau de complexidade mais elevado, podendo vir a prejudicar o desempenho do serviço disponibilizado.

Já em **grupos hierárquicos** têm-se um ou mais elementos pertencentes ao grupo com responsabilidades e poder de decisão superiores ao demais membros. Por exemplo, podem existir grupos

onde somente um único membro se comunica com entidades externas, repassando as requisições dessas entidades para os demais membros do grupo quando decidirem que isso é necessário. Esses membros com responsabilidades adicionais se tornam pontos de vulnerabilidade do grupo, sendo denominados de pontos únicos de falha (do inglês *single point of failure*). Nesse tipo de grupo, é importante a adição de mecanismos de eleição que permitam que qualquer um dos membros assumam as responsabilidades dos membros de maior hierarquia, caso esses membros venham a falhar. Nesse tipo de grupo, a implementação das técnicas para alcance de consenso podem ser menos complexas, ocasionando uma possível queda de desempenho do serviço disponibilizado.

Para comunicação entre os membros de um grupo, estático ou dinâmico, hierárquico ou não hierárquico, são utilizadas primitivas de comunicação. Como a possibilidade de ocorrências de faltas durante o processo de comunicação entre seus membros não pode ser descartada, a utilização de primitivas de comunicação de grupo é necessária para, ao menos, garantir confiabilidade no serviço de troca de mensagens. Cada uma dessas primitivas fornece diferentes níveis de confiabilidade e ordenação das mensagens que são trocadas entre os membros. Essas diferenças fazem com que a escolha de uma dessas primitivas afete diretamente a consistência das informações presentes nos membros do grupo. Logo, encontram-se na literatura diversas primitivas de comunicação de grupo, cada qual com seu propósito. São elas: a difusão confiável, difusão FIFO, difusão causal e difusão atômica.

A **difusão confiável** [Ekwall et al., 2004] tem como principal característica a garantia de que todas as mensagens enviadas para o grupo serão recebidas por todos os membros ativos desse grupo. Isso quer dizer que somente os membros não faltosos pertencentes a um determinado grupo receberão todas as mensagens enviadas para o mesmo. Para isso, a técnica de difusão confiável deve satisfazer algumas características, tais como:

- Validade: Se algum membro do grupo difunde uma mensagem nesse grupo, então algum membro não faltoso do grupo entregará a mensagem ou nenhum membro pertencente ao grupo está correto (ou seja, não apresenta falha);
- Acordo: Se um membro correto do grupo entrega uma mensagem, então todos os processos corretos pertencentes a esse grupo entregarão essa mesma mensagem.
- Integridade: Se uma mensagem foi previamente difundida no grupo, então cada processo correto pertencente a esse grupo entrega no máximo uma vez essa mensagem.

A **difusão FIFO (*First-In-First-Out*)** (figura 2.5) é uma variante da técnica de difusão con-

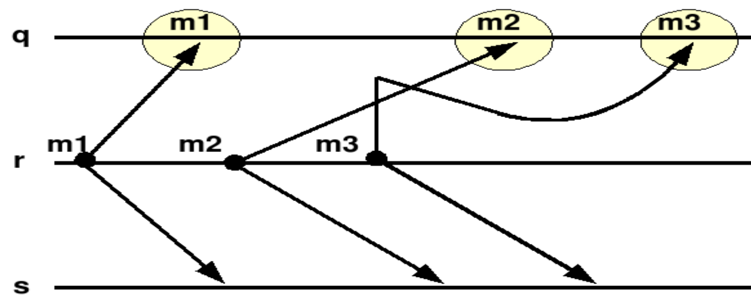



**Figura 2.5:** Difusão FIFO [Veríssimo and Rodrigues, 2001]

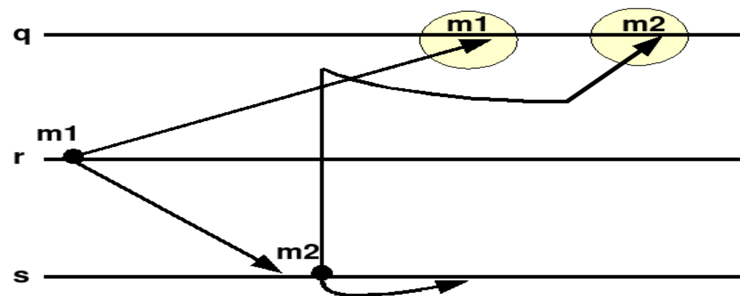



**Figura 2.6:** Difusão Causal [Veríssimo and Rodrigues, 2001]

fiável que garante que mensagens difundidas por um membro do grupo serão recebidas pelos outros membros na mesma ordem em que foram enviadas. Para realizar a implementação da difusão FIFO, é preciso realizar algum tipo de marcação nas mensagens, afim de que se possa manter sua ordem de difusão. Isso pode ser alcançado, inserindo um número seqüencial no cabeçalho de cada mensagem. Esse número representará a ordem que deve ser respeitada na entrega de cada mensagem.

A **difusão causal** (figura 2.6) surge como uma técnica alternativa que supre uma deficiência da técnica de difusão FIFO. A utilização de difusão FIFO é somente adequada quando as mensagens emitidas para o grupo em questão partem de um mesmo membro. Na difusão causal, a ordenação das mensagens é realizada com base na teoria dos relógios lógicos de Lamport [Lamport, 1978]. Logo, na difusão causal, se uma mensagem $m$ precede causalmente a difusão de outra mensagem $n$ difundida no grupo, então nenhum membro correto entregará $n$ antes de entregar $m$.

Por fim, a **difusão atômica** (figura 2.7) é uma técnica que garante que todas as mensagens difundidas no grupo sejam recebidas por todos os membros na mesma ordem que foram enviadas.

Nesse caso, o ideal é que todos os membros tenham a mesma visão do grupo, evitando assim a necessidade de comunicações adicionais para manutenção da consistência. Sua implementação pode ocorrer de diferentes formas, cada qual com sua abordagem específica. Existem algumas formas de implementação da difusão atômica, tais como:

- Histórico de Comunicação: Nesse caso, o algoritmo baseado em histórico de comunicação faz com que os membros do grupo recebam as mensagens em uma ordem total, tomando como base o algoritmo de ordenação de eventos de Lamport [Lamport, 1978];

- Baseado em privilégio: Nessa abordagem existe a formação de um grupo virtual de membros que podem difundir mensagens. Dentro desse grupo virtual, é utilizado um *token* que delimita quem tem o privilégio de difundir mensagem para os membros do grupo real naquele instante de tempo. Baseado nesse privilégio, é estabelecida uma ordenação total, já que existe uma seqüência determinística de passagem do *token*.

- Seqüenciador Fixo: Ao utilizar a abordagem com seqüenciador fixo, um dos membros do grupo é eleito o seqüenciador. Dessa forma, todas as mensagens difundidas para o grupo, primeiro devem passar pelo membro seqüenciador. Esse membro estabelece uma ordem nas qual as mensagens devem ser entregues por todos os membros do grupo.

- Seqüenciador Móvel: É uma variação do seqüenciador fixo. A idéia do seqüenciador móvel é prover um grau de tolerância a faltas no membro seqüenciador. Isso ocorre de forma análoga à abordagem baseada em privilégios. Nesse caso, o membro que estiver de posse do *token* define a ordem de entrega das mensagens em um determinado instante de tempo. A ordem total então é estabelecida com a seqüência determinística de passagem do *token* entre os membros seqüenciadores.

Além disso, pode-se utilizar uma combinação entre as primitivas de comunicação de grupo formando assim um tipo de difusão com características que possam suprir a necessidade específica de cada nova formação de grupo. As técnicas de replicação provêm diversas formas de composição de grupo e permitem que diversas primitivas de comunicação possam ser utilizadas para a obtenção de um grupo consistente.

### 2.4.2 Replicação

Técnicas de replicação, também conhecidas como técnicas de redundância, são técnicas aplicadas tanto ao nível de hardware quanto software capazes de prover, juntamente com outras

**Figura 2.7:** Difusão Atômica [Veríssimo and Rodrigues, 2001]

técnicas, tolerância a faltas a um sistema, possibilitando dessa forma um aumento na confiança de funcionamento do mesmo. Uma de suas principais características é a formação de grupos de membros que representam unicamente uma aplicação. Esses membros são denominados de réplicas por serem cópias idênticas de uma mesma entidade. Na literatura, a replicação é tida como a técnica chave para fornecimento de TF em sistemas distribuídos [Coulouris et al., 2005, Veríssimo and Rodrigues, 2001]. Em um SD, a replicação tem implicações cruciais no desempenho do sistema. Pelo fato de réplicas estarem distribuídas ao longo da rede, surge a necessidade de comunicação entre elas. Essa comunicação tem o objetivo de manter o estado das réplicas consistente. Quando um usuário acessa um determinado serviço que encontra-se replicado em uma rede, o desempenho está diretamente ligado a quantas operações de atualização dos dados acontecerão em um determinado intervalo de tempo [Wolfson et al., 1997].

Quando somente consultas às informações do serviço são realizadas, a réplica que recebeu essas consultas pode executá-las localmente, não sendo necessária a propagação dessas operações para as demais réplicas. Porém, a cada vez que uma informação sofre uma alteração, essa alteração deve ser propagada para as demais réplicas do serviço [Baldoni et al., 1998]. Essa propagação é responsável pela perda de desempenho do serviço e pela manutenção da consistência de estado entre as réplicas. Logo, existem técnicas de replicação presentes na literatura com a finalidade de facilitar a organização, formação e gerenciamento de réplicas de um serviço, definindo como essa propagação de operações deve ser realizada. Essas técnicas são a replicação ativa, replica passiva e a replicação semi-ativa.

Na **replicação ativa**, todas as réplicas executam a mesma requisição solicitada por um usuário do serviço. Essa técnica também é conhecida como máquina de estados, já que todas as réplicas não faltosas são ativas. Uma réplica ativa pode ser definida como uma entidade que recebe e processa requisições de usuários do sistema. Logo, na replicação ativa as réplicas representam máqui-

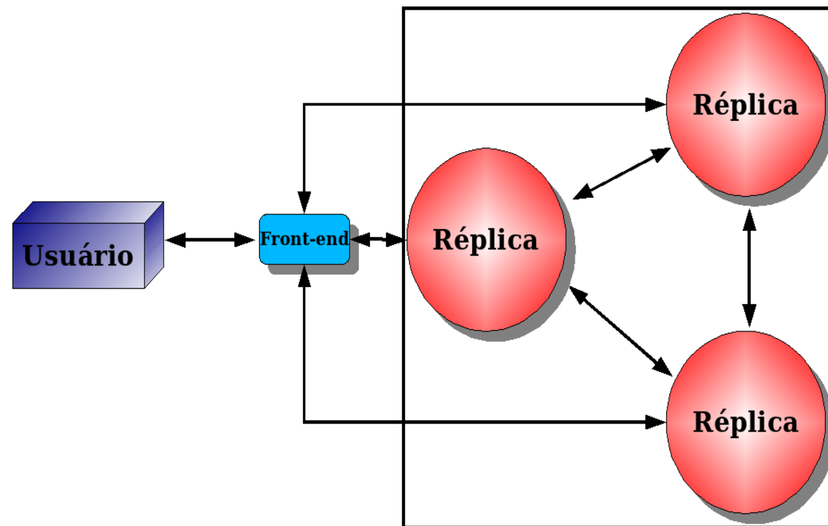



**Figura 2.8:** Esquema de Funcionamento da Replicação Ativa

nas de estado com regras equivalentes, sendo organizadas como um grupo [Coulouris et al., 2005]. Esse grupo não possui uma hierarquia definida, sendo que todas as réplicas presentes tem o mesmo poder de decisão, processando e respondendo requisições enviadas por um usuário do serviço fornecido.

Essa técnica deve manter um grupo determinístico, ou seja, o número de réplicas do grupo deve ser conhecido previamente, já que todas as réplicas são ativas. Caso essa técnica fosse utilizada com uma formação de grupo não determinística, teria uma alta probabilidade do surgimento de problemas de inconsistência entre as réplicas. A figura 2.8 demonstra o esquema de funcionamento da replicação ativa.

Pode-se observar que, no esquema de replicação ativa, todas as réplicas se comunicam entre si. Essa comunicação é de extrema importância, pois é assim que a manutenção da consistência do estado das réplicas é obtida. Essa comunicação também pode vir a ter o papel de detecção da falha de réplicas do grupo. Essa falha, na replicação ativa, é mascarada quase que instantaneamente. Como todas as réplicas devem processar a requisição, técnicas para armazenamento e retenção de resultados devem ser aplicadas na tentativa de atenuar o impacto negativo que essa técnica de replicação insere no desempenho do serviço fornecido.

Na **replicação passiva**, também conhecida como cópia primária, somente uma réplica recebe e executa as requisições provenientes de usuários do serviço. Em outras palavras, somente uma réplica do grupo é considerada ativa, sendo as demais réplicas consideradas réplicas *backups*. A réplica ativa é denominada réplica primária ou réplica mestre. A iteração com o grupo de réplicas

fica mais simplificado já que somente a réplica primária mantém contato direto com os usuários. Esse isolamento das demais réplicas propicia uma maior transparência do número total de réplicas do grupo.

No momento que uma requisição de usuário é solicitada, a réplica primária tem a tarefa de identificar se essa requisição solicitará uma atualização sobre os dados do serviço. Em caso negativo, a réplica primária simplesmente processa a requisição e devolve a resposta sem que as demais réplicas tenham conhecimento do ocorrido. Porém, em caso positivo, a réplica primária tem o papel de realizar a sincronização de estado com as demais réplicas, mantendo assim consistente todas as réplicas pertencentes ao grupo. Para a realização dessa atualização pode ser definido um mecanismo que leve em consideração algum tipo de regra. Essa regra pode ser feita com base temporal, número de atualizações, entre outras possíveis combinações. Quanto maior a freqüência de atualizações realizadas, maior será o impacto no desempenho do serviço, já que essas atualizações também consomem tempo e uma certa largura de banda do canal de comunicação.

Além disso, na replicação passiva é importante que as demais réplicas do grupo verifiquem periodicamente se a réplica primária continua ativa. A partir do momento que alguma das réplicas identifica a falha da réplica primária, um algoritmo de eleição é executado no domínio do grupo e uma nova réplica primária é eleita, mantendo o último estado de atualização realizado.

A transparência de faltas é outro fator importante na replicação passiva. Dentro dessa técnica, pode-se definir, de acordo com o serviço, qual o numero mínimo de réplicas necessárias para que o serviço se mantenha ativo. A definição desse número mínimo de réplicas tem relação com a(s) classe(s) de falta(s) que deseja(m) ser suportada(s) pelo grupo.

Existe ainda uma terceira classificação proposta que é denominada **replicação semi-ativa**. Essa técnica é uma mescla das duas técnicas citadas anteriormente, combinando assim algumas características dessas duas técnicas. Na replicação semi-ativa todas as réplicas recebem e executam as requisições solicitadas pelas entidades externas. Porém, somente uma das réplicas fica responsável por ordenar a execução e retornar a resposta da requisição para a entidade externa. A réplica com essa responsabilidade é denominada líder, tendo a responsabilidade de realizar a intercomunicação das entidades externas e os outros membros pertencentes ao grupo de réplicas do serviço. A figura 2.10 demonstra a arquitetura de funcionamento da replicação semi-ativa.

Por se tratar de uma mescla das duas abordagens citadas anteriormente, a replicação semi-ativa tem algumas vantagens e desvantagens em relação as outras duas abordagens. O número total de réplicas presentes no serviço pode ser transparente, assim como na replicação passiva. Isso pode ocorrer, caso seja implementado um algoritmo de reintegração de réplica no grupo,

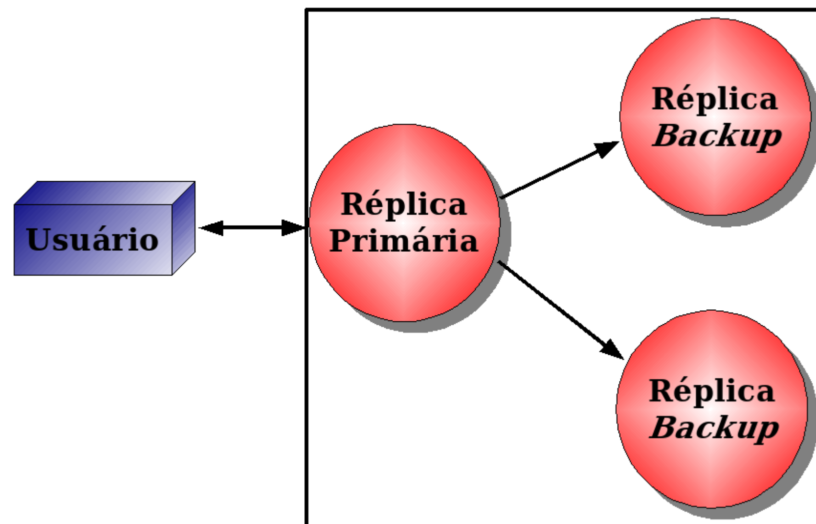



**Figura 2.9:** Esquema de Funcionamento da Replicação Passiva

sendo que a réplica que deseja ser reintegrada deve ser bloqueada até que a mesma esteja com o estado sincronizado com as demais réplicas. Porém isso influencia e deixa mais complexo o algoritmo de eleição de um novo líder em caso de falha do atual. Caso seja utilizado um número de réplicas fixo, o trabalho do algoritmo de eleição é facilitado. Em caso de falha do atual líder, o algoritmo somente necessita escolher qualquer outra réplica para assumir a liderança, já que todas as réplicas do grupo são ativas e executam todas as operações executadas pela réplica líder. Para ganho de desempenho, variações da replicação semi-ativa fazem com que somente o líder execute operações que não alterem o estado do serviço. Em outras palavras, somente as operações que realizam alteração de estado são executadas por todas as réplicas do serviço.

### 2.4.3 Detecção de Falhas

O processo de detecção de falhas é uma técnica considerada fundamental para sistemas distribuídos [Hayashibara et al., 2004, Défago et al., 2005]. Em sistemas tolerantes a faltas, esse processo é quase tão importante do que o próprio serviço fornecido pelo sistema. O principal objetivo da detecção de falhas é indicar o mais rápido possível a falha de um dos componentes do sistema. Com isso, a falha pode ser confinada, tratada e, se possível, recuperada, evitando assim a propagação da mesma [Chandra and Toueg, 1996, Hayashibara et al., 2004, Défago et al., 2005, Larrea et al., 2004].

O processo de detecção de uma falha pode ser considerado uma tarefa de análise de di-

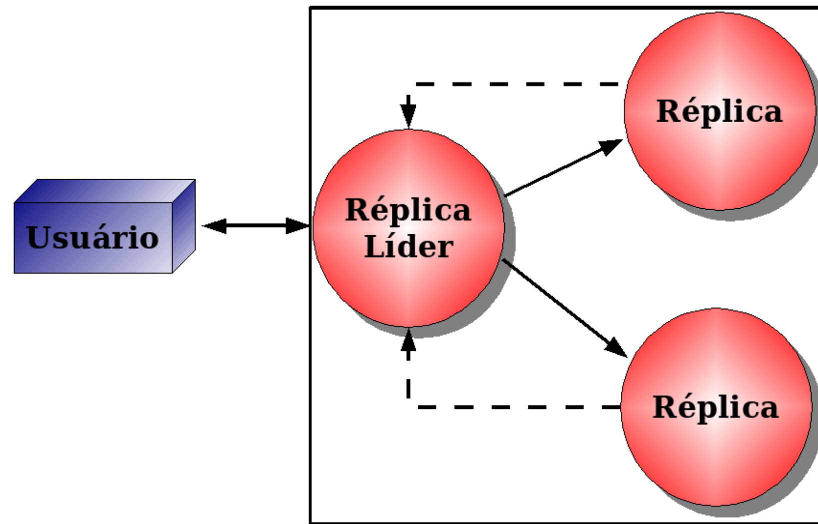



**Figura 2.10:** Esquema de Funcionamento da Replicação Semi-ativa

versos fatores, que relacionados, se encaixam em algum padrão que indiquem, com propriedade, a ocorrência dessa falha. Como um SD é composto por diversos componentes que são suscetíveis a falhas, é importante que falhas ocorridas nesses componentes sejam identificadas. Para realizar essa tarefa é necessária a existência de um outro componente com a responsabilidade única de detectar falhas. Esse componente é chamado de **detector de falhas** (DF). Segundo [Chandra and Toueg, 1996], detectores de falhas devem satisfazer as seguintes propriedades: completude (*completeness*) e exatidão (*accuracy*). Sendo o DF um outro componente, é fundamental que seja estabelecido um canal de comunicação entre o componente monitorado e o detector, possibilitando que o processo de detecção ocorra.

Segundo [Veríssimo and Rodrigues, 2001] esse canal de comunicação pode ser classificado em dois tipos: canais perfeitos e canais imperfeitos. Canais perfeitos são meios de comunicação entre componente monitorado e detector que permitem que mensagens sejam trocadas sem sofrer qualquer tipo de perda ou interferência. Em outras palavras, oferecem suporte para que o detector possa identificar a falha do componente monitorado com bastante precisão. Detectores de falha perfeitos [Chandra and Toueg, 1996] são um exemplo de detectores que utilizam canais perfeitos. Exemplos de canais de comunicação perfeitos são mecanismos de checagem de paridade que podem ser implementados em memória, disco ou barramento.

Já canais imperfeitos são meios de comunicação suscetíveis a falha, podendo sofrer perda ou atraso de informações que estejam trafegando por esses canais. Logo, fica mais complexo para o detector afirmar que o componente monitorado falhou, sendo mais complexo distinguir

se a falha foi do componente monitorado ou do meio de comunicação entre eles. Detectores de falhas imperfeitos [Chandra and Toueg, 1996] são um exemplo de detectores que utilizam canais imperfeitos. Exemplos de canais de comunicação imperfeitos são as LANs (*Local Area Networks*) e as WANs (*Wide Area Networks*).

Ao contrário dos detectores de falha perfeitos, detectores imperfeitos não conseguem diferenciar problemas com o canal de comunicação, da falha de um processo correto participante do sistema. Nesse sentido, Chandra e Toueg [Chandra and Toueg, 1996] definiram variações das propriedades de completude e exatidão, formalizando a possibilidade de definição de diferentes classes de detectores de falhas. Logo, pode-se ter um detector de falha, que atue sobre uma canal imperfeito (ex: LAN), mas possua características de funcionamento de detectores de falhas perfeitos. O principal problema desses detectores é a influência que problemas no canal de comunicação podem causar sobre o seu funcionamento. Isso quer dizer que um processo correto pode ser considerado falho, caso o canal de comunicação entre ele e o detector de falha adicione arbitrariamente um tempo de atraso na troca de mensagens entre essas entidades.

Em redes de computadores que possuem ligações de baixa velocidade e atraso acentuado, é praticamente impossível a implantação deste tipo de detector. A tentativa de simulação de um canal de comunicação perfeito sobre um meio físico que possui um tempo de atraso variável pode provocar diferentes problemas no processo de detecção no decorrer da execução do sistema. Uma alternativa que deve ser considerada na tentativa de evitar que processos corretos sejam constantemente considerados falhos, e assim, atrapalhem a execução do sistema ao qual fazem parte, é a adoção de técnicas de ajuste de *timeout* dinâmico ou a utilização de consenso no processo de detecção de falha, possibilitando que um número menor de processos corretos sejam indevidamente considerados falhos.

Assumindo que os detectores de falha somente se preocupam com falhas de *crash*, pode-se definir dois métodos de falha: *push* e *pull*. No **método pull**, os componentes monitorados enviam periodicamente mensagens *heartbeats* do tipo *"I am alive!"*, informando aos detectores que estão ativos. Se o detector não receber a mensagem no limite de tempo esperado, ele passa a suspeitar do componente. A desvantagem desse método é que o componente monitorado precisa conhecer a freqüência em que as mensagens devem ser enviadas. No **método PUSH** os detectores periodicamente enviam mensagens aos componentes questionando se os mesmos estão ativos. Essas mensagens são do tipo *"Are you alive?"*, aguardando a resposta dos componentes em um determinado limite de tempo. Se a resposta não for enviada pelo componente dentro desse limite de tempo, o detector também passa a suspeitar desse componente.

Em ambos os métodos, se o serviço não enviar a resposta dentro de um novo intervalo, o detector considera que o componente falhou e, a partir desse momento, pode iniciar uma série de ações. Essas ações podem incluir desde a notificação para outros detectores (quando existirem) que o componente monitorado falhou até a inicialização do processo de recuperação de falha desse componente.

### 2.4.4 Recuperação de Falhas

As técnicas de recuperação de falhas propiciam ao SD maior flexibilidade para o tratamento das falhas que possam ocorrer em seus componentes. Essas técnicas também são muito importantes para a manutenção da TF presente em um sistema. Sem a recuperação de falhas, é muito complexo e custoso manter o fornecimento de TF em um nível aceitável. Para recuperar uma falha, é necessário primeiramente detectá-la. Logo, técnicas de recuperação dependem diretamente do estágio de detecção de falhas. Ou seja, assim que é identificada a falha de um componente, o processo de recuperação de falhas deve ser ativado, objetivando a recuperação desse componente do sistema. Para isso, é necessário o armazenamento confiável de informações que sirvam de base para execução dos métodos de recuperação de falhas. Essas informações podem conter dados relativos ao estado do componente antes da falha, mensagens recebidas pelo componente dentro de um intervalo de tempo, operações que foram executadas, entre outros dados relevantes. O armazenamento dessas informações pode ser realizado em diversos instantes de tempo. Os pontos que delimitam o momento de obtenção desas informações são chamados de pontos de recuperação. Pontos de recuperação são marcações no tempo que permitem saber o estado do componente em instantes de tempo pré-definidos.

O processo de recuperação pode ser realizado de dois modos: **recuperação por avanço** e **recuperação por retrocesso**. Na recuperação por avanço, o componente faltoso é levado a um estado posterior ao estado presente no momento da falha. Esse novo estado deve permitir que o componente volte ao seu funcionamento habitual, de forma correta e confiável. Esse modo de recuperação é bastante utilizado em sistemas de tempo real. Já a recuperação por retrocesso utiliza das informações coletadas pelos pontos de recuperação para retornar ao último estado armazenado antes do momento da falha do componente. A recuperação por retrocesso é uma técnica mais simples, porém em SDs, onde componentes tem um grau de dependência muito elevado, a tarefa de recuperação pode se tornar complexa. Essa complexidade está na tarefa de recuperação de estado de um componente, que pode exigir a recuperação de estados de outros componentes dos

quais o componente faltoso depende.

## 2.5 Considerações sobre o capítulo

Esse capítulo apresentou diversos conceitos referentes a confiança no funcionamento de sistemas computacionais, dando ênfase para o tópico de tolerância a faltas em sistemas distribuídos. Com isso foi possível entender que ao adicionar réplicas de um serviço para aumentar a confiabilidade e disponibilidade do mesmo, a degradação no desempenho da aplicação se torna inevitável. Porém, se as técnicas de replicação forem estudadas corretamente e adaptadas para as características e necessidades do serviço, essa perda de desempenho pode ser bastante reduzida.

Além disso, como em um SD podem existir diversos e diferentes componentes que são parte integral desse sistema, podendo estar localizados ao longo de uma rede, a aplicação dos conceitos abordados no capítulo, tais como difusão confiável, replicação, detecção e recuperação de falhas, permitem que esses sistemas sejam tolerantes a faltas e agreguem características de confiabilidade e disponibilidade em sua execução.

Uma das tecnologias que tem sido bastante utilizada na concepção de SDs é a de Serviços Web [Ayala et al., 2002]. Sendo assim, conceitos relacionados com essa tecnologia que tem como principal característica a utilização de protocolos independentes de hardware/software e linguagem de programação, são abordados no próximo capítulo.

# Capítulo 3

# Serviços Web

As tecnologias para desenvolvimento de software têm se voltado para a construção de sistemas que possam estar disponíveis para qualquer pessoa que possua acesso à rede mundial de computadores, a Internet e autorização para acesso a esses sistemas. Para isso, a Web tem colaborado constantemente como alicerce no desenvolvimento de sistemas com essa característica [Bernes-Lee and Cailliau, 1990]. Junto a isso surgiu a necessidade de integração entre os sistemas, obrigando a incorporação de técnicas que permitissem a troca de informações, possibilitando a realização de computação distribuída entre esses softwares.

A computação distribuída visa a execução de diferentes tarefas em diversos computadores conectados por uma rede de comunicação. os sistemas distribuídos surgiram no momento em que diversos softwares passaram a trocar informações entre si e formaram um conjunto integrado de sistemas. Para auxiliar nessa troca de informações entre esses diversos softwares, surge um conceito chamado de Serviço Web. Um Serviço Web pode ser definido como sendo qualquer serviço, disponível na Internet, que se comunique utilizando protocolos padrões, baseados em XML (*eXtensible Markup Language*), e não seja limitado a um determinado sistema operacional e/ou linguagem de programação [Cerami, 2002].

Aproveitando toda a infraestrutura fornecida pela Web (tecnologias, ferramentas, protocolos, entre outros) cada serviço pode representar uma funcionalidade específica, um componente integral de uma aplicação ou até mesmo uma aplicação completa. Com isso, os Serviços Web se tornaram uma ferramenta atrativa para a construção e concepção de sistemas distribuídos [Ye and Shen, 2005]. Os principais protocolos utilizados para a construção de qualquer Serviço Web possibilitam a adição de interoperabilidade aos serviços [Ayala et al., 2002], levando à sua caracterização como um sistema aberto.

Essa interoperabilidade facilita de forma significativa o desenvolvimento de aplicações distribuídas, já que problemas tais como a diversidade de *hardwares* e *softwares* utilizados são solucionados pelos protocolos utilizados por esses serviços [Newcomer, 2002]. Como dito anteriormente, Serviços Web são bastante utilizados na integração de sistemas, possibilitando que esses sistemas troquem informações de uma maneira uniforme. Toda dinâmica de funcionamento de Serviços Web é definida por uma arquitetura própria. Essa arquitetura é denominada *Service Oriented Architecture* (SOA) [Erl, 2005], que em português pode ser traduzida para Arquitetura Orientada a Serviços.

Nesse capítulo serão abordados os principais conceitos sobre a SOA tais como seu modelo conceitual, principais entidades e o relacionamento dessas entidades com a base de tecnologias e padrões que envolvem Serviços Web. Tais tecnologias e padrões adotados para a construção de Serviços Web serão descritos em seguida. Adicionalmente, será descrita a WS-Reliable Mesaging, que está relacionada com o trabalho proposto nessa dissertação. Para finalizar, será fornecida uma visão geral de tolerância a faltas em Serviços Web.

## 3.1 Arquitetura de Serviços Web

O principal objetivo da SOA é permitir a colaboração e troca de informações entre aplicações. A tecnologia de Serviços Web pode ser então classificada como uma implementação concreta das entidades e características presentes na SOA. A SOA fornece um modelo conceitual que serve de base para construção de qualquer Serviço Web. Nesse modelo conceitual são definidas três entidades que formam seu alicerce principal. Essas entidades são o provedor de serviços, o consumidor de serviços e o registro de serviços (figura 3.1).

O **provedor de serviços** é a entidade que fornece o serviço propriamente dito. O provedor registra seu serviço no registro de serviços, para que consumidores possam encontrá-lo e utilizar suas funcionalidades. O **registro de serviços** é a entidade que mantém informações sobre a descrição dos serviços disponíveis e suas localizações na rede. Já o **consumidor de serviços** utiliza as informações encontradas na entidade de registro de serviços para invocar as funcionalidades do provedor de serviço escolhido. Para a interação entre essas entidades foram definidas três operações conceituais que juntamente com essas entidades formam a base da SOA. A operação "*publish()*"ocorre quando o provedor do serviços interage com o registro, enviando a esse registro informações com a finalidade de descrever suas próprias características e fornecer informações sobre o serviço provido. A operação "*find()*"permite que o consumidor realize uma consulta no

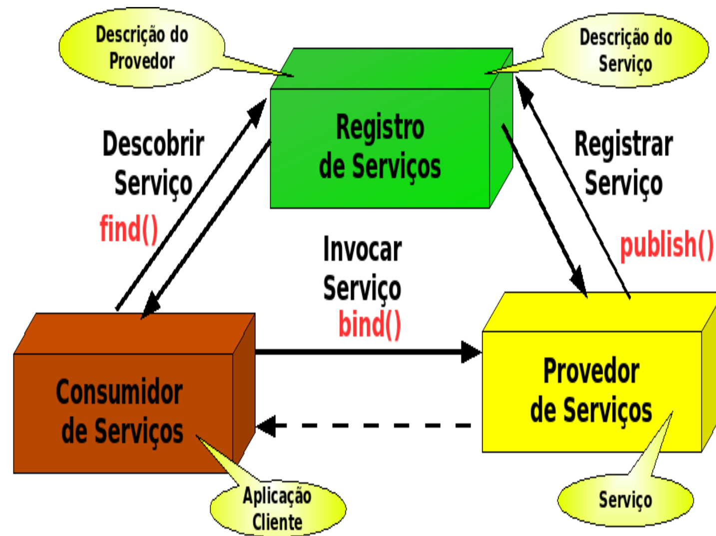



**Figura 3.1:** Modelo Conceitual da SOA

registro e obtenha as informações registradas pelo provedor durante a operação "*publish()*". Já a operação "*bind()*"ocorre quando o consumidor (aplicação cliente), de posse de informações sobre o provedor e o serviço propriamente dito, troca mensagens com esse provedor, podendo assim efetuar a invocação do serviço.

É importante salientar que toda a comunicação realizada entre as entidades que fazem parte do modelo conceitual de Serviços Web é realizada com protocolos baseados em XML. Entre esses protocolos destacam-se o SOAP [W3C, 2007a], o WSDL (*Web Services Description Language*) [W3C, 2007b] e o UDDI (*Universal Description, Discovery and Integration*) [OASIS, 2004a], utilizados para troca de mensagens, descrição e descoberta de serviços, respectivamente. Juntamente com outros protocolos responsáveis pelo transporte de mensagens, esses padrões e protocolos formam a base tecnológica dos serviços Web [Cerami, 2002, Tindwel et al., 2001]. Essa base, representada pela figura 3.2, contém quatro camadas que especificam os protocolos e padrões utilizados pela maioria das aplicações desenvolvidas com Serviços Web. Essa pilha não contem protocolos e padrões que contemplem questões relativas a segurança, confiabilidade, integridade e composição de serviços, além de outras características importantes para diversas aplicações implementadas com Serviços Web. Porém, essas características podem ser adicionadas com o auxílio de especificações que complementem essa base de protocolos fundamentais para Serviços Web.

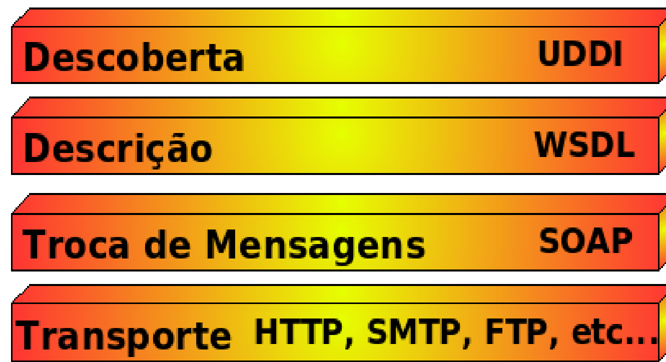



**Figura 3.2:** Pilha de protocolos utilizados em Serviços Web

## 3.2 Camada de Transporte

A camada de transporte é a camada base dos protocolos utilizados em Serviços Web. Essa camada tem a responsabilidade de transportar as mensagens trocadas entre consumidor e provedor de serviços, atuando logo acima da camada de rede. Na camada de transporte são usados os mesmos protocolos utilizados na Web, tais como HTTP (*HyperText Transfer Protocol*), SMTP (*Simple Mail Transfer Protocol*), FTP (*File Transfer Protocol*), entre outros. Em Serviços Web, os dados transportados possuem o formato de mensagens SOAP e o protocolo mais utilizado é o HTTP. Apesar disso, sua utilização não é obrigatória, permitindo que outros protocolos possam ser utilizados como meio de comunicação entre duas aplicações que trocam informações com auxílio de Serviços Web.

## 3.3 Camada de Troca de Mensagens (SOAP)

A camada de troca de mensagens é responsável pela formatação da informação que será transmitida entre as entidades presentes no modelo conceitual da SOA (figura 3.1). Para realizar essa tarefa, foi definido um protocolo com um nível de abstração alto suficiente para que qualquer combinação de sistema operacional e linguagem de programação pudessem ser usados para criar e executar aplicações capazes de trocar informações ente si. Esse protocolo foi nomeado de SOAP. SOAP é um protocolo baseado em XML que define um conjunto de regras que especificam o formato da mensagem, não se preocupando com detalhes de como essa mensagem será transmitida. SOAP é uma especificação mantida pela W3C e é a evolução de uma outra especificação denominada XML-RPC (*XML Remote Procedure Call*), uma tentativa inicial de definir um padrão para

**Figura 3.3:** Representação dos elementos de uma mensagem SOAP

realizar chamadas de procedimentos remotos utilizando mensagens em XML [Cerami, 2002].

Além de permitir a realização de RPC (*Remote Procedure Call*), SOAP pode ser utilizado para EDI (*Eletronic Document Interchange*), permitindo a troca de informações entre aplicações sem a necessidade obrigatória da realização de algum tipo de invocação de método. Na especificação do SOAP, quando uma mensagem representa uma chamada ou resposta a um procedimento remoto, essa mensagem é classificada como "RPC-Style". Já quando trata-se de uma mensagem para troca eletrônica de documentos é dita "Document-style". O uso mais comum é para troca de mensagens do tipo "RPC-style", já que RPC é a base da computação distribuída [Ayala et al., 2002].

Assim como qualquer documento XML, mensagens SOAP possuem uma estrutura hierárquica bem definida. Essa estrutura é a base dos protocolos de troca de mensagens "RPC-Style"e "Document-style". Na figura 3.3 é possível visualizar o esquema de organização dos elementos que compõem uma mensagem SOAP. Pode-se notar que o envelope, definido pela *tag* XML *<env:Envelope>*, é o elemento raiz de uma mensagem SOAP. Dentro dele é definido o conteúdo da mensagem que será trocada entre consumidor e provedor do serviço. É com auxílio desse elemento que um documento XML pode ser identificado como uma mensagem SOAP.

Na seqüência, tem-se o cabeçalho. Esse elemento é identificado pela tag XML *<env:Header>* e permite a inclusão de informações adicionais que auxiliem o processo de troca de mensagens entre as aplicações. Sua declaração é opcional, porém, caso seja declarado, deve ser o primeiro elemento dentro do envelope. Um exemplo de informação que é inserida no cabeçalho é o número de seqüência para controle da ordenação e entrega confiável de uma mensagem. Tanto a especi-

ficação WS-Reliability quanto o componente de ordenação de mensagens da arquitetura proposta nesse documento, ambos vistos mais adiante no texto, utilizam esse número para controlar suas ações.

Junto com o envelope, o corpo da mensagem é um dos elementos mais importantes de uma mensagem SOAP. Declarado pela tag XML *<env:Body>* esse elemento permite a definição dos dados que serão enviados pela aplicação. É dentro do corpo que, por exemplo, são inseridas as chamadas de procedimentos remotos e seus parâmetros. Além disso, dentro do corpo também pode ser declarado um elemento opcional que serve para reportar problemas no processamento da mensagem ou na execução do serviço. Esse elemento é denominado falta. O elemento falta somente pode ser declarado em uma mensagem SOAP de resposta. Ele é identificado pela tag XML *<env:Fault>* e dentro desse elemento pode-se declarar o tipo da falta e a mensagem que informa o motivo da ocorrência dessa falta. Com esse conjunto de elementos, o SOAP permite a troca de informações e chamadas de métodos que possibilitam a interoperabilidade entre aplicações.

## 3.4 Camada de Descrição (WSDL)

A camada de descrição é responsável por disponibilizar mecanismos que auxiliem os consumidores a obter informações sobre as características técnicas de um determinado serviço, possibilitando a utilização posterior do serviço por parte desses consumidores. A especificação WSDL é a responsável por fornecer subsídio para realização da descrição dessas características. Essa especificação é o padrão para descrição de Serviços Web, sendo mantida pela W3C [W3C, 2007b]. Pode-se dizer que a WSDL desempenha um papel similar ao papel desempenhado pela *Interface Definition Language* (IDL) na arquitetura CORBA (*Common Object Request Broker Architecture*).

Um arquivo de descrição WSDL pode ser logicamente dividido em descrições concretas e abstratas (3.4). Essa subdivisão é delimitada por um conjunto de elementos que pertencem a cada uma dessas classificações. Nas descrições abstratas, é possível a identificação de quatro elementos:

- **<wsdl:types>** : utilizado para declarar no arquivo WSDL um tipo de dado utilizado pelo serviço;
- **<wsdl:message>** : permite a definição do conteúdo das mensagens trocadas entre consumidor e serviço;
- **<wsdl:portType>** : permite a definição abstrata de todas as operações disponíveis no Serviço Web;

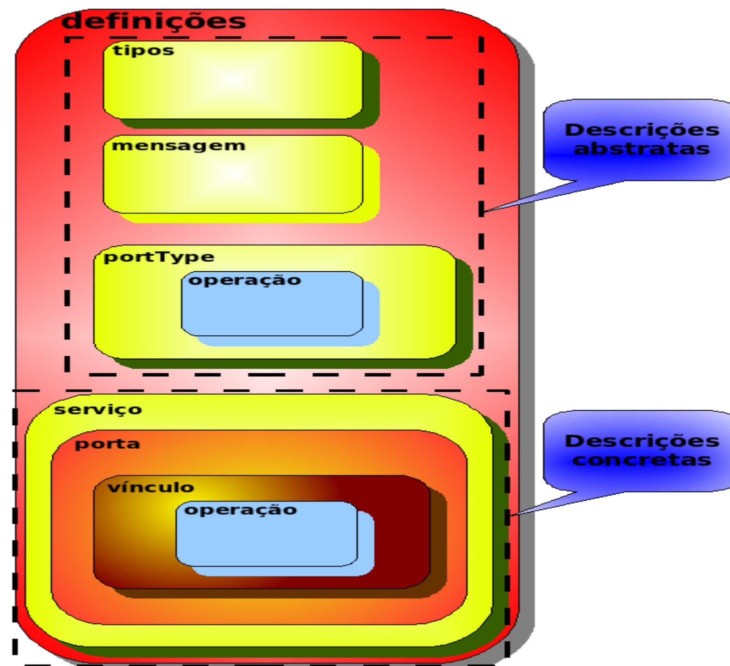



**Figura 3.4:** Organização de um documento WSDL

- **<wsdl:operation>** : permite a definição de uma operação (funcionalidade) presente no Serviço Web.

Além dos elementos que descrevem o serviço de forma mais abstrata, existem elementos que contribuem para uma descrição mais concreta do serviço. Esses elementos descrevem os detalhes técnicos que precisam ser realizados para obter acesso às funcionalidades do serviço em questão. Os elementos que fazem parte dessa descrição são:

- **<wsdl:service>** : define o conjunto de portas utilizadas pelo serviço;

- **<wsdl:port>** : permite a especificação do endereço IP e da porta nos quais o Serviço Web estará esperando por requisições de consumidores do serviço;

- **<wsdl:binding>** : possui a responsabilidade de realizar a ligação entre os elementos abstratos e os elementos concretos do documento WSDL, além de permitir a definição de quais são os protocolos que devem ser utilizados para comunicação com o Serviço Web.

A figura 3.4 permite a visualização do esquema que representa a organização de um documento WSDL. Relembrando o que já foi dito anteriormente, esse documento representa uma descrição completa do Serviço Web, possuindo todas as informações necessárias para que um cliente possa realizar uma troca de mensagens com esse Serviço Web.

## 3.5 Camada de Descoberta (UDDI)

A camada de descoberta é responsável por disponibilizar mecanismos que permitem o armazenamento de informações referentes, por exemplo, à localização e a forma de utilização de provedores de serviços. Esses mecanismos são descritos na especificação UDDI, que é padronizada pela OASIS [OASIS, 2004a]. A especificação do registro UDDI está baseada no WSDL e no SOAP, outras duas padronizações que são essenciais para Serviços Web. Seu principal objetivo é disponibilizar formas de armazenamento e classificação dos serviços. Em outras palavras, os registros UDDI são responsáveis por armazenar informações sobre quem publicou o serviço, suas especificações técnicas (documento WSDL) e de negócio.

Os registros UDDI podem ser públicos, privados ou restritos. Registros públicos são registros que podem ser pesquisados por qualquer consumidor de serviços. Registros privados são registros utilizados apenas internamente em uma organização. Já os registros restritos são registros que podem ser acessados por um conjunto finito de organizações. Normalmente esse tipo de registro é implantado entre organizações que têm relações de negócio entre si.

Dentro de um registro UDDI as informações podem ser compostas por três tipos de entradas: páginas brancas, amarelas e verdes. Páginas brancas contém informações básicas, tais como nomes e telefones de contato da organização que publicou o serviço. Utilizando informações contidas nessa páginas, é possível, por exemplo, listar todos os serviços disponibilizados por essas organizações. Nas páginas amarelas são disponibilizadas informações relativas à classificação do negócio associado a um determinado serviço. Com auxílio dessas informações é possível descobrir, por exemplo, quais os serviços disponíveis em um determinado nicho de mercado. Já as páginas verdes contém informações técnicas sobre os serviços, como por exemplo seu arquivo de descrição WSDL, possibilitando que consumidores possam descobrir como um serviço deve ser acessado.

## 3.6 WS-Reliable Messaging

WS-Reliable Messaging (WS-RM) [OASIS, 2004b] é uma especificação baseada no SOAP que permite a troca de mensagens confiáveis entre aplicações. Para isso, emissor e receptor devem entrar em acordo sobre as características e o nível de qualidade de Serviço (QoS) utilizados na entrega confiável dessas mensagens. Segundo a especificação [OASIS, 2004b], a confiabilidade fornecida na entrega de mensagens é baseada em quatro semânticas:

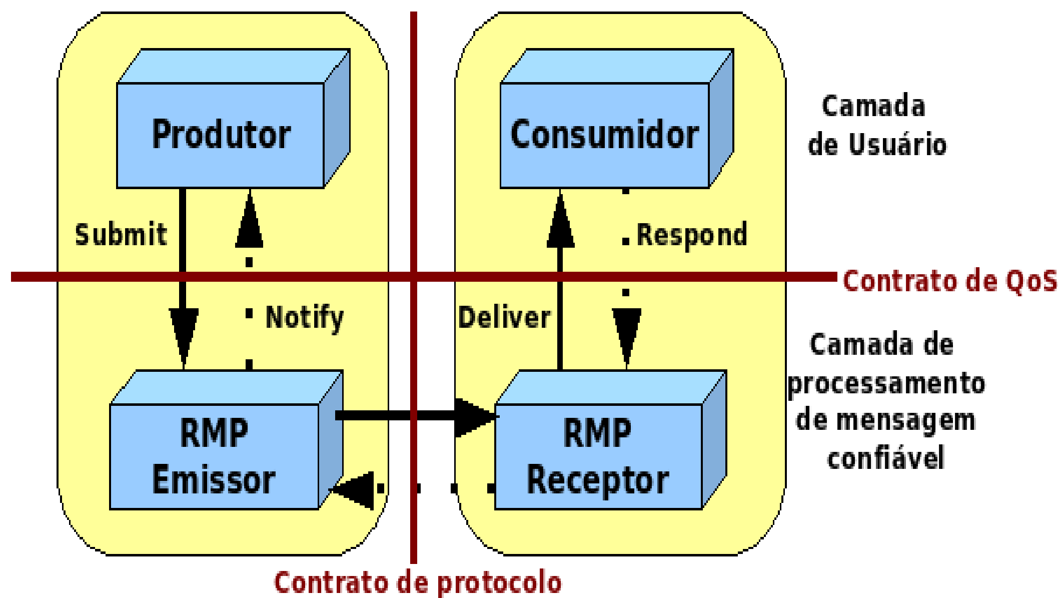



**Figura 3.5:** Contratos de entrega confiável de mensagens

- At-Least-Once: Nessa semântica o emissor pode enviar múltiplas mensagens para o receptor enquanto não receber uma confirmação de chegada de suas mensagens. Logo, em caso de falha das mensagens de confirmação, o receptor pode executar uma mesma mensagem mais de uma vez;

- At-Most-Once: Nessa semântica, mensagens serão executadas no máximo uma vez. Essa característica é garantida pela aplicação de filtros para detecção de mensagens duplicadas e retransmissão de respostas sem executar novamente uma mesma mensagem;

- Exactly-Once: Nessa semântica mensagens serão executadas exatamente uma vez e sem duplicação;

- In-Order (FIFO): Nesse semântica, um grupo de mensagens serão entregues na mesma ordem em que foram enviadas.

A figura 3.5 mostra os contratos de confiabilidade e as operações efetuadas entre entidades para processamento de mensagens confiáveis e aplicações de usuário. Esses contratos definem qual a semântica, ou uma combinação delas, que será utilizada na troca de mensagens entre emissor e receptor. Pode-se notar na figura 3.5 que os processadores de mensagens confiáveis (RMP) trocam mensagens em seu nível para satisfazer as características impostas pelos contratos de confiabilidade. Do lado emissor, a operação de **Submit** permite que a aplicação solicite a RMP o envio de

uma mensagem contendo os dados repassados a ele. Caso essa mensagem não possa ser entregue ou retorne uma resposta para o RMP emissor, a operação *Notify* é invocada, notificando uma possível falha no envio da mensagem ou entregando a resposta da mensagem enviada à aplicação emissora.

Quando uma mensagem é recebida pelo RMP receptor, a operação **Delivery** é invocada par que os dados contidos nessa mensagem sejam repassados para a aplicação receptora. Caso a mensagem precise de resposta por parte da aplicação receptora, a operação **Respond** é invocada, fazendo com que o RMP receptor envie uma resposta para a mensagem confiável recebida anteriormente. A especificação WS-RM não define detalhes de como o acordo entre emissor e receptor, para entrega confiável de mensagens, deve ser implementado. Na especificação somente são definidas características que devem ser seguidas e respeitadas pelas implementações que forem realizadas. uma das deficiências dessa especificação é a definição de ordem somente entre duas entidades, entidade emissora e entidade receptora. Na arquitetura proposta nesse trabalho, existe a necessidade de ordenação das mensagens levando em consideração um grupo de entidades que podem enviar e receber mensagens simultaneamente. Logo, foi especificado um mecanismo próprio para a ordenação das mensagens que permitisse o suporte para que um acordo sobre a ordem na entrega das mensagens, entre mais de duas entidades, pudesse ser realizado.

## 3.7 Tolerância a Faltas em Serviços Web

Serviços que são executados sobre uma infra-estrutura de rede podem sofrer com os problemas inerentes a essa infra-estrutura. Esses problemas podem causar uma indisponibilidade do serviço, podendo ser resumidamente de três tipos: problemas no serviço, na plataforma de execução (hardware e software) e na rede de comunicação. Uma das principais características fornecidas pelos padrões adotados na construção de Serviços Web é a interoperabilidade [Ayala et al., 2002]. Os Serviços Web, por serem serviços executados sobre uma infraestrutura de rede, sofrem dos mesmos problemas que foram introduzidos no início dessa sessão. Portanto, técnicas de tolerância a faltas devem ser aplicadas com o objetivo de minimizar o impacto negativo da falha de fornecimento de um determinado serviço, causada pela manifestação desses problemas.

Com base em diversas fontes da literatura pode-se definir que tolerância a faltas (TF) em Serviços Web é a capacidade de fornecimento do serviço requisitado, mesmo na presença de faltas [Salas et al., 2006, Fang et al., 2007, Santos et al., 2005, Ye and Shen, 2005, Li et al., 2005, He, 2004, Aghdaie and Tamir, 2002, Dialani et al., 2002]. O fornecimento de TF está fundamen-

tado na utilização de técnicas que forneçam um alicerce sólido para que uma determinada falta possa ser tolerada da melhor maneira possível, de modo a não afetar o funcionamento do software quando esta se manifestar. Nessa base fundamental, técnicas tais como Replicação (seção 2.4.2), detecção de falha (seção 2.4.3), comunicação de grupo (seção 2.4.1), recuperação de falhas (seção 2.4.4), entre outras técnicas, se tornam importantíssimas no intuito de fornecer TF a um nível aceitável.

## 3.8 Considerações sobre o capítulo

Nesse capítulo foram apresentados os principais conceitos e padrões utilizados por Serviços Web na concepção de sistemas. Esses padrões são caracterizados pela independência de plataforma de hardware e software, proporcionando maior abertura ao sistemas que usufruem dessa tecnologia. Além dos padrões e especificações tais como SOAP, WSDL e UDDI, que formam o alicerce de Serviços Web, foi apresentada a especificação WS-RM que permite a adição de confiabilidade na comunicação entre consumidor e serviço, possibilitando a ordenação e eliminação da duplicação das mensagens trocadas. Como essa especificação foi elaborada para troca de mensagens confiáveis entre entidades com relacionamento um-para-um, suas características não cobrem a formação de grupos de Serviços que troquem mensagens entre si e necessitem de ordenação das mensagens entre todos os membros do grupo, já que esses serviços utilizam o relacionamento um-para-muitos na difusão de mensagens para o grupo.

Além disso, até o momento não foram especificadas padronizações que tivessem o objetivo de fornecer confiança no funcionamento para Serviços Web. Nesse sentido, a união das técnicas de TF, apresentadas anteriormente, juntamente com os padrões utilizados em Serviços Web se tornam bastante atrativas para que os serviços possam fornecer mais confiança às aplicações que os utilizam. Aproveitando essa lacuna em aberto, o próximo capítulo descreve alguns trabalhos relacionados que apresentam propostas de arquiteturas e infra-estruturas de software que têm o objetivo de fornecer TF para Serviços Web.

# Capítulo 4

# Trabalhos Relacionados

Como dito no final do capítulo anterior, Serviços Web estão suscetíveis a falhas por serem serviços executados sobre uma infra-estrutura de rede. Porém, esse motivo não é o único responsável pelos possíveis problemas com esses serviços. Todo software é sujeito a ter problemas, sejam eles de origem física ou lógica. O mal funcionamento do processo de leitura de dados de um disco rígido, ou, o problema de escrita em uma região específica da memória RAM, podem fazer com que um software tenha problemas e pare de funcionar mesmo que sua parte lógica esteja correta. Outra situação passível de ocorrer são problemas na plataforma de software cuja falha impossibilite a utilização do sistema.

Para ajudar a fornecer TF para esses serviços, diversos trabalhos foram elaborados e propostos com o objetivo de atenuar o impacto negativo da manifestação das faltas que ocorrem nesses serviços, possibilitando que os mesmos continuem em funcionamento mesmo na presença de faltas. Sendo assim, os principais trabalhos relacionados com a proposta de arquitetura oferecida nessa dissertação são apresentados. Esses trabalhos relacionados são infra-estruturas e *frameworks* que tem o objetivo de tolerar faltas em Serviços Web. Nesse capítulo, cada um desses trabalhos é apresentado em linhas gerais, com destaque para as descrições de seus principais componentes e princípios de funcionamento. O conteúdo apresentado nesse capítulo possibilita, que nos capítulos posteriores, possa ser feita uma comparação com a arquitetura proposta nessa dissertação de maneira mais clara e objetiva.

## 4.1 A Middleware for Replicated Web Services

Ye e Shen [Ye and Shen, 2005] desenvolveram um *middleware* de suporte à concepção de Serviços Web confiáveis. Esse *middleware* é responsável por manter a consistência de estado das réplicas do serviço com base na técnica de replicação ativa. Cada réplica é composta por duas entidades: um proxy do Serviço Web (PWSS) e o serviço Web propriamente dito (WSS). Requisições de clientes enviadas ao serviço são interceptadas pelo PWSS. Logo, o PWSS é o *middleware* entre o cliente e o WSS. Apesar da utilização da replicação ativa, clientes somente necessitam enviar suas requisições para uma das réplicas do serviço. O PWSS presente nessa réplica tem a responsabilidade de realizar um multicast dessa requisição para as demais réplicas e retornar o resultado para o cliente. Na figura 4.1 é possível realizar a visualização conceitual de uma réplica contendo o PWSS e o WSS.

Pode-se observar na figura 4.1 que clientes do serviço devem ser construídos com auxílio de um componente de software chamado de RWS. O RWS é escrito na linguagem Java [SUN, 2006] e possui um conjunto de classes responsáveis por realizar a interação entre a aplicação cliente e o PWSS do serviço. Utilizando esse componente, a aplicação cliente tem uma imagem única do grupo de réplicas, percebendo esse grupo de réplicas como um único serviço.

Cada PWSS é composto por dois módulos: um módulo *message handler* (MH) e um gerente de comunicação de grupo (GCM). O MH é responsável por (a) processar as mensagens SOAP recebidas de consumidores, (b) armazenar o identificador dessas mensagens, (c) armazenar cópias das respostas das requisições enviadas pelos consumidores, enviando essas respostas novamente para esses consumidores (se necessário), e, (d) tratar as falhas ocorridas no WSS presente na réplica. Já o GCM (*Group Communication Manager*) é responsável por (a) realizar o *multicast* das requisições dos clientes para as demais réplicas do serviço utilizando o protocolo TOPBCAST [Hayden and Birman, 1996], e, (b) monitorar a falha dos outros PWSSs.

Para manter a consistência das réplicas do serviço, os PWSSs devem executar todas as requisições dos clientes na mesma ordem. A definição dessa ordem é baseada em uma lista das operações que realizam alterações no estado do serviço. Essas operações são denominadas de operações não-comutativas. Operações que não realizam alterações no estado do serviço não são ordenadas. Requisições enviadas para operações não-comutativas distintas não são ordenadas entre si. Logo, cada operação possui sua própria fila de entrega, sendo que diferentes operações podem ser executadas simultaneamente pelo WSS, já que requisições são classificadas por essas operações e ordenadas em suas respectivas filas de entrega.

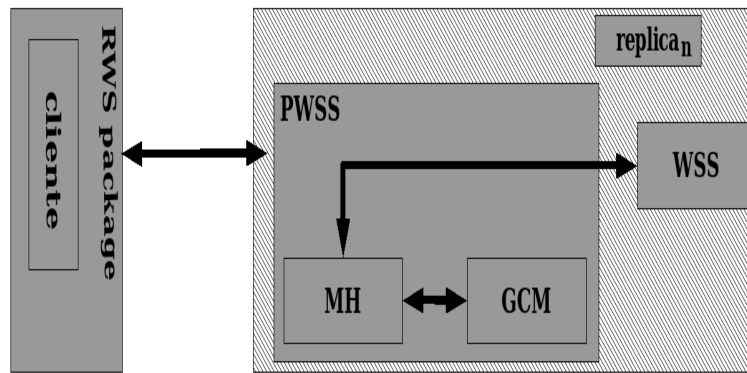



**Figura 4.1:** Representação de uma réplica contendo o PWSS e o WSS [Ye and Shen, 2005]

Esse *middleware* permite que PWSS e WSS sejam distribuídos no mesmo *host* ou em *hosts* distintos. Se estiverem em *hosts* distintos, caso o WSS falhe, o PWSS pode continuar funcionando normalmente. Caso o PWSS falhe antes do envio de requisições de clientes, essa falha é identificada pelas classes do componente RWS no momento do envio da requisição a esse PWSS. Nesse caso, a requisição é reenviada para outro PWSS. Caso a falha ocorra após o recebimento da requisição de algum cliente, podem ocorrer duas situações: (a) a requisição é enviada para as outras réplicas, ou, (b) a requisição é perdida com a falha. As classes do RWS identificam a falha e reenviam a requisição para outro PWSS. Nesse PWSS, o módulo MH elimina a duplicação da mensagem, caso essa duplicação seja identificada, devolvendo a resposta da requisição ao cliente do serviço.

## 4.2 A Framework to Support Survivable Web Services

o *framework* proposto por [Li et al., 2005] possibilita a adição de características de confiança no funcionamento a Serviços Web. O SWS *framework* tem como principal objetivo manter um determinado serviço em funcionamento mesmo na presença de faltas ou ataques de segurança. Ataques que segurança podem comprometer os dados fornecidos pelo serviço já que podem violar a confidencialidade e integridade do sistema. Para que esses ataques sejam tratados pelo SWS *framework*, os mesmos foram modelados como faltas bizantinas. Além disso, esquemas de replicação e protocolos baseados em Redundância N-Modular [Hopkins, 1978] são utilizados para realizar o mascaramento das faltas.

Com o SWS, um Serviço Web é replicado de forma que as réplicas sejam organizadas em

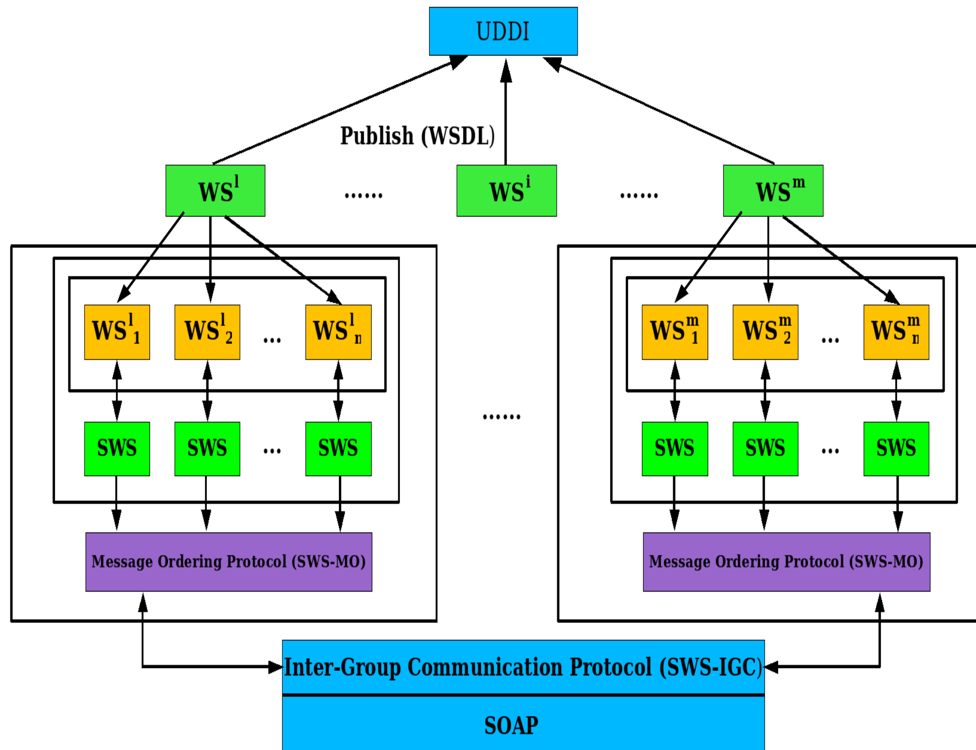



**Figura 4.2:** Modelo de Sistema do SWS *Framework* [Li et al., 2005]

um grupo. Pode-se notar na figura 4.2 que, por exemplo, $ws^l$ representa um grupo de Serviços Web composto por *n* réplicas. Cada uma das réplicas mantém informações sobre o *membership* do grupo, incluindo localização dos membros, status e pares de chaves para autenticação. Para facilitar a localização do grupo, modificações no registro UDDI foram realizadas com o intuito de manter informações referentes a todas as réplicas presentes nesse grupo em uma única entrada. Essa informação é necessária para clientes do serviço que precisam saber a localização de todas as réplicas. Como o SWS utiliza replicação ativa, todas as réplicas de um determinado grupo processam a mesma requisição de forma independente. Isso é extremamente necessário para a tolerância de faltas bizantinas.

Além disso, para facilitar a comunicação entre grupos distintos e a ordenação das mensagens dentro de um grupo, foram criados protocolos que têm essas responsabilidades. O protocolo para comunicação inter-grupos é chamado de *SWS Inter-Group Communication Protocol* (SWS-IGC) enquanto que o protocolo para ordenação de mensagens é denominado *SWS Message Ordering Protocol* (SWS-MO). O SWS-IGC é um protocolo que permite a comunicação entre dois grupos distintos de réplicas, facilitando o processo de troca de informações entre esses grupos. Para a utilização desse protocolo, o grupo que está fornecendo o serviço é chamado de grupo servidor,

enquanto que o grupo que irá utilizar esse serviço é chamado de grupo requisitor. Suponha um grupo requisitor $ws^l = (ws^l_1, ws^l_2, ..., ws^l_n)$ e um grupo servidor $ws^m = (ws^m_1, ws^m_2, ..., ws^m_n)$. Dentro do grupo $ws^l$, é eleita aleatoriamente uma réplica líder chamada de $ws^l_L$. Quando o grupo $ws^l$ realiza uma requisição para o grupo $ws^m$, somente a réplica $ws^l_L$ envia essa requisição para todas as réplicas do grupo $ws^m$. As outras réplicas presentes no grupo $ws^l$ só tem a responsabilidade de verificar se as computações realizadas no grupo servidor foram efetuadas com sucesso. Quando as réplicas do grupo $ws^m$ recebem a requisição, cada réplica processa e devolve a resposta para todas as réplicas do grupo $ws^l$. Para assegurar que a verificação será realizada corretamente, o $ws^l_L$ assina e anexa um caminho de certificação à requisição que foi enviada.

O envio das requisições para o serviço replicado com o SWS pode ser realizado de duas formas: (a) se o cliente for replicado, ou seja, for um grupo de réplicas que realiza requisições para um grupo do serviço, o protocolo SWS-IGC pode ser utilizado para comunicação entre o grupo de réplicas clientes e o grupo de réplicas do serviço, ou, (b) se o cliente não é replicado então esse cliente deve enviar requisições para todas as réplicas do grupo que fornece um determinado serviço, aguardar as respostas de todas as réplicas e aplicar algum mecanismo de votação que determinará a resposta correta.

## 4.3 FTWEB: A Fault Tolerant Infrastructure for Web Services

O FTWEB [Santos et al., 2005] é uma proposta de infra-estrutura de TF para Serviços Web. Essa infra-estrutura é baseada no FT-CORBA [OMG, 2002], definindo um conjunto de componentes responsáveis por invocar métodos nas réplicas do serviço, aguardar seu processamento, analisar as respostas retornadas por essas réplicas e devolver a resposta ao cliente. A figura 4.3 exibe a organização dos componentes presentes nessa infra-estrutura.

A principal idéia presente no FTWEB é a utilização da técnica de replicação ativa para fornecer TF para Serviços Web. Para isso, réplicas do serviço formam um único grupo, onde todas as réplicas ativas desse grupo recebem, executam e respondem as requisições enviadas por clientes. Para que o serviço se mantenha consistente, o FTWEB utiliza a abordagem de seqüenciador, possibilitando que as requisições sejam ordenadas e executadas na mesma ordem em todas as réplicas do serviço. Essa infra-estrutura utiliza as idéias e conceitos presentes no FT-CORBA para utilizar objetos CORBA que possuem métodos que são invocados como se fossem Serviços Web. Para realizar essa invocação de métodos em objetos CORBA e a tradução dessas invocações para os protocolos e padrões utilizados em Serviços Web, foram definidos alguns componentes com essa

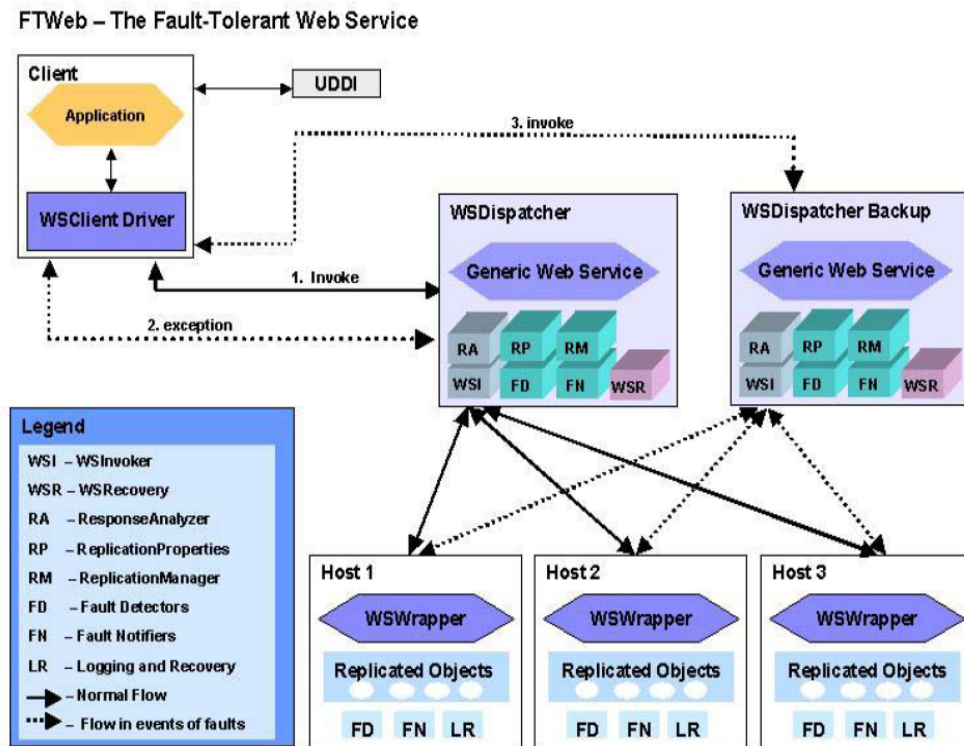



**Figura 4.3:** Infra-estrutura FTWEB [Santos et al., 2005]

responsabilidade. Esses componentes (presentes na figura 4.3) são o *WSClientDriver*, *WSDispatcher* e o *WSWrapper*.

O *WSClientDriver* é um componente utilizado junto com a aplicação cliente responsável por enviar as requisições desse cliente para o serviço. Além de enviar e receber as requisições do serviço, sua função é detectar a falha do *WSDispatcher Engine* (componente que recebe todas as requisições), e, em caso de detectar sua falha, encaminhar as requisições para o *WSDispatcher Engine Backup*. A figura 4.3 apresenta o fluxo normal (passo 1) e o fluxo alternativo (passo 3) que é executado em caso da falha do *WSDispatcher Engine*.

O *WSDispatcher Engine* é o componente central da infra-estrutura FTWEB. Esse componente é responsável por gerenciar as réplicas, invocar os métodos dessas réplicas de forma concorrente, analisar as respostas dessas invocações, detectar falhas de réplicas e iniciar o processo de recuperação de estado das réplicas. Para realizar todas essas tarefas, o *WSDispatcher Engine* é formado pelos seguintes componentes: *Generic Web Service*, *WSInvoker*, *ResponseAnalizer*, *ReplicationManager*, *ReplicationProperties*, *FaultDetector*, *FaultNotifier* e *WSRecovery*. O *GenericWebService* é responsável pela interação com o cliente do Serviço Web. Esse componente é quem recebe e responde as requisições dos clientes. O *WSInvoker* é responsável por realizar as in-

vocações dos métodos nas réplicas do Serviço. Após obter a resposta dessas invocações, esse componente repassa essas resposta para o componente *ResponseAnalizer* que tem a responsabilidade de realizar um votação e escolher a resposta que será enviada ao cliente. O *ReplicationManager* é responsável por gerenciar o grupo de réplicas do serviço. Esse componente estende as funcionalidades do gerenciamento de réplicas do FT-CORBA. O *ReplicationProperties* é responsável pelo mapeamento das propriedades de TF definidas no FT-CORBA para o FTWEB. Essas propriedades definem o estilo de replicação, o estilo e intervalo de monitoração do detector de falhas, *time-out* para resposta das requisições por parte das réplicas e a forma de recuperação dessas réplicas. O *FaultDetector* é o componente responsável pela detecção da falha de alguma das réplicas do serviço. Quando uma falha é detectada, esse componente aciona o *FaultNotifier* que tem a responsabilidade de notificar essa falha avisando ao *ReplicationManager* que uma das réplicas do serviço falhou. Por fim, após confirmada a falha de uma réplica, o *WSRecovery* tem a responsabilidade de tentar recuperar o estado da réplica faltosa.

Já o *WSWrapper* é responsável pela integração dos objetos CORBA com as tecnologias de Serviços Web. Esse componente converte requisições SOAP em invocações CORBA, utilizando a interface de invocação dinâmica que permite a chamada de métodos presentes em qualquer objeto CORBA. Com isso, é possível replicar os objetos CORBA em servidores dispersos geograficamente, deixando a administração desses objetos por conta do *WSDispatcher Engine*.

## 4.4 Fault Tolerant Web Services

Assim como o FTWEB [Santos et al., 2005], o FT-SOAP [Fang et al., 2007] é uma infra-estrutura de TF para Serviços Web baseada no FT-CORBA. No FT-SOAP, a infra-estrutura de suporte a TF em Serviços Web é baseada na replicação passiva. Isso significa que existe somente uma réplica responsável por receber e responder as requisições de clientes do serviço. A infra-estrutura é composta por quatro componentes que fornecem, segundo [Fang et al., 2007], quatro funcionalidades fundamentais para TF: *replication manager*, *fault detector*, *fault notifier* e mecanismos de *logging/recovery*. Além desses componentes, o FT-SOAP realiza uma extensão da WSDL adicionando um elemento identificado pela *tag* <WSG/>.

O *replication manager* tem a responsabilidade de realizar o gerenciamento das réplicas dentro do grupo de Serviços Web formados pelo FT-SOAP. Para isso esse componente gerencia a constituição do grupo, adicionando e/ou removendo réplicas, com o objetivo de manter o *membership* atualizado e consistente. O *fault detector* é o componente responsável pela detecção de

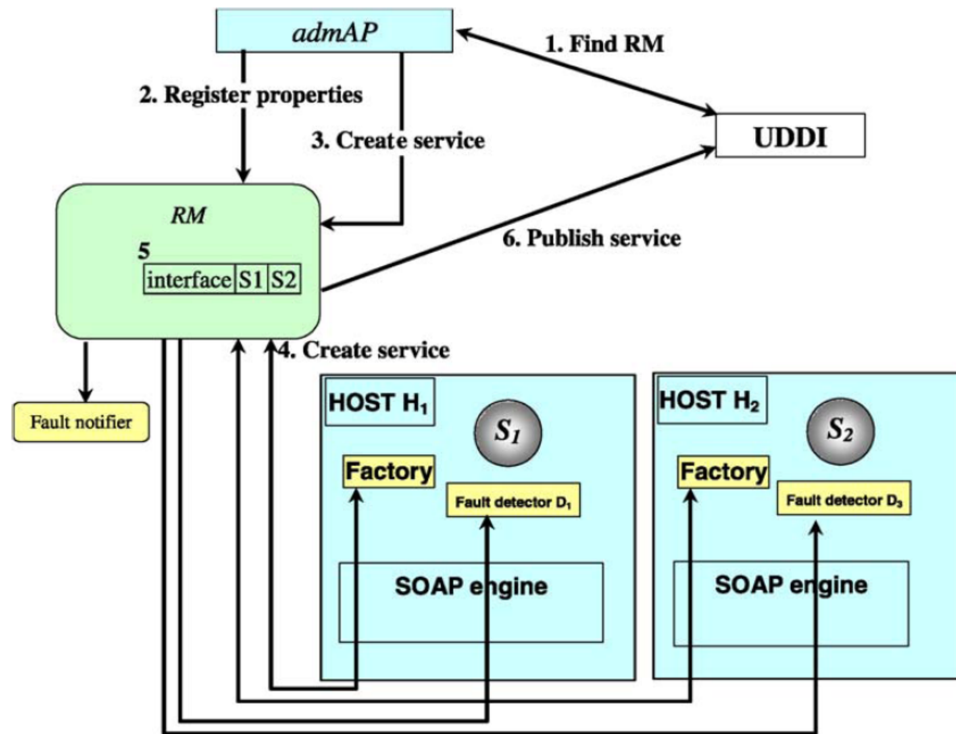



**Figura 4.4:** Grupo de serviços com o FT-SOAP [Fang et al., 2007]

falha das réplicas do grupo. Assim que a falha é identificada, outro componente, o *fault notifier*, é acionado para notificar essa falta. Já o componente de *logging/recovery* tem a responsabilidade de armazenar informações das requisições enviadas por clientes, para que a recuperação de uma réplica faltosa possa ser realizada com base nessas informações.

Na figura 4.4 é possível visualizar os passos necessários para a inicialização do FT-SOAP. Para iniciar o funcionamento do FT-SOAP, o administrador da aplicação (admAP) primeiro deve localizar o componente *replication manager* em um registro UDDI (passo 1). Após localizado, o admAP utiliza o *replication manager* para constituir o grupo de réplicas do serviço. Para realizar essa constituição, o *replication manager* disponibiliza funcionalidades que permitem que o admAP registre as propriedades de TF que serão utilizadas, bem como o número inicial de réplicas do grupo (passo 2). Feito isso, o admAP requisita ao *replication manager* a criação do serviço (passo 3). Para realizar esse processo de criação (passo 4), o *replication manager* automaticamente: (a) requisita que as fábricas presentes nos *hosts* que farão parte do grupo (H1 e H2 na figura 4.3) criem as réplicas com base nas propriedades de TF que foram registradas, (b) ativa seus detectores de falhas para monitorar essas réplicas, (c) registra a notificação de falta que será utiliza nessa grupo no componente *fault notifier* e, (d) define quem será a réplica primária, ficando as demais

réplicas configuradas automaticamente como *backups*. Após isso, as informações da constituição do grupo são salvas no arquivo WSDL que descreve o serviço (passo 5) e esse arquivo é publicado no registro UDDI (passo 6) para que clientes possam enviar requisições ao serviço.

Para acessar o serviço e usufruir das propriedades de TF impostas pelo FT-SOAP, o cliente deve utilizar um componente de software capaz de redirecionar as requisições para outra réplica do serviço em caso de falha da réplica primária. Para conhecer todas as réplicas participantes do serviço, esse componente utiliza as informações contidas no arquivo WSDL, mais precisamente da *tag* <*WSG/*>, que descreve a localização da réplica primária e das demais réplicas do serviço. Essa *tag* é uma extensão proposta pelo FT-SOAP à especificação WSDL. Assim que a falha de alguma das réplicas é detectada, o componente *fault detector* aciona o componente *fault notifier*, que notifica a falha ao componente *replication manager*. O *replication manager* então é responsável por coordenar o processo de recuperação da réplica, acionando os mecanismos necessários para realização dessa tarefa.

## 4.5 WS-Replication: A Framework for Highly Available Web Services

WS-Replication [Salas et al., 2006] é um *framework* que foi desenvolvido para adicionar características de alta disponibilidade para Serviços Web. Os seus autores afirmam que esse *framework* foi inicialmente formulado para replicação de Serviços Web em uma WAN. *WS-Replication* utiliza replicação ativa e usa uma implementação própria, chamada de *WS-Multicast*, para a comunicação entre as réplicas do serviço. O *WS-Multicast* utiliza o SOAP como protocolo de transporte. A comunicação com o *framework* é exclusivamente baseada em SOAP, não forçando a utilização de protocolos alternativos de comunicação que não sejam os padronizados para Serviços Web.

Aplicações clientes invocam operações em serviços replicados com o *WS-Replication* da mesma forma que invocam operações em serviços não replicados, não sendo necessária a utilização de um componente específico de software. A figura 4.5 mostra uma visão geral do *WS-Replication*, evidenciando os passos necessários para o processamento de requisições submetidas por clientes. Nessa figura, o serviço replicado é denominado **ws**. Nota-se que esse serviço é provido por duas réplicas, representadas pelos retângulos maiores, utilizando a técnica de replicação ativa. É possível distinguir a interface WSDL que contém as operações do serviço (representada pelo retângulo com o nome ws), o proxy (representado pela elipse) e a implementação do serviço

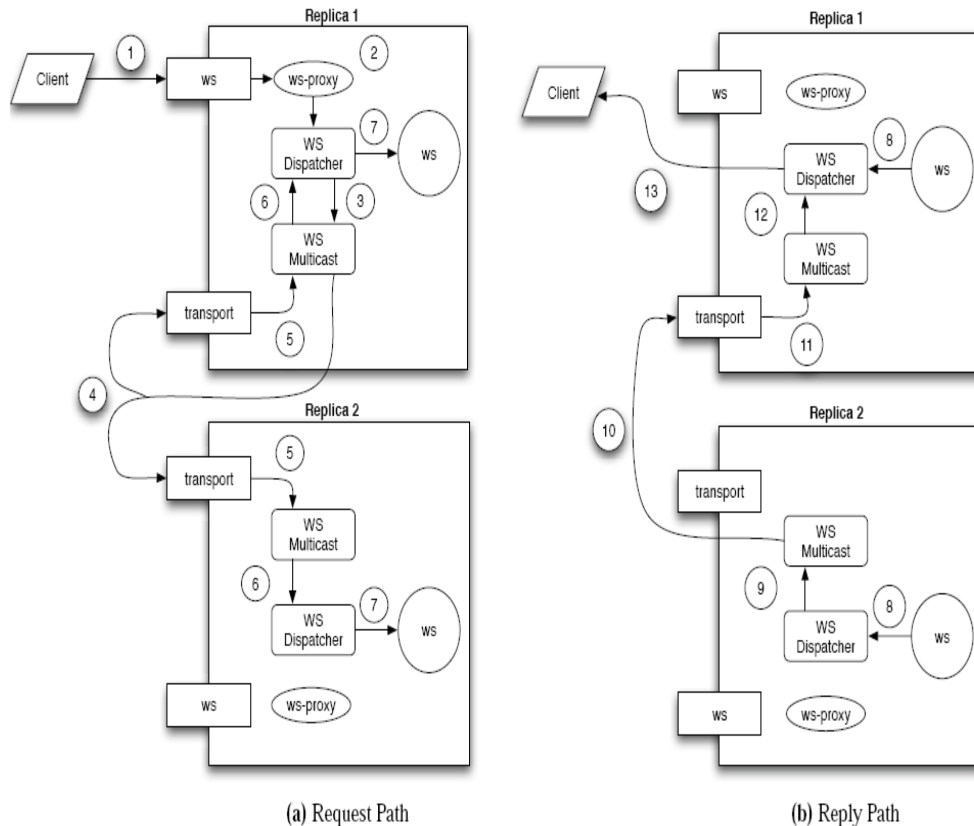



**Figura 4.5:** WS-Replication *Framework* [Salas et al., 2006]

(representada pelo círculo). Além disso, o WS-Replication pode ser segmentado conceitualmente em dois componentes: um componente de replicação e um componente de multicast confiável.

O componente de replicação permite a utilização de replicação ativa como técnica de replicação dos Serviços Web. Esse componente de replicação consiste em um componente de implantação, um gerador de *proxy* e um Serviço Web despachante (*WS-Dispatcher*). O componente de implantação permite que o Serviço Web seja replicado facilmente em diferentes *hosts*, utilizando apenas um único arquivo, similar a um aquivo com extensão .ear ou .jar, utilizados em aplicações Java.

O gerador de *proxy* se encarrega da geração de um *proxy* para cada operação do Serviço Web. Esse *proxy* intercepta as invocações dessa operação e interage com o *WS-Dispatcher* para distribuir essa invocação para as demais réplicas. Já o *WS-Dispatcher* é o componente responsável por intermediar e propagar as invocações encaminhadas pelo *proxy* ao longo das réplicas do serviço. Esse componente tem duas funções principais: (a) transformar a invocação encaminhada pelo *proxy* em uma mensagem *multicast* que será encaminhada para as outras réplicas do serviço, e, (b)

receber a mensagem *multicast* e reconstruir a invocação da operação que deverá ser executada na implementação concreta do serviço. Dependendo da especificação, o *WS-Dispatcher* encaminha a resposta ao cliente assim que receber a primeira, a maioria ou todas as respostas das réplicas do serviço.

Já o componente de *multicast* é responsável por fornecer toda a infra-estrutura de comunicação de grupo do *WS-Replication*. Esse componente é um Serviço Web descrito por um arquivo WSDL que expressa a funcionalidade de *multicast* confiável usando SOAP como protocolo de transporte. Esse componente foi implementado com auxílio do JGroups [JGroups, 2008], sendo que toda a infra-estrutura para a realização de *multicast* confiável é fornecida por esse *framework*.

Ainda na figura 4.5 é possível verificar a seqüência de passos que marcam o envio (figura 4.5a) e a recepção da resposta (figura 4.5b) de uma requisição realizada ao serviço. O passo 1 representa o envio de uma requisição do cliente para o serviço. Essa requisição representa a invocação de alguma das operações presente no Serviço Web replicado. Essa invocação é interceptada pelo *proxy* e encaminhada ao *WS-Dispatcher* (passo 2). O *WS-Dispatcher* encaminha a representação da mensagem *multicast* dessa invocação para o *WS-Multicast* (passo 3). O *WS-Multicast* utiliza SOAP para a disseminação dessa mensagem para todas as demais réplicas do serviço (passo 4). Depois de enviada, todos os componentes*WS-Multicast* recebem a mensagem (passo 5). *WS-Multicast* verifica a ordem da mensagem recebida para garantir as propriedades de ordenação total e confiabilidade definidas pelo *WS-Replication*. Após isso, o *WS-Multicast* encaminha a mensagem para o seu respectivo *WS-Dispatcher* (passo 6). O *WS-Dispatcher* executa a operação requisitada no Serviço Web local (passo 7).

A resposta dessa invocação é retornada para o *WS-Dispatcher* (passo 8). Cada *WS-Dispatcher* das réplicas que não receberam diretamente a requisição do cliente, encaminham a resposta a seus respectivos componentes *WS-Multicast* (passo 9). Cada *WS-Multicast* encaminha a resposta recebida, utilizando uma mensagem *unicast*, para a réplica que recebeu a requisição do cliente. O *WS-Dispatcher* da réplica que recebeu a requisição do cliente, aguarda por um número específico de respostas das réplicas (primeira, maioria, todas). Enquanto as respostas vão sendo recebidas (passo 11), o *WS-Multicast* vai encaminhando essas respostas ao *WS-Dispatcher* (passo 12), que assim que receber número específico de respostas, encaminha a resposta selecionada ao cliente (passo 13).

Como o WS-Replication utiliza replicação ativa, as réplicas faltosas são mascaradas pelas réplicas que ainda permanecem ativas. Caso seja colocado um *proxy* no cliente, esse cliente poderá acessar o serviço enquanto existirem réplicas disponíveis.

| | Middleware | SWS | WS-Replication | FTWEB | FT-SOAP |
|---|---|---|---|---|---|
| **Técnicas de replicação** | Ativa | Ativa | Ativa | Várias | Passiva |
| **Clientes legados** | Não | Não | Sim | Não | Não |
| **Vários grupos do mesmo serviço** | Não | NE | Não | Não | Não |
| **TF configurável** | NE | Sim | NE | Não | Não |
| **Recuperação de componentes** | NE | NE | NE | NE | NE |
| **Detecção de Falhas** | NE | NE | NE | Sim | Sim |
| **Recuperação de Falhas** | NE | Sim | NE | Sim | Sim |
| **Ordenação de mensagens** | Sim | Sim | Sim | Sim | Sim |

**Tabela 4.1:** Comparativo entre as diferentes infra-estruturas e *frameworks* de TF para Serviços Web

## 4.6 Análise Comparativa

Após a descrição das principais características presentes em cada um dos cinco trabalhos citados nesse capítulo, é interessante realizar uma comparação entre esses trabalhos. Para isso foram utilizados critérios que levam em consideração o fornecimento e manutenção da TF em Serviços Web, sendo responsáveis por avaliar os trabalhos em relação à: técnicas de replicação que podem ser utilizadas; suporte a consumidores legados; suporte a formação de diferentes grupos de réplicas do mesmo serviço; capacidade de configuração para fornecimento de TF que suporte qualquer uma das classes de faltas; suporte a recuperação dos componentes da infra-estrutura responsáveis pelo fornecimento de TF para os Serviços Web. A tabela 4.1 descreve a relação entre esses critérios e cada um dos trabalhos relacionados citados nesse capítulo. Essa relação é definida e classificada de acordo com as seguintes convenções: Sim = Suporta, Não = Não suporta; NE = Não especificado.

Analisando a tabela, pode-se verificar que $80\%$ dos trabalhos relacionados utilizam replica-

ção ativa como técnica de replicação para os Serviços Web. Apenas o FT-SOAP utiliza replicação passiva o que permite a essa infra-estrutura somente o suporte a faltas de *crash*. Em relação aos consumidores legados, apenas o *WS-Replication* possui suporte, porém a constituição de sua infra-estrutura não fornece o mesmo grau de TF fornecido ao clientes que utilizem um componente especializado. Já nos outros trabalhos, a utilização de componentes especializados do lado do cliente é crucial para a utilização do serviço.

Com relação ao estilo de formação de grupos de réplicas pode-se notar que quatro dos cinco trabalho não suportam a formação de mais de um grupo de replicação do mesmo serviço. O *SWS* possui interação entre grupos, porém essas interações são exemplificadas entre grupos de réplicas de diferentes serviços. A divisão em diferentes grupos pode, por exemplo, tornar mais simples o algoritmo de ordenação total de mensagens, caso esses grupos possuam um líder. Nesse caso, a definição da ordem é efetuada somente pelos lideres dos grupos. A característica de TF configurável exprime a capacidade das infra-estruturas e *frameworks* de modificarem suas configurações para suporte de grande parte das classes de faltas presentes na literatura.

Como Serviços Web são utilizados para concepção de sistemas dos mais variados tipos, é interessante que as diferentes classes de faltas possam ser suportadas, possibilitando uma adaptação das infra-estruturas e *frameworks* às necessidades de TF de cada tipo de serviço. Nesse sentido, somente o *SWS* possui esse suporte. O *FTWEB* e o *FT-SOAP* suportam somente alguns tipos de faltas e, o *WS-Replication* e o *Middleware* não especificam quais as classes de faltas são suportadas.

Por fim, todos os trabalhos não especificam como seus componentes são recuperados caso ocorra uma falha em algum deles. A recuperação desses componentes é extremamente importante, já que, todas as propriedades de TF fornecidas à Serviços Web são garantidas por eles. Como em grande parte dos trabalhos, o processo de recuperação é abordado somente do ponto de vista do serviço e, diversos componentes não estão acoplados a ele, a manutenção da TF fornecida pela recuperação da falha desses componentes é um tópico importante e que não é mencionado.

## 4.7 Considerações sobre o capítulo

Apesar de propiciarem um aumento na disponibilidade de Serviços Web, esses trabalhos apresentam restrições tais como a incompatibilidade com consumidores legados, a falta de definições sobre a recuperação dos componentes ou a exigência de alterações nos padrões existentes em Serviços Web. Essas restrições dificultam a adoção desses trabalhos já que além da alteração no serviço que é disponibilizado, na maioria dos casos, é necessária a alteração e reestruturação

também no consumidor do serviço. Essa restruturação muitas vezes não é viável, impossibilitando que o serviço possa usufruir das características de TF fornecidas por esses trabalhos.

Sendo assim, o próximo capítulo descreve a arquitetura de TF para Serviços Web proposta nessa dissertação. Essa arquitetura chamada de WSFTA (*Web Services Fault Tolerant Architecture*), busca atingir o mesmo objetivo destes trabalhos relacionados eliminando tais restrições.

# Capítulo 5

# WSFTA - Uma Arquitetura para Tolerância a Faltas em Serviços Web

Tolerar faltas em Serviços Web não é uma tarefa trivial. O fato de Serviços Web possuírem como ambiente de funcionamento uma rede de computadores, trocando mensagens de forma síncrona e/ou assíncrona, torna mais complexa a realização de tal feito. Prova disso são os diversos trabalhos relacionados a esse assunto, citados no capítulo 4, que tem esse objetivo de fornecer meios para que faltas presentes em um Serviço Web não afetem seu funcionamento.

Cada um desses trabalhos propôs uma forma peculiar de solução para TF em Serviços Web. Essas soluções tem como base experiências com outros ambientes, tecnologias, especificações e teorias sobre técnicas de TF presentes na literatura. Logo, o que se vê é um grande número de infra-estruturas e *frameworks* que utilizam, em sua maioria, as mesmas técnicas para fornecer TF para Serviços Web. O que realmente diferencia uma proposta da outra é a forma de aplicação dessas técnicas, resultando em diferentes princípios de funcionamento, protocolos de comunicação específicos, componentes que permitam que consumidores do serviço possam usufruir da TF, além de fornecer desempenho diferenciado em detrimento ao grau de TF fornecido por esses trabalhos.

Porém, todos os trabalhos relacionados (capítulo 4), sem exceção, possuem a descrição e princípio de funcionamento de suas propostas fortemente acoplados com as tecnologias e/ou plataformas de hardware/software utilizadas para implementação de seus componentes. Como Serviços Web podem ser construídos em diferentes linguagens de programação, utilizando diferentes arquiteturas de hardware e software, a dependência em uma determinada tecnologia ou plataforma de hardware/software mostra uma limitação do uso dessas plataformas de TF. Além disso, na maioria desses trabalhos não fica claro se o Serviço Web deve ser implementado utilizando as mesmas

tecnologias e plataformas utilizadas na infra-estrutura ou *framework*. Um exemplo é o FTWEB [Santos et al., 2005], que suporta a invocação de objetos CORBA como se fossem Serviços Web mas não faz nenhuma ressalva dizendo que o desenvolvedor que utilize o FTWEB terá obrigatoriamente que implementar o serviço como um objeto CORBA.

Tomando como base a filosofia da SOA e observando as dependências de tecnologias e plataformas dos trabalhos relacionados, foi desenvolvida uma arquitetura de TF para Serviços Web chamada de WSFTA (*Web Services Fault Tolerant Architecture*). A WSFTA pode ser definida como uma arquitetura que descreve de forma abstrata seus componentes e princípio de funcionamento, permitindo sua implementação e execução em diferentes tecnologias e plataformas de hardware e software. Além disso, a WSFTA se caracteriza pela extração e abstração das experiências descritas nos trabalhos relacionados, objetivando a agregação das melhores características desses trabalhos, de forma a proporcionar uma maior e melhor adaptação às características de TF que cada tipo de serviço necessita.

Portanto, esse capítulo tem o objetivo de descrever todos os aspectos relacionados a essa proposta. Para isso, inicialmente é realizada uma descrição geral da arquitetura e suas principais características. Logo em seguida são abordados os princípios de funcionamento de seus principais componentes. O comportamento de cada um desses componentes é descrito no anexo A com auxílio da linguagem CSP (*Communicating Sequential Processes*) [Hoare, 2004, Roscoe, 2005]. Por fim, são discutidas questões referentes ao armazenamento de estado em Serviços Web e expressadas as considerações finais sobre o capítulo.

## 5.1 Descrição Geral da Arquitetura

A especificação dessa arquitetura de *software* descreve de forma abstrata um conjunto de componentes necessários ao fornecimento de TF para Serviços Web. A segmentação da arquitetura em diferentes componentes proporciona uma separação implícita do problema de fornecimento de TF nesse cenário. Essa separação permite a divisão das responsabilidades e funcionalidades entre esses componentes, sendo cada um deles responsável por um domínio específico desse problema. Além disso, por serem especificados de forma abstrata, esses componentes podem ser implementados em diferentes tecnologias. Essa independência de tecnologia permite que várias versões dos componentes possam ser implementadas e utilizadas em conjunto, mantendo a interoperabilidade entre essas versões. Nesse caso, serviços que utilizem diferentes versões dos componentes da WSFTA, como meio de obtenção de TF, possibilitam que a agregação de características da redun-

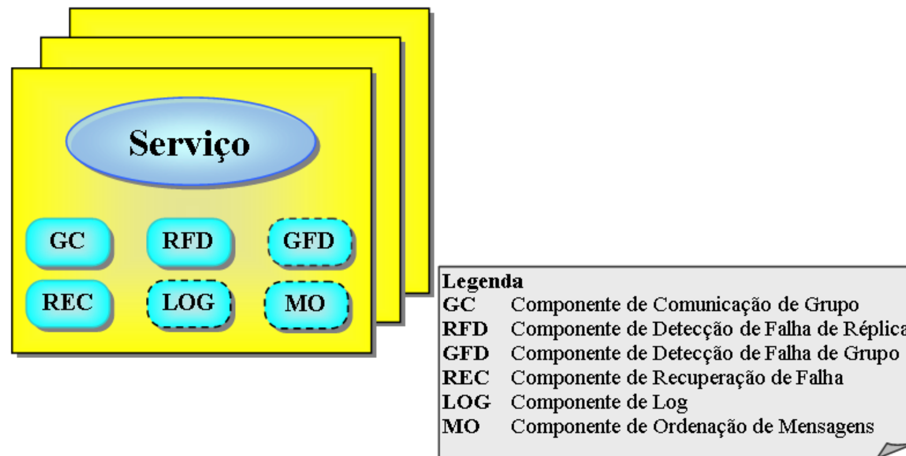



**Figura 5.1:** Representação dos componentes da WSFTA

dância N-modular [Hopkins, 1978] a esses componentes, aumentando assim sua confiabilidade.

Para o entendimento do princípio de funcionamento da WSFTA, é importante que sejam definidos seus componentes e as responsabilidades atribuídas a cada um deles. Sendo assim, primeiramente a figura 5.1 exibe a representação da WSFTA composta por seus componentes. Percebe-se nessa representação exibida pela figura 5.1 que esses componentes estão localizados no mesmo *host* que hospeda o serviço. Essa localização conjunta traz vantagens, que serão expressadas ao longo do texto, principalmente para o processo de detecção e recuperação de falhas. Além desses processos, ordenação de mensagens, comunicação de grupo e armazenamento de *log* são cobertos pelos componentes da WSFTA. Logo, os componentes que permitem essa cobertura são:

- Componente de Comunicação de Grupo (GC): responsável pela abstração do protocolo de transporte utilizado na troca de mensagens entre réplicas do serviço;

- Componente de Ordenação de Mensagens (MO): responsável pela garantia de ordenação das mensagens em todas as réplicas do serviço;

- Componente de Recuperação (REC): responsável pela recuperação de estado de uma réplica faltosa;

- Componente de *Log* (LOG): responsável por armazenar informações sobre as mensagens recebidas e estados das réplicas do serviço;

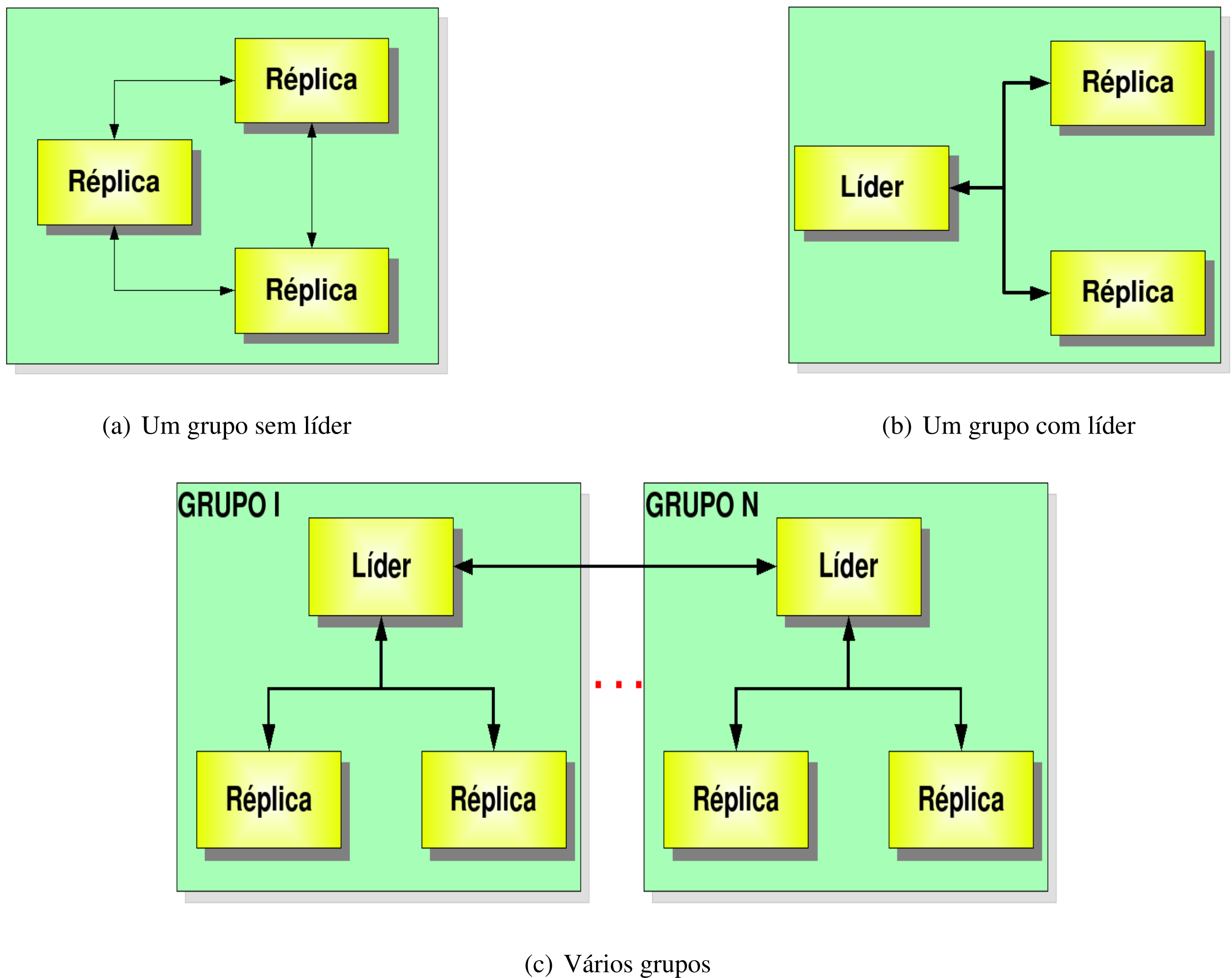
- Componente de Detecção de Falha de Réplica (RFD): responsável pela detecção de falha de uma réplica específica;

- Componente de Detecção de falha de Grupo (GFD): responsável pela detecção de falha de um grupo do serviço replicado;

A definição pura e simples dos componentes da WSFTA não permite o fornecimento de TF se eles forem agregados somente a um único Serviço Web. Logo, na WSFTA a replicação é a técnica chave no fornecimento de TF para Serviços Web. Manter réplicas de um mesmo serviço é a solução utilizada para manter esse serviço em funcionamento mesmo na presença de faltas. Uma das desvantagens de manter réplicas em diferentes locais ao longo da rede é o aumento da complexidade envolvida no gerenciamento e na manutenção da consistência dos dados. Porém, isso pode ser compensado pelo fornecimento de características tais como confiabilidade e disponibilidade ao serviço, já que assim não existe um ponto único de falha. Com isso, *hosts* que hospedem serviços que utilizem os componentes da WSFTA podem ser agrupados objetivando a formação de um grupo de réplicas do mesmo serviço. Aproveitando a flexibilidade e abstração imposta pelos componentes da WSFTA, é possível que esse grupo possa utilizar técnicas de replicação, comunicação de grupo, ordenação de mensagens, detecção e recuperação de falhas, que se adaptem melhor às necessidades de TF do serviço.

Para satisfazer essas necessidades, a WSFTA impõe algumas restrições. A primeira restrição especifica que um grupo de réplicas deve ser a menor unidade de formação presente na WSFTA, ou seja, réplicas do serviço devem obrigatoriamente formar pelo menos um grupo. Na figura 5.2 são exibidas as possíveis formações que podem ser obtidas com a WSFTA. Nota-se que os grupos podem ser de dois tipos: grupos hierárquicos (contendo uma réplica líder) e grupos não-hierárquicos. A segunda restrição especifica que, ao serem formados vários grupos do mesmo serviço, esses grupos devem ser obrigatoriamente grupos hierárquicos. Nesse caso, a terceira restrição especifica que caso um consumidor esteja enviando requisições para o serviço, a transparência de falha de grupos somente será total se esse consumidor estiver utilizando um componente opcional chamado de *WSFTA-Proxy*. Como o *WSFTA Proxy* conhece o endereço de todas as réplicas "visíveis", ou seja, o endereço de todos os líderes dos grupos, em caso de falha do grupo que recebeu a requisição, o *WSFTA-Proxy* reenvia essa requisição para outro grupo ativo. Para consumidores legados, requisições são enviadas somente para um dos líderes dos grupos ativos. Logo a falha desse grupo não é transparente para esse tipo de cliente.

Além de ser a réplica "visível", o líder tem a responsabilidade de encaminhar a requisição para as outras réplicas do grupo. Dependendo da técnica de replicação utilizada, a réplica líder precisa receber a resposta da requisição encaminhada, realizar a votação da resposta que será en-




(a) Um grupo sem líder

(b) Um grupo com líder

(c) Vários grupos

**Figura 5.2:** Diferentes formações suportadas pela WSFTA

viada para o consumidor e efetivamente enviar a resposta escolhida. A quarta restrição, já citada anteriormente, define que somente o líder de cada grupo pode receber requisições de consumidores legados. Na existência de mais de um grupo, o WSFTA-Proxy só poderá enviar requisições aos líderes, fornecendo também transparência do número total de réplicas do serviço. Para consumidores legados, o acesso ao serviço possuindo um ou mais grupos fica transparente, pois somente um dos líderes recebe as requisições desses consumidores. Isso permite que o número total de grupos e réplicas do serviço seja transparente para esses consumidores.

A formação de diferentes grupos, permite uma maior transparência de falha das réplicas do serviço. Em caso de falha de um grupo inteiro, consumidores legados que realizavam requisições para esse dos grupo, não tem transparência de falha. Para tolerar falhas de grupo consumidores podem ser implementados com auxílio do WSFTA-Proxy.

Nota-se na figura 5.1 que cada réplica possui os componentes GC, RFD, LOG, REC, MO

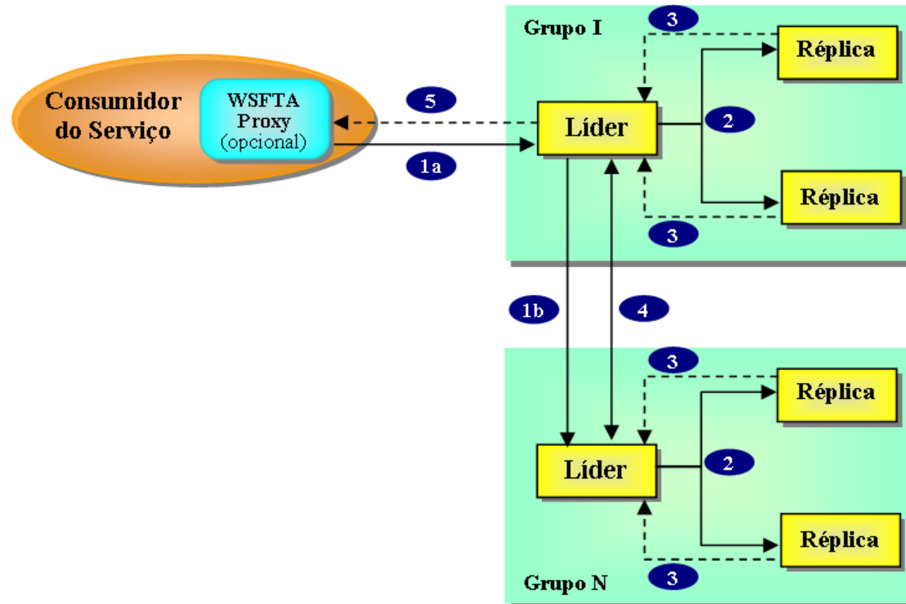



**Figura 5.3:** Réplicas organizadas em dois grupos

e GFD. Porém, nem todos os componentes são habilitados em todas as réplicas do grupo. Esses componentes, que podem não estar habilitados, são representados pelo contorno tracejado. Por exemplo, o componente GFD somente é habilitado no líder do grupo. Esse componente é habilitado quando existe mais de um grupo de réplicas do serviço. Sua responsabilidade é detectar a falha de outros grupos de réplicas, sendo necessária sua execução somente quando realmente existirem diversos grupos ativos. Outro exemplo seria a não utilização do componente de LOG juntamente com um serviço sem estado. Se o serviço não necessita de armazenamento de informações sobre as requisições, o LOG pode ser desabilitado, e o serviço recuperado somente com a reinicialização do mesmo. A decisão de habilitar/desabilitar um determinado componente depende diretamente das propriedades de TF impostas pelo serviço.

Para facilitar o entendimento do processo de comunicação de réplicas do serviço que utilizem a WSFTA, a figura 5.3 exibe um cenário que descreve a seqüência de ações necessárias para utilização do serviço. Esse serviço é composto por seis réplicas organizadas em dois grupos. O primeiro passo é realizar o envio da requisição para um dos líderes dos grupos (Passo 1a). Após receber a mensagem, esse líder encaminha a requisição para os demais lideres (Passo 1b). Toda comunicação realizada entre as réplicas utiliza o componente GC. O próximo passo é a definição da ordem de execução das requisições recebidas. Essa tarefa é realizada pelos líderes dos grupos. O componente MO permite a utilização de diversos algoritmos de ordenação. Como os passos para o

processo de ordenação das requisições dependem do algoritmo de ordenação escolhido, esses passos não foram representados na figura. Após definir a ordem de execução das requisições, todos os líderes encaminham essas requisições (na mesma ordem) para as réplicas de seus grupos (Passo 2). Caso a técnica de replicação utilizada em um grupo exija que todas as réplicas processem a requisição, o líder do grupo utiliza seu componente GC para receber as respostas das réplicas do seu grupo (Passo 3). Após isso, os líderes dos grupos elegem a resposta que será entregue, caso necessário, ao consumidor do serviço (Passo 4). Os número de passos e comunicações necessárias para realizar essa eleição depende diretamente do algoritmo de ordenação utilizado. Após isso, esse líder realiza uma votação e elege a resposta que será enviada para o consumidor legado. Em uma interação *RPC-Style*, somente um dos líderes entrega a resposta ao consumidor do serviço. (Passo 5).

É importante salientar que consumidores que utilizam o WSFTA-Proxy usufruem da transparência de falha de grupo, sendo que, em caso de falha de grupo, esse componente automaticamente reenvia a mensagem para outro líder disponível. Com auxílio de seu componente LOG, esse líder verifica se a mensagem já foi executada. Em caso positivo, o líder somente reenvia a resposta para o WSFTA-Proxy. Em caso negativo, a mensagem é encaminhada para as demais réplicas do grupo, ordenada e, após sua execução e votação, a resposta é enviada para o WSFTA-Proxy.

Apesar da descrição de um cenário que exibe o passos necessários para utilização de um serviço com a WSFTA, é importante abordar os princípios de funcionamento e as principais características dos componentes presentes nessa arquitetura. Para isso, as próximas seções descrevem cada um dos componentes presentes na WSFTA.

## 5.2 Componente de Comunicação de Grupo - CG

O componente de comunicação de grupo (CG) atua na base de comunicação das réplicas que utilizam a WSFTA. Sua tarefa é fornecer meios para a troca de mensagens entre as réplicas do serviço, realizando uma abstração do protocolo de comunicação de grupo utilizado para essa troca de mensagens. A figura 5.4 exibe a abstração oferecida pelo componente GC.

Pode-se notar na figura 5.4 que esse componente é quem realiza a mediação de toda a comunicação entre as réplicas do serviço. Isso permite que diferentes protocolos possam ser utilizados, possibilitando a escolha do protocolo que melhor se adapte à rede de comunicação na qual as réplicas estão alocadas. Por ser um componente de comunicação de grupo, o GC também é responsável por armazenar o *membership* do grupo. Conforme réplicas vão entrando e saindo de um determi-

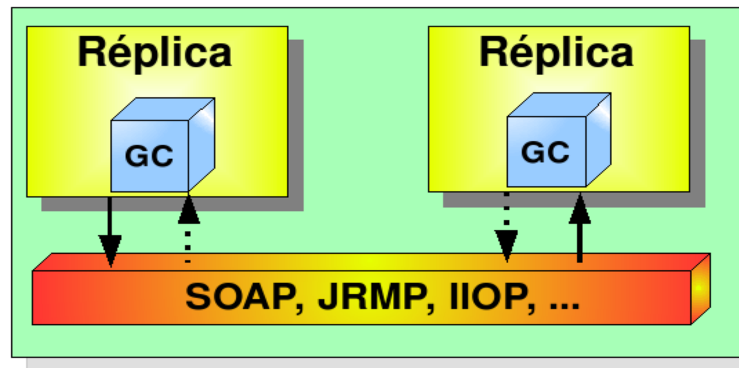



**Figura 5.4:** Abstração oferecida pelo componente GC

nado grupo, o *membership* é atualizado em todas as réplicas, fazendo com que todas tenham a mesma visão do grupo de réplicas em um determinado instante de tempo.

Quando são formados vários grupos, seus líderes precisam ter conhecimento da localização uns dos outros, para que mensagens possam ser trocadas entre eles. Essa tarefa também é de responsabilidade do GC, que mantém, similar ao que é realizado dentro de um grupo, um *membership* que contém referências para todos os líderes ativos. Se essa troca de mensagens e armazenamento de *membership* dos líderes for examinada mais detalhadamente, pode-se dizer que esses líderes formam um grupo virtual, onde o principal objetivo da formação desse grupo é a manutenção da consistência do funcionamento dos diferentes grupos. Além disso, o protocolo de comunicação utilizado para troca de mensagens entre os líderes pode ser diferente do protocolo utilizado para troca de mensagens dentro de um grupo. Isso permite que, por exemplo, líderes troquem mensagens entre si utilizando o protocolo SOAP e encaminhem essas mesmas mensagens para as réplicas do seu grupo utilizando um outro protocolo tal como *IIOP*. Para isso, cada réplica mantém duas instâncias do componente GC para gerenciamento do grupo virtual de líderes e do grupo da qual a réplica faz parte. Qualquer mudança nesse grupo virtual é notifica para todas as réplicas para que todas estejam aptas a assumir a liderança de seu grupo em casa de falha do líder atual.

Adicionalmente, o componente GC permite a utilização de implementações de difusão confiável, garantindo assim a confiabilidade da comunicação de grupo realizada pelas réplicas. Nota-se que esse componente é extremamente importante para a WSFTA por permitir a troca de mensagens entre as réplicas do serviço. Logo, é com seu auxílio que as mensagens marcadas pelo componente MO são encaminhadas às outras réplicas do serviço, permitindo que o processo de ordenação possa ocorrer tranqüilamente. Esse processo de ordenação é descrito na próxima seção

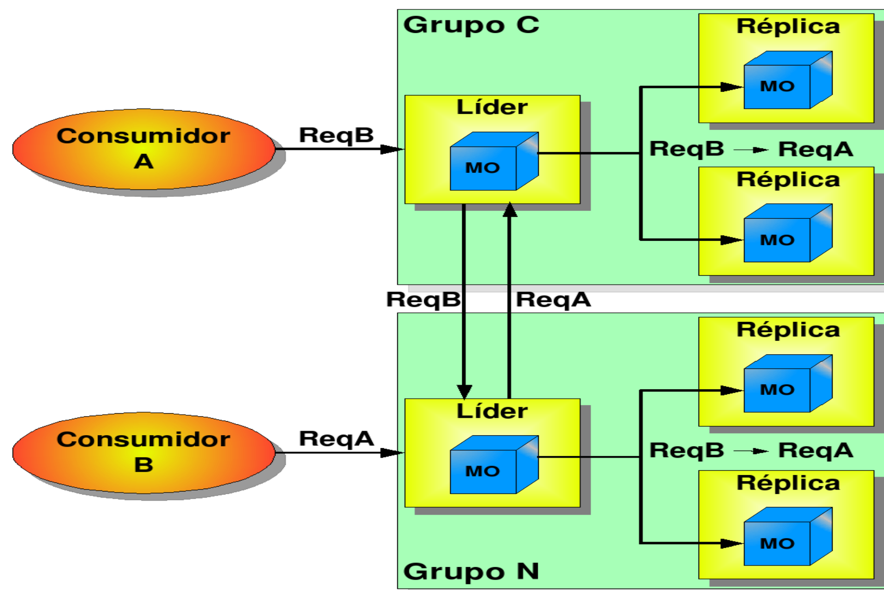



**Figura 5.5:** Funcionamento conceitual do componente MO com réplicas distribuídas em dois grupos

que trata dos aspectos envolvidos em torno do componente MO.

## 5.3 Componente de Ordenação de Mensagens - MO

Para manter a consistência de estado entre todas as réplicas do serviço é necessário que todas as mensagens recebidas sejam executadas na mesma ordem em todas as réplicas. Obviamente, essa garantia de ordenação precisa ser satisfeita em dois tipos de serviços: (a) serviços do tipo *stateful*, isto é, serviços que possuem armazenamento de estado entre mensagens enviadas por consumidores do serviço, ou, (b) serviços do tipo *stateless* (sem estado) que interagem com outros softwares ou serviços que possuem estado. Para fornecer essa garantia de ordenação, a arquitetura proposta possui um componente para ordenação de mensagens chamado de MO. Esse componente tem a tarefa de realizar a marcação e ordenação das mensagens recebidas, certificando que todas as réplicas executem as mensagens de maneira ordenada, seja para processamento imediato ou posterior. Essa ordenação tem o intuito de fazer com que todas as réplicas ativas cheguem a um mesmo estado, ou seja, estejam em um estado consistente. A figura 5.5 demonstra conceitualmente a base de funcionamento da ordenação de mensagens em um exemplo que utiliza dois grupos de réplicas.

No exemplo ilustrado pela figura 5.5, dois consumidores (A e B) estão enviando mensagens

para o serviço replicado em dois grupos. Após receber as mensagens, o líder do grupo C encaminha a mensagem *ReqA* para o líder do grupo N e o líder do grupo N encaminha a mensagem *ReqB* para o líder do grupo C. Após estarem de posse das mensagens *ReqA* e *ReqB*, os componentes MO de cada um dos líderes iniciam o processo responsável por definir a ordem de execução das mensagens *ReqA* e *ReqB*. A definição efetiva da ordem de execução dessas mensagens vai depender do algoritmo de ordenação utilizado. Após a execução desse algoritmo, pode-se visualizar na figura 5.5 que a ordenação das mensagens enviadas pelos consumidores resulta na fila de entrega ordenada 'ReqB→ReqA'. As mensagens ordenadas são encaminhadas pelos líderes para as outras réplicas de cada um dos grupos. Caso as mensagens sejam recebidas fora de ordem, o componente MO dessas réplicas armazena a mensagem em um *buffer* até que a ordem de execução dessa mensagem seja alcançada. Esse processo resulta conceitualmente na ordenação total das mensagens em todas as réplicas de ambos os grupos.

Além da ordenação de mensagens, que tem como principal objetivo manter a consistência de estado entre todas as réplicas, é importante que as falhas que podem vir a ocorrer nessas réplicas sejam detectadas corretamente. Para isso, a próxima seção apresenta os componentes de detecção de falha presentes na WSFTA. Esses componentes auxiliam na detecção da falha do serviço que influencia diretamente na manutenção do grau de TF fornecido ao serviço.

## 5.4 Detecção de Falha

Durante a interação entre cliente e serviço podem ocorrer problemas que impossibilitem o processamento de mensagens em uma ou mais réplicas do serviço. Se for feita uma análise mais criteriosa, durante qualquer instante de tempo, independentemente de o serviço estar recebendo mensagens dos consumidores ou não, pode-se ter algum tipo de problema que leve ao mal funcionamento ou à parada de uma das réplicas presentes. Para identificar problemas ocorridos nessas réplicas são utilizados detectores de falhas. Os detectores de falhas ficam responsáveis pelo monitoramento das réplicas ativas, realizando essa tarefa periodicamente. Essas informações são importantíssimas para que mecanismos de recuperação possam restaurar o estado de um membro faltoso.

Na WSFTA pode ser formado mais de um grupo de réplicas do mesmo serviço. Logo, além da detecção da falha de uma dessas réplicas é necessário detectar a falha de um grupo inteiro. Para isso, o processo de detecção de falha foi dividido em dois componentes: o Detector de Falha de réplica (RFD) e o Detector de Falha de Grupo (GFD). Detalhes sobre cada um desses componentes

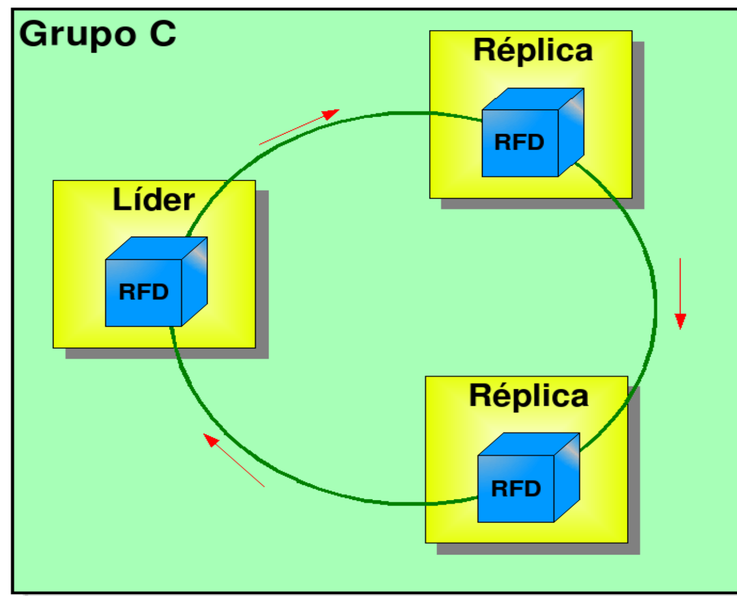



**Figura 5.6:** RFDs formando um anel virtual de monitoramento em grupo com líder

podem ser visualizados a seguir.

### 5.4.1 Componente de Detecção de Falha de Réplica - RFD

Os RFDs são responsáveis pelo monitoramento das réplicas ativas alocadas em um grupo específico. Cada réplica possui um RFD com a responsabilidade de monitorar periodicamente seu estado. O tempo gasto para cada ciclo de monitoramento vai depender do tempo de monitoramento especificado. Esse tempo define o período no qual o RFD precisa checar a atividade da réplica monitorada.

Os RFDs são úteis na detecção de falhas internas nos serviços e de falhas de software na plataforma de execução. Juntamente com o RFD, a utilização de sistemas heterogêneos e de múltiplas versões [Xu and Bruck, 1998] de um mesmo serviço pode garantir que uma mesma falha não ocorrerá simultaneamente em todas as réplicas. Dentro do processo de detecção de falhas ocorrem duas situações que necessitam de tratamento diferenciado. Quando grupos são formados é importante diferenciar a detecção de falha dos líderes e das demais réplicas que pertencem ao grupo. Quando a falha de uma dessas réplicas se manifesta, a primeira ação a ser tomada é a exclusão dessa réplica do grupo. Paralelamente a esse processo, é necessário o acionamento do mecanismo de recuperação dessa réplica, para que seja possível restabelecer seu estado e possibilitar o retorno posterior dessa réplica ao grupo o mais rápido possível.

Porém, quando um líder falha, a primeira ação tomada deve ser o disparo de um processo de eleição com a responsabilidade de escolher uma das outras réplicas para assumir o lugar de líder do grupo. Após o término do processo de eleição, a réplica eleita utiliza seu CG para notificar os outros líderes dos demais grupos que ela é a nova representante desse grupo. Em paralelo ao processo de eleição, assim como no caso anterior, o mecanismo de recuperação é acionado para recuperar o estado da réplica faltosa.

Como o RFD fica localizado junto à réplica, um problema de funcionamento do *host* paralisa o funcionamento tanto do serviço quanto do RFD. A paralisação do RFD não permite que a falha do serviço seja reportada às outras réplicas do grupo. Logo, é importante que os RFDs do grupo possuam alguma forma de monitoramento que possibilite detectar a falha de RFDs sem inundar a rede com mensagens destinadas a essa tarefa. Conseqüentemente, fazer com que cada RFD monitore todos os outros RFDs do grupo, representando uma interação "muitos para muitos", não se torna adequada. O processo de detecção aconteceria perfeitamente, porém a rede seria inundada por mensagens trocadas simultaneamente entre todos os RFDs do grupo.

Uma das formas de representação de um grupo de elementos é a formação de um anel, onde cada elemento constitui um pedaço desse anel. Esses elementos são circundados por outros elementos, que podem ser classificados como seus vizinhos. Se cada elemento ficar responsável por monitorar um de seus vizinhos pode-se dizer que esse anel é um anel de confiança, pois a retirada de qualquer elemento do anel será identificada. Utilizando esse mesmo princípio, RFDs formam um anel virtual de confiança para monitorar suas possíveis falhas. A simples falha de um dos RFDs cria uma lacuna no anel que é percebida pelo vizinho responsável por monitorar o elemento faltoso. Assim que a falha é identificada, os mesmos procedimentos realizados no processo de detecção de falha do serviço são disparados. Em paralelo, o anel virtual é reconstituído, fechando a lacuna aberta pela réplica faltosa que deixou o grupo. A figura 5.6 exemplifica conceitualmente esse processo.

### 5.4.2 Componente de Detecção de Falha de Grupo - GFD

GFDs têm a responsabilidade de detectar a falha de um grupo inteiro de réplicas. Esse componente somente é ativado quando existe mais de um grupo do mesmo serviço. Pode-se observar na figura 5.2 que em formações que contém mais de um grupo, os líderes desses grupos realizam a interconexão entre todas as réplicas do serviço. Sendo assim, o processo de detecção de falhas inter-grupos tem como principal objetivo a detecção da falha dos líderes que representam cada um

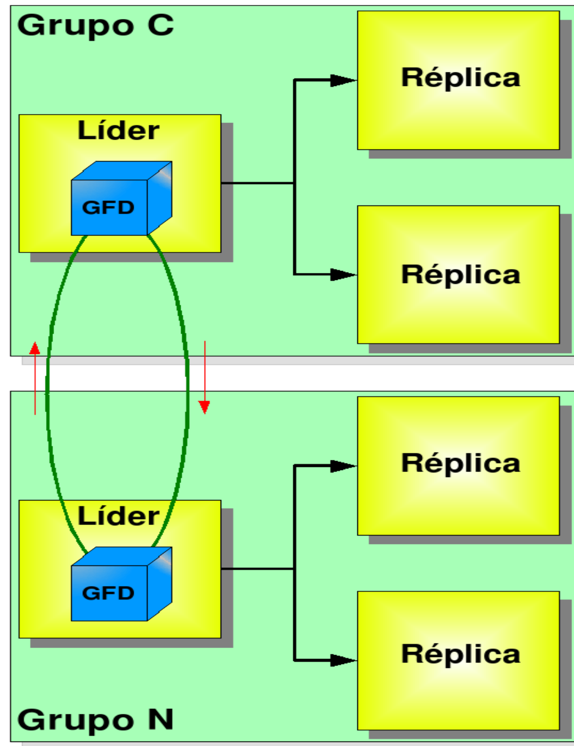



**Figura 5.7:** Anel virtual de detecção de falha dos GFDs

dos grupos.

Após a detecção da falha desse líder, o grupo ao qual ele pertence se torna inacessível; logo, sua falha é extremamente crítica e deve ser tratada. A figura 5.7 fornece a visualização da forma de detecção de falha de grupos. Observa-se na figura 5.7 que a detecção de falha de grupos é baseada em um anel virtual assim como a detecção de falha de RFDs dentro de um grupo específico. Uma das diferenças desse processo de detecção é a necessidade de utilização de protocolos de comunicação adequados para ambientes heterogêneos, caso grupos estejam localizados em redes distintas. O SOAP é um exemplo desse tipo de protocolo.

Como mencionado anteriormente, a falha de um grupo é detectada tomando como base a falha do líder que representa o grupo. Relembrando os conceitos apresentados para RFDs, percebe-se que a falha do líder pode ser detectada tanto por um RFD quanto por um GFD. Quando a falha é detectada pelo RFD, existe a possibilidade da falha de um grupo se tornar transparente para os demais grupos ativos. Essa transparência é alcançada com auxílio da nova réplica, que substitui o líder faltoso e notifica todos os outros líderes sobre a mudança ocorrida no grupo.

Já quando a falha é detectada pelo GFD, é assumido que o grupo falhou e não receberá mais mensagens, ficando inativo até que uma nova réplica assuma a liderança do grupo e notifique os

demais líderes que o grupo está ativo novamente. Porém, nesse caso, antes de voltar ao funcionamento normal, o grupo deve sincronizar seu estado com o estado dos outros grupos, caso seja necessário. Se a falha detectada por GFDs for originada de uma falha física da rede de comunicação na qual o grupo está alocado, será necessário o acionamento de mecanismos que possibilitem a criação, inicialização e integração de um novo grupo de réplicas ao serviço.

Falhas na comunicação no domínio no qual um grupo está situado podem deixá-lo inativo por um tempo indeterminado, sendo necessário o procedimento de substituição desse grupo por outro, em outro domínio, de modo a manter o nível de tolerância a faltas fornecido pela WSFTA. Esse procedimento é realizado pela arquitetura sem afetar o funcionamento dos demais grupos, que continuam a atender requisições normalmente.

## 5.5 Componentes de LOG (LOG) e Recuperação de Falhas (REC)

Durante todo o período de execução do serviço, o mesmo está suscetível a falhas que impossibilitem sua execução normal. Assim que essa falha é detectada, é de suma importância a inicialização de um processo que permita a recuperação dessa falha. Em serviços do tipo *stateful*, o processo de recuperação deve utilizar informações armazenadas anteriormente para que seja realmente possível restabelecer o estado normal de funcionamento do serviço. Esse armazenamento de informações é de responsabilidade do componente de LOG.

O componente de LOG permite que informações trocadas entre e com as réplicas do serviço sejam armazenadas para uma possível utilização posterior. As informações armazenadas pelo componente de LOG podem ser utilizadas para: (a) recuperar o estado de uma réplica faltosa, (b) identificar quais foram as últimas $n$ mensagens submetidas ao serviço, entre outras atribuições. Na WSFTA os *logs* são utilizados principalmente pelo processo de recuperação de estado de um serviço faltoso.

Arquiteturas tolerantes a faltas para sistemas distribuídos necessitam de mecanismos que permitam que os componentes da arquitetura e do serviço possam ser recuperados após suas falhas. Sem esse tipo de mecanismo, essas arquiteturas não conseguem manter o grau de suporte a faltas em um nível aceitável para uma determinada aplicação e/ou situação.

O mecanismo de recuperação depende diretamente do estágio de detecção de falhas. Ou seja, assim que é identificada a falha de uma réplica do serviço, a recuperação dessa réplica deve ser iniciada o mais rápido possível, objetivando sua reativação e reintegração ao serviço. Caso não seja possível realizar essa recuperação, técnicas para criação e inicialização de uma nova réplica,

que substituirá a réplica faltosa, devem ser ativadas. O componente REC tem exatamente essa responsabilidade. Esse componente permite a recuperação de réplicas faltosas de serviços que utilizem a WSFTA. Como a WSFTA também utiliza o conceito de grupos, o componente REC também permite a recuperação de um grupo inteiro de réplicas.

Para realizar essa tarefa, o componente REC conta com a interação direta com os componentes de detecção de falha (RFD e GFD) e com o componente de log (LOG). Os componentes de detecção de falha são os responsáveis por disparar o início do processo de recuperação tanto de uma réplica quanto de um grupo de serviços. Sem esses componentes seria muito complexo determinar o instante exato de tempo que o componente REC deve ser executado. Além desses componentes, o componente de LOG armazena informações fundamentais para a recuperação de falhas de serviços do tipo *stateful*.

## 5.6 Manutenção e armazenamento de estado

Durante todo o capítulo que descreve a WSFTA foi comentada a diferença entre serviços *stateless* e *stateful*. Para contextualizar algumas características e exceções na utilização de arquiteturas de TF nesses tipos de serviços, é importante recapitular o significado de cada uma dessas classificações. Serviços do tipo *stateless* são serviços que não armazenam estado entre mensagens recebidas. Já serviços *stateful* são serviços que armazenam estado entre essas mensagens. Juntamente com essas definições é importante resgatar a definição de serviço segundo a SOA. Um serviço pode ser definido como um artefato de software com funções bem definidas, auto contidas e que não depende do contexto ou estado de outros serviços [Erl, 2005].

Pode-se notar nessas definições que a diferença entre serviços do tipo *stateless* e *stateful* é o armazenamento de estado entre as mensagens. Logo, serviços *stateful* têm a necessidade de manter a consistência de seu estado, independente do número de consumidores e mensagens que são enviadas para o serviço. Se for levado em consideração que, em sua maioria, serviços TF são replicados, as técnicas que auxiliam diretamente essa manutenção de estado em todas as réplicas são a difusão confiável e a ordenação de mensagens. Em conjunto, técnicas tais como armazenamento de *log*, detecção e recuperação de falhas, auxiliam na manutenção da TF fornecida ao serviço.

Porém, segundo [Erl, 2005], serviços que são modelados e implementados com base na filosofia SOA, devem ser serviços *stateless*. [Erl, 2005] afirma que serviços *stateless* promovem maior escalabilidade e reusabilidade de serviços pois não precisam armazenar estado entre men-

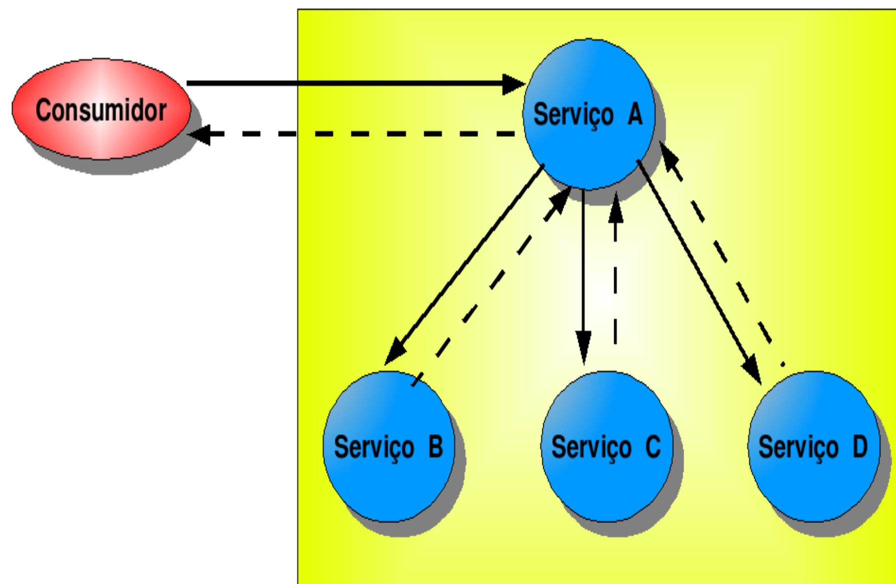



**Figura 5.8:** Serviço A invocando funcionalidades de outros serviços

sagens, permitindo que diversos consumidores submetam suas mensagens sem a necessidade de controlar o momento que essas mensagens devem ser executadas para que o estado do serviço possa se manter consistente. Para esse tipo de serviço, a princípio, ordenação de mensagens e *logs* não são extremamente necessários para o fornecimento de TF. Nesse caso, a promoção de formas de comunicação de grupo, detecção e recuperação de falhas para as réplicas do serviço, cobre satisfatoriamente suas necessidades de TF.

Apesar de serviços *stateless* não realizarem a manutenção de estado entre as mensagens enviadas por seus consumidores, alguns desses serviços estendem softwares legados e se transformam em uma camada, sem estado, utilizada para acessar as funcionalidades desses softwares. Apesar do serviço não manter estado, o software que ele representa possivelmente deve manter. Se as operações disponibilizadas por esse serviço invocarem operações no software legado que alterem seu estado, apesar da classificação de serviço *stateless*, esse serviço passa a ter algumas características de um serviço *stateful*. Uma dessas características é a necessidade de ordenar a execução de suas mensagens. Sem essa ordenação, o serviço pode tornar os dados do software legado inconsistente. Portanto, mesmo não mantendo estado, serviços *stateless* que representam softwares legados (que mantem estado), disponibilizando suas funcionalidades na forma de Serviços Web, precisam ser tratados parcialmente, por arquiteturas de TF, como serviços *stateful*.

Outra questão interessante é a composição de Serviços Web. Os princípios da SOA, por na-

tureza, permitem que vários serviços troquem informações de forma a atingir um objetivo comum. Esse objetivo pode ser, por exemplo, um processo de negócio, composto por diferentes Serviços Web que realizam, cada um, uma tarefa que é parte desse processo. Observando a possível natureza dos serviços (*stateful* e *stateless*), pode-se ter serviços em que suas funcionalidades invocam diversos outros serviços, assim como o serviço representado pela figura 5.8. Nesse caso, fornecer TF para esse tipo de serviço não é uma tarefa trivial. Na figura 5.8, uma mensagem enviada por um consumidor ao serviço A, requer que esse serviço utilize funcionalidades de três outros serviços. Se for incorporada algum mecanismo de TF a esse serviço, juntamente com essa incorporação, dependendo do mecanismo, seriam adicionados problemas em sua execução.

Normalmente, características de TF são fornecidas com base na replicação do serviço. Se a técnica de replicação ativa for utilizada para essa propósito, o número de invocações da mesma funcionalidade nos serviços B, C e D seria igual ao número de réplicas do serviço A. Só esse fator já poderia causar um sobrecarga nos serviços B, C e D. Além disso, se por exemplo, o serviço B for um serviço *stateful*, o fato de todas as réplicas do serviço A invocarem uma funcionalidade do serviço B, tornaria o mesmo, além de sobrecarregado, inconsistente. Logo, para solucionar esse problema, ou utilizasse outra técnica de replicação, por exemplo replicação passiva, ou um mecanismo precisa ser desenvolvido para realizar a interação entre o serviço replicado e os serviços utilizados por ele.

Relembrando que a WSFTA permite a utilização de diferentes formações de grupo que podem utilizar diferentes técnicas de replicação, é possível afirmar que essa arquitetura pode fornecer TF a qualquer tipo de serviço, seja ele *stateful*, *stateless*, composto por outros serviços ou não. Porém, é importante salientar que para cada tipo de serviço devem ser escolhidos os protocolos, técnicas e formação de grupo que melhor se adaptem no fornecimento correto de TF ao serviço.

## 5.7 Considerações sobre o capítulo

Foi visto nesse capítulo que a WSFTA é uma arquitetura que pode ser adaptada às necessidades de TF de cada tipo de serviço. Suas características e princípios de funcionamento de seus componentes permitem que, diferentemente dos trabalhos relacionados descritos no capítulo 4, essa proposta se molde às necessidades do serviço e não o contrário. A descrição formal de cada um dos componentes possibilita que implementações realizadas em diferentes tecnologias possam interagir entre si, desde que sigam corretamente o comportamento modelado.

Nesse sentido, para a realização de testes que permitam avaliar a arquitetura proposta, é

necessária a implementação dos seus diversos componentes. Para isso, o próximo capítulo apresenta uma implementação de referência da arquitetura proposta. Essa implementação de referência se caracteriza pela elaboração de protótipos dos componentes citados anteriormente, permitindo assim que seja verificado o seu funcionamento em um cenário real de utilização.

# Capítulo 6

# Implementação, Experimentos e Análise de Resultados

Para validar a descrição dos componentes da WSFTA , foi implementado um protótipo dos componentes descritos no capítulo 5. Essa implementação foi realizada na linguagem Java utilizando a versão 6.0 do *Java Development Kit* (JDK) [SUN, 2006]. Sendo assim, toda a descrição dos componentes da arquitetura foi implementada na forma de classes Java, utilizando diversas APIs (*Application Programming Interface*) fornecidas pela linguagem. Para o processamento e manipulação de mensagens SOAP foi utilizado o *framework* para desenvolvimento de Serviços Web JAX-WS [Sun, 2006]. Esse *framework*, além de suas APIs, fornece um servidor Web leve que foi utilizado para processar requisições SOAP sobre o protocolo HTTP (*HyperText Transfer Protocol*). Desse modo, não se fez necessária a utilização de um servidor de aplicação para dar suporte à execução de Serviços Web.

Além disso, foram utilizados manipuladores *(handlers)* que possibilitam a inserção transparente dos componentes da arquitetura nos Serviços Web desenvolvidos. Com isso, primeiramente este capítulo descreve a implementação dos componentes da WSFTA, suas características e alguns algoritmos que esboçam parte do funcionamento desses componentes. No anexo B pode-se ter uma visão de todas as classes que fazem parte dessa implementação. Em seguida são descritos experimentos que tem o objetivo de avaliar o impacto de dos componentes da WSFTA no tempo de resposta do serviço. Para isso, diferentes configurações de grupo e técnicas de replicação foram utilizadas, possibilitando a realização de uma análise que indique algumas particularidades inseridas na implementação dos componentes da arquitetura. Por fim, na parte final do capítulo, são realizadas comparações da WSFTA com seus trabalhos relacionados e expressas algumas conclu-

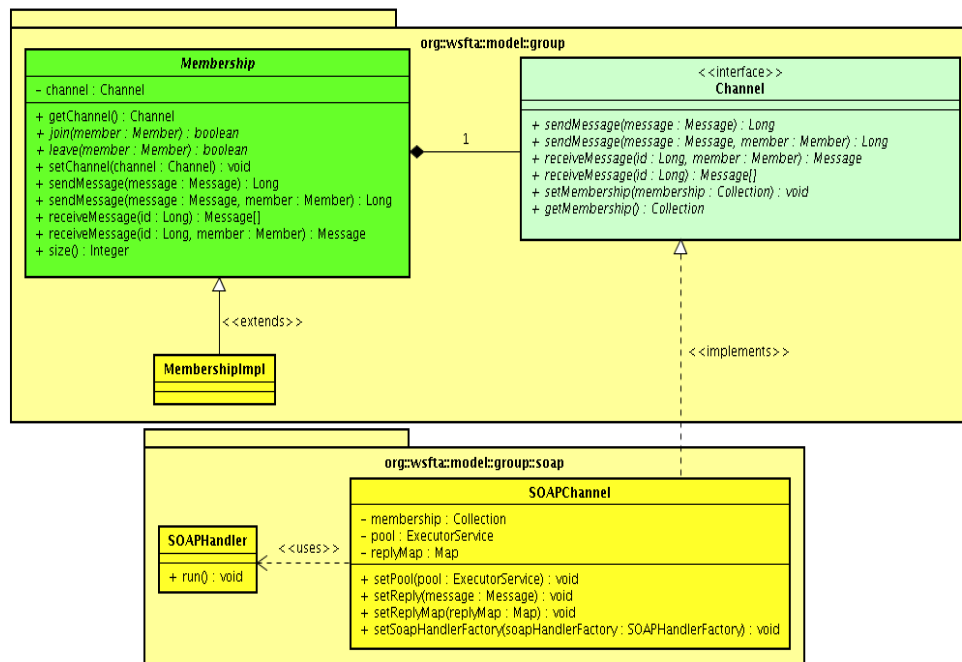



**Figura 6.1:** Principais classes do componente GC

sões referentes a este capítulo.

## 6.1 Comunicação de Grupo

O componente de comunicação de grupo foi implementado de acordo com o comportamento descrito no capítulo 5. Nessa implementação de referência foram especificadas classes e interfaces que permitem que esse componente possa proporcionar uma abstração do protocolo de transporte utilizado para troca de mensagens entre as réplicas. Na figura 6.1 é possível visualizar as principais classes do componente GC.

A classe abstrata "Membership"é responsável pelo armazenamento das informações referentes a um grupo de réplicas. Essa classe foi projetada para ser o elemento central do componente GC, contendo além de informações sobre as réplicas, uma referência para o canal de comunicação utilizado na transmissão e recepção das mensagens do grupo. Utilizando essa classe é possível:

- enviar mensagens para todas as réplicas presentes no grupo;
- enviar mensagens para apenas uma das réplicas do grupo;
- adicionar novas réplicas ao grupo;

**Algoritmo 6.1** Enviando mensagem para um grupo de réplicas

1: INÍCIO
2: $idMessage = getNextId();$
3: $insertIdInMessage(message, idMessage)$
4: **if** mensagem é sincrona **then**
5: $\quad memberMessage = createBuffer(idMessage);$
6: **end if**
7: **for** $i = 0$ to $membershipSize - 1$ **do**
8: $\quad member = getMember(i);$
9: $\quad$ **if** réplica destino é diferente de réplica remetente **then**
10: $\quad\quad pool.send(this, idMessage, message, member);$
11: $\quad\quad memberMessage.put(member, null);$
12: $\quad$ **end if**
13: **end for**
14: *return idMessage*;
15: FIM

- remover réplicas do grupo;
- receber mensagens enviadas para réplicas do grupo;

Para que todos os aspectos presentes na classe "Membership"possam ser utilizados, a classe "MembershipImpl", uma classe concreta que estende as funcionalidades da classe "Membership", foi criada. Essa classe contém as implementações e referências concretas que foram definidas na classe "Membership". Para manter a abstração do protocolo de comunicação utilizado por seus métodos, essa classe utiliza um canal de comunicação representado pela interface "Channel". Essa interface possibilita a implementação e utilização de diferentes protocolos de comunicação. Cada um desses protocolos é representado por uma classe que implementa essa interface, adicionando a seus métodos a lógica de programação necessária para a transmissão e recepção de mensagens sobre o protocolo de comunicação ao qual representa.

A classe "SOAPChannel"implementa essa interface e define que o protocolo de comunicação utilizado é o SOAP. É importante salientar que o protocolo SOAP é utilizado sobre os protocolos *HTTP* e *TCP*, sendo que o *TCP* fornece a confiabilidade na difusão ponto a ponto de mensagens realizada por intermédio desse componente. Com base nessa confiabilidade fornecida pelo proto-

**Algoritmo 6.2** Recebendo mensagens de outras réplicas do grupo

1: INÍCIO
2: **if** $isResponse(message) == true$ **then**
3: $idMessage = extractId(message);$
4: $member = extractMember(message);$
5: **if** $verifyBuffer(idMessage) == false$ **then**
6: $createBuffer(idMessage);$
7: **end if**
8: $insertMessageInBuffer(idMessage, message, member);$
9: **else**
10: $notifyRecvMessage(message);$
11: **end if**
12: FIM

colo TCP, o componente GC realiza controles adicionais para garantir difusão de grupo confiável. O algoritmo 6.1 representa os passos necessários para o envio de mensagens que tenham como destinatários todas as réplicas ativas de um grupo. Esses passos descrevem o comportamento do método $sendMessage(message)$ presente na interface "Channel"e implementado pela classe "SOAPChannel".

Antes de efetivamente enviar a mensagem para o grupo, é necessário identificar o envio dessa mensagem unicamente. Essa identificação permite o reconhecimento das respostas relacionadas com a mensagem enviada. No algoritmo 6.1 essa identificação é armazenada pela variável *idMessage* que recebe um número gerado pela chamada do método $getNextId()$. Após isso, esse número é inserido na mensagem (na chamada do método *insertIdMessage*) para que as respostas dessa mensagem possam ser identificadas. Se essa mensagem for síncrona, um *buffer* é criado para armazenar as respostas que obrigatoriamente precisam ser enviadas pelos destinatários dessa mensagem. Observa-se no algoritmo 6.1 a existência de um laço finito responsável por disparar o envio da mensagem para cada uma das réplicas presentes no grupo. Esse envio utiliza um *pool* de *threads*, permitindo que todas as mensagens possam ser enviadas em paralelo. Para finalizar a execução do algoritmo, o número de identificação é retornado para que as respostas dessa mensagem possam ser resgatadas posteriormente.

Quando uma réplica recebe mensagens de outras réplicas, o algoritmo 6.2 é executado. Esse algoritmo representa a execução do método $run()$ presente na classe "SOAPHandler". Essa classe

**Algoritmo 6.3** Verifica e devolve resposta(s) de mensagem(ns) enviada(s) anteriormente

1: INÍCIO
2: $isFinish = false$;
3: $memberMessage = getBuffer(idMessage)$;
4: **while** $isFinish == false$ **do**
5: **if** memberMessage $\subset$ *null* **then**
6: $wait(100)$;
7: **end if**
8: **end while**
9: $removeBuffer(idMessage)$;
10: $return\ memberMessage.values()$;
11: FIM

é responsável pela recepção de todas as mensagens utilizando para isso várias instâncias que são executadas concorrentemente/paralelamente. Assim que a mensagem é recebida, é verificada a existência de algum identificador que permita classificar a mensagem recebida como uma resposta de alguma mensagem enviada anteriormente ou não. Dessas respostas são extraídas informações sobre o identificador da mensagem e a réplica que enviou a resposta (chamada dos métodos *extractId* e *extractMember*). Na seqüência é verificada a existência de um *buffer* para o identificador extraído da mensagem e, caso não exista, esse *buffer* é criado. Logo após isso, a mensagem recebida é inserida no *buffer*, terminando assim a execução do algoritmo. Caso a mensagem não possua um identificador que a caracterize como uma resposta, o padrão de projeto *Observer* [Freeman et al., 2004] é utilizado para notificar que uma nova mensagem foi recebida e precisa ser tratada pela arquitetura. Essa notificação é realizada quando o método *notifyRecvMessage* é invocado.

Por fim, para verificar se respostas de mensagens enviadas anteriormente já estão disponíveis nos *buffers* criados pelos algoritmos 6.1 e 6.2, o método *receiveMessage*, presente na interface "Channel"é invocado. O comportamento desse método pode ser visualizado no algoritmo 6.3. Nota-se nesse algoritmo que o processo de verificar se respostas já estão disponíveis é do tipo bloqueante, ou seja, caso a(s) resposta(s) não esteja(m) disponível(is) o método ficará bloqueado até que todas as respostas relacionadas a uma determinada mensagem sejam recebidas. Para isso, existe um laço principal responsável por realizar verificações periódicas no *buffer* de mensagens, representado pela variável *memberMessage*. Quando todas as respostas forem recebidas, esse

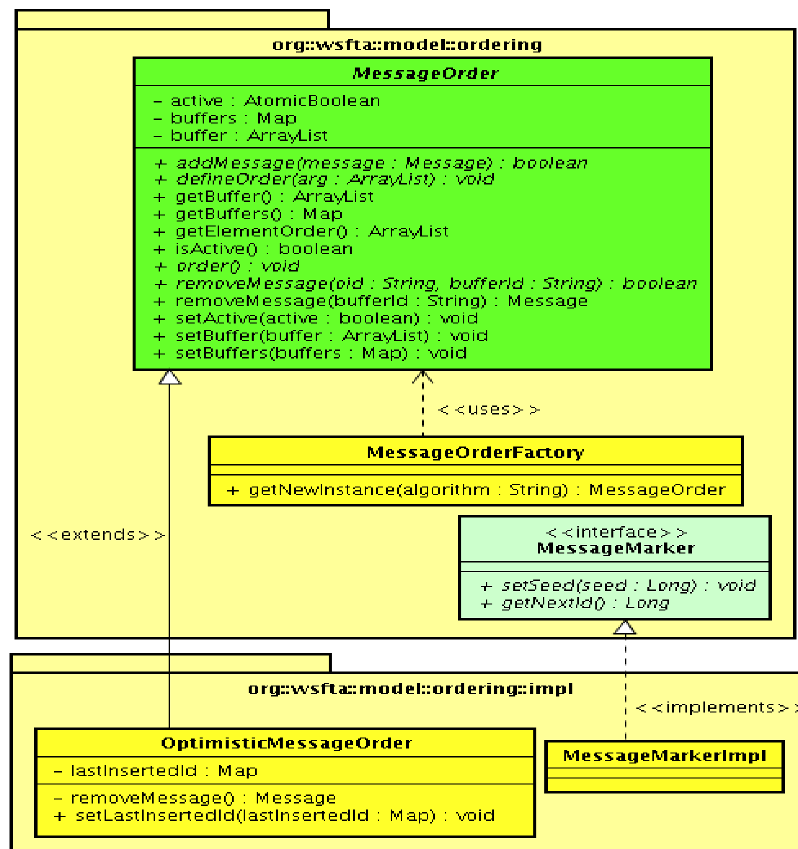



**Figura 6.2:** Principais classes do componente MO

*buffer* é removido e as mensagens contidas nele são retornadas.

## 6.2 Ordenação de Mensagens

O componente MO tem a responsabilidade de garantir a ordenação das mensagens, utilizando para isso um algoritmo de ordenação. Esse componente é formado por um conjunto de classes que permitem a definição de todas as suas características. Entre essas classes, destacam-se as classes presentes na figura 6.2. A interface "MessageMarker"é responsável por definir métodos que possibilitam a geração de números seqüenciais que são utilizados para a marcação das mensagens recebidas de consumidores do serviço. Assim que uma mensagem de um consumidor de serviços é recebida, essa mensagem é passada para a instância da classe "MessageMarkerImpl", que se encarrega de fornecer o número de identificação que será utilizado para marcar essa mensagem. Essa marcação só é realizada pelos líderes de cada grupo, já que são esses líderes que são visíveis para os consumidores do serviço. Após essa marcação, a mensagem já pode ser encaminhada para

os outros líderes e também para as demais réplicas do grupo.

Outra importante classe do componente MO é a classe "MessageOrderFactory". Essa utiliza o padrão de projeto *Factory* [Freeman et al., 2004] e é responsável por instanciar objetos de classes que estendam a classe abstrata "MessageOrder". Em outras palavras, "MessageOrderFactory"instancia objetos de classes que representam algoritmos de ordenação que podem ser utilizados pelo componente. Nessa implementação de referência, a classe "OptimisticMessageOrder"representa a implementação de um algoritmo de ordenação otimista, adaptado às características presentes na WSFTA e, baseado em princípios descritos em [Sousa et al., 2002].

Para entender o funcionando dessa ordenação, é importante entender o funcionamento do algoritmo 6.4 que representa o método "addMessage"presente na classe "MessageOrder". A primeira ação tomada é verificar qual o tipo de mensagem que foi recebida. Caso seja uma mensagem convencional, ou seja, uma mensagem que não contém uma lista de identificadores usada para definição de ordem, o método *verifyMessage* retorna *true* executando as ações presentes dentro do desvio condicional no qual está inserido. Assim que começa a execução desse desvio são extraídos da mensagem os identificadores da mensagem e da réplica que marcou essa mensagem com este identificador. Para cada líder, existe um *buffer* que auxilia a manutenção da ordem de marcação realizada por cada um dos líderes. Sendo assim, é verificado se o identificador da mensagem é o igual ao identificador que pode ser inserido na lista ordenada de mensagens. Se for igual, a mensagem é inserida na lista de mensagens ordenadas e seu identificador é inserido na lista que armazena o último identificador, do líder representado por *idMember*, que foi inserido na lista de mensagens ordenadas.

Caso a réplica tenha a responsabilidade de definir a ordem das mensagens e a variável *checkThread* seja diferente de *null*, o *timeout* para ordenação de mensagens é cancelado. Na seqüência o *buffer* da réplica representada por *idMember* é verificado no intuito de saber se a(s) próxima(s) mensagem(ns) a ser(em) inserida(s) na lista de mensagens ordenadas está(ão) no *buffer*. Em caso positivo, essa inserção é realizada durante essa verificação. Após isso, o método *defineOrder* é disparado caso o número de mensagens que precisam ser agrupadas na janela de ordenação seja alcançado e a réplica seja a ordenadora de mensagens. Caso esse número de mensagens ainda não tenha sido alcançado, o *timeout* de ordenação de mensagens é disparado novamente. Esse *timeout* define um tempo máximo que o componente MO aguardará por novas mensagens. Caso esse limite seja atingido, a ordem das mensagens é definida, mesmo que o número de mensagens da janela de ordenação não tenho sido atingido.

Se o identificador da mensagem for maior que o identificador que deve ser inserido na lista de

mensagens ordenadas, marcada pela réplica representada por *idMember*, essa mensagem é inserida em um *buffer*, aguardando a chegada da(s) mensagem(ns) que precede(m) sua inserção. Por fim, caso a mensagem contenha uma lista com a ordem dos identificadores das mensagens que podem ser entregues ao serviço, nenhuma ação anterior é executada e o método *defineOrder* é invocado, recebendo essa lista como parâmetro.

Na execução do método *defineOrder* o algoritmo 6.5 é executado. Nesse método as ações executadas dependem da classificação da réplica perante o seu grupo. Caso a réplica seja um líder responsável pela definição da ordem de execução, a ordem é enviada para todos os demais líderes, se existirem, e para todas as réplicas pertencentes a seu grupo. Após isso, as mensagens são despachadas uma a uma para que possam ser executadas pelo serviço na ordem estabelecida. É importante salientar que, no caso da existência de diversos grupos, o líder que define a ordem não é fixo, sendo a escolha do líder que deve ordenar as mensagens realizada com base na seqüência na qual foram inseridos no *membership* dos líderes.

Se a réplica for o líder do seu grupo, mas não for o líder com a responsabilidade de definir a ordem de execução das mensagens, o *array* que contém essa ordem é enviado para as demais réplicas do grupo e as mensagens são entregues na ordem definida no *array*. Caso esse líder seja o próximo líder com responsabilidade de ordenar as mensagens, o método *setOrderer* é invocado. Por fim, em uma réplica que não é a líder de seu grupo, as mensagens são entregues na ordem definida no *array* recebido pelo parâmetro *arg*.

Nessa implementação de referência, um líder realiza um *multicast* de uma mensagem utilizando $n-1$ *unicasts*, assim como foi descrito no componente GC, onde $n$ é igual ao número de líderes. Portanto, cada uma dessas mensagens *unicast* recebe a adição de um tempo de atraso na tentativa de garantir sua entrega ordenada. Esse atraso é calculado com base no tempo de comunicação de rede entre emissor e receptor. Considere um conjunto de líderes $L = A1, B1, C1$. Imagine que o líder $A1$ envie mensagens para os líderes $B1$ e $C1$. O tempo de envio dessa mensagem para o líder $B1$ é de 15ms, enquanto que esse tempo de envio para o líder $C1$ é de $10ms$. Pode-se notar que o líder $C1$ recebe mensagens de $A1$ antes que o líder $B1$. Para compensar essa diferença dos tempos de comunicação do líder $A1$ para os líderes $B1$ e $C1$, um atraso deve ser adicionado ao processo de envio de mensagens. A fórmula 6.1 descreve como este tempo de atraso deve ser calculado para cada réplica.

$$T_{a_{(r \longrightarrow r')}} = maxTnode_{(r)} - T_{(r \longrightarrow r')} \tag{6.1}$$

Onde:

$T_{a_{(r \longrightarrow r')}}$: Tempo de atraso adicionado ao envio de uma mensagem de uma réplica $r$ para outra $r'$;

$T_{(r \longrightarrow r')}$: Tempo de envio de mensagens de uma réplica $r$ para outra $r'$;

$maxTnode_{(r)}$: Maior $t_{(r \longrightarrow r')}$.

No exemplo citado anteriormente no texto, $maxTnode_{(A1)} = 15ms$ e $T_{(A1->C1)} = 5ms$, isto é, mensagens enviadas de $A1$ para $C1$ sofrem a aplicação de um atraso de $5ms$. Caso ocorra uma inconsistência na ordenação otimista, essa inconsistência será corrigida pelo *array* que contém a ordem correta de execução das mensagens. Como o tempo de resposta de uma mensagem depende do tempo de ordenação da mesma, em seu pior caso, o tempo de resposta terá o acréscimo de um atraso definido pela fórmula 6.2.

$$Delay = timeoutOrder + tSearch \tag{6.2}$$

Onde:

*Delay*: Tempo que será adicionado ao tempo de resposta de um mensagem, em seu pior caso;

*timeoutOrder*: *Timeout* para disparar a definição da ordem de execução das mensagens;

*tSearch*: Tempo de busca da mensagem no *buffer* local.

Quanto maior a diferença entre a ordem dos identificadores presentes no *array* e do *buffer* local que contém as mensagens recebidas pela réplica, maior será o valor do tempo de busca da mensagem, sendo por conseqüência, maior o tempo de resposta da mesma. Além disso, é importante destacar que nessa implementação a difusão de mensagens confiáveis é garantida pelo componente de comunicação de grupo presente na implementação da WSFTA.

## 6.3 Detecção de Falha

Além da garantia de ordenação, é necessário realizar o monitoramento do serviço para a identificação de possíveis falhas. Essa tarefa é desempenhada pelos RFDs e GFDs. As principais classes que representam esses componentes podem ser visualizadas na figura 6.3. A interface "FailureDetector"define os métodos que são invocados na atualização do tempo de resposta do serviço e do detector de falha monitorado. A classe abstrata "AbstractFailureDetector"define o

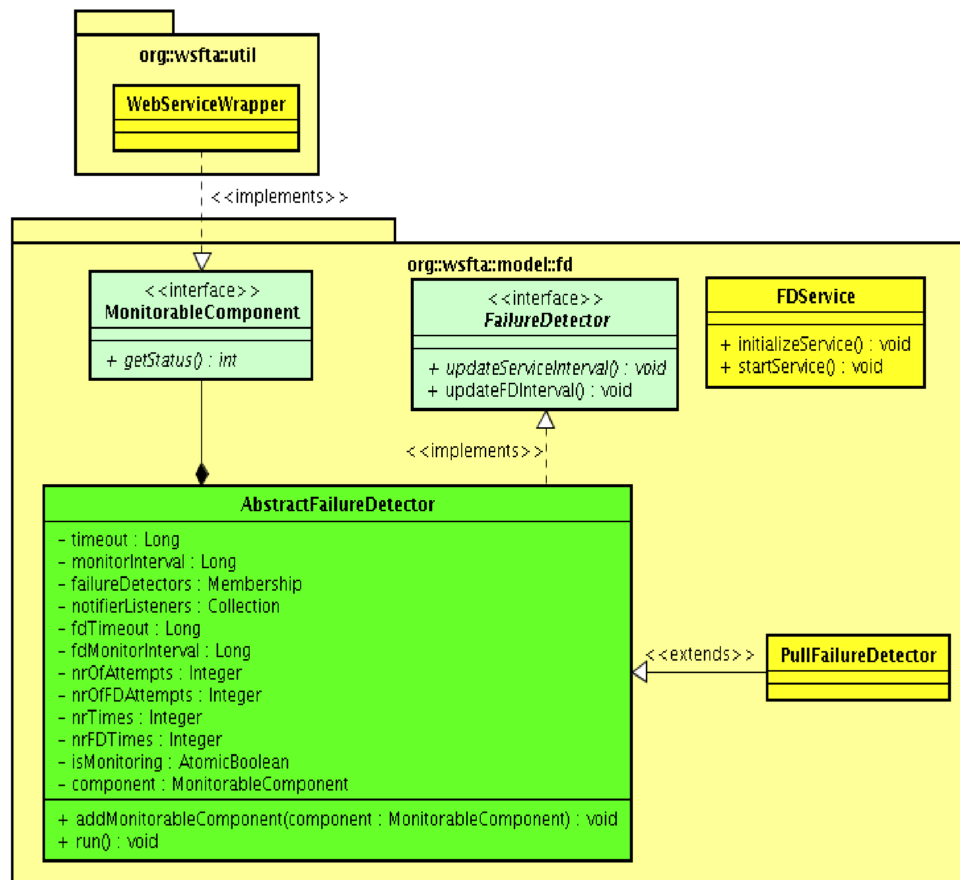



**Figura 6.3:** Principais classes dos componentes RFD e GFD

corpo de todo detector de falha presente na WSFTA. Essa classe contém os atributos e métodos que devem ser implementados por classes concretas para que a detecção de falha possa ocorrer. A classe concreta definida nessa implementação de referência é a classe "PullFailureDetector". Essa classe é responsável por implementar o método de monitoração *Pull*. A interface "MonitorableComponent"define o método que que será invocado pelo detector de falha para verificar se o serviço está ativo. Para isso, a classe "WebServiceWrapper"foi criada, abstraindo do detector detalhes de implementações e de protocolo de comunicação utilizados para realizar essa verificação. Para completar, a classe "FDService"define métodos responsáveis pela inicialização e execução do serviço de detecção de falha.

No método de monitoração *Pull*, os detectores enviam periodicamente mensagens do tipo *"Are you alive?"* para verificar se o serviço monitorado está ativo. Se o detector não receber a resposta desta mensagem no limite de tempo esperado, ele passa a suspeitar do serviço. Apesar de ocasionalmente o serviço demorar mais que o limite de tempo (*timeout*) esperado para responder

a mensagem do detector, esta demora não se caracteriza uma falha. Vários fatores podem afetar esse fenômeno: latência, links de comunicação falhos, carga no serviço, entre outros.

Todos esses fatores podem induzir o detector a identificar uma falha no serviço quando, na verdade, o serviço está levando mais tempo para responder que o *timeout* configurado inicialmente. Sendo assim, o algoritmo de detecção de falha utilizado pelos detectores da WSFTA ajusta o *timeout* conforme seu próprio histórico e tempo de resposta do serviço. A figura 6.4 exibe um diagrama de eventos que representa a troca de mensagens entre detector e serviço e a detecção da falha deste.

Para calcular o ajuste dos tempos envolvidos no processo de detecção, informações tais como tempo de resposta do serviço, *timeout* atual do detector e intervalo de monitoração são cruciais para que este cálculo obtenha uma maior exatidão. Visando um melhor entendimento desse processo, o conjunto de equações 6.3 a 6.5 formalizam e descrevem melhor a tarefa de ajuste do *timeout* realizada pelos detectores.

$$T_{timeout_{(i+1)}} = \alpha T_{timeout_{(i)}} + (1-\alpha) T_{resp_{(i)}} \tag{6.3}$$

$$T_{monitor_{(i+1)}} = T_{timeout_{(i+1)}} + \theta \tag{6.4}$$

$$T_{falha} = \beta T_{monitor_{(i-1)}} + T_{timeout_{(i)}} + c \tag{6.5}$$

Onde:

$T_{timeout}$ **:** Tempo de timeout do detector;

$T_{monitor}$ **:** Tempo atribuído ao intervalo de monitoração;

$T_{resp}$ **:** Tempo de resposta do serviço;

$T_{falha}$ **:** Tempo total para a detecção da falha do serviço;

$i$ **:** Número de mensagens enviadas ao serviço;

$c$ **:** Tempo necessário para o processamento e obtenção dos valores de *timeout* e intervalo de monitoração;

$\alpha$ **:** Constante para cálculo de ajuste do *timeout*;

$\beta$ **:** Número máximo de falhas na resposta do serviço;

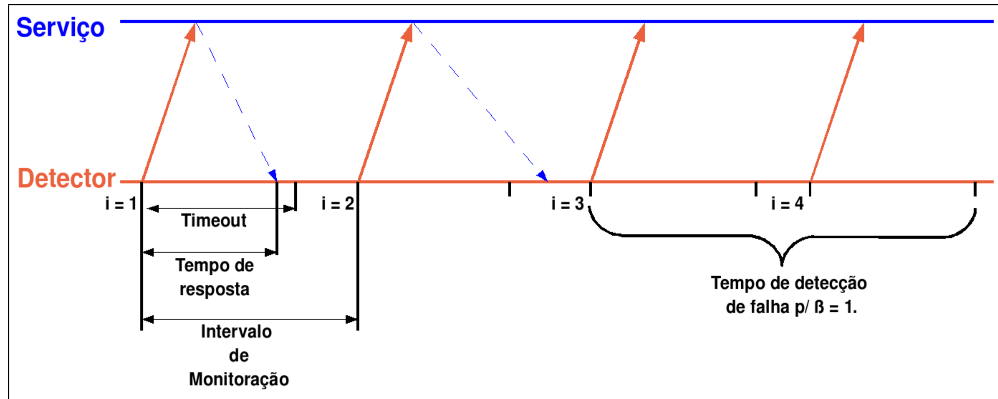



**Figura 6.4:** Diagrama de eventos entre Detector e Serviço

$\gamma$ **:** Valor para o *timeout* inicial do detector. Logo, $T_{timeout_{(0)}} = T_{resp_{(0)}} = \gamma$;

$\theta$**:** Constante para cálculo do ajuste do intervalo de monitoração;

Para exemplificar o ajuste de tempo realizado pelos detectores e utilizando como base o diagrama da figura 6.4, pode-se calcular qual o tempo de detecção de falha e os tempos de ajuste do *timeout* ($T_{timeout}$) e intervalo de monitoração ($T_{monitor}$) para uma situação hipotética. Para isso, serão assumidos os valores:

$c$ **=** $0$ (Desconsidera o tempo gasto para processamento e obtenção dos valores de intervalo de tempo);

$\alpha$ **=** $0,9$ (Atribui $90\%$ do tempo de ajuste do *timeout* aos tempos de *timeout* anteriores);

$\beta$ **=** $1$**;**

$\gamma$ **=** $100ms$**;**

$\theta$ **=** $200ms$**.**

Para $i = 0$, tem-se:

$$T_{timeout_{(1)}} = \alpha T_{timeout_{(0)}} + (1-\alpha)T_{resp_{(0)}} \tag{6.6}$$

$$T_{timeout_{(1)}} = 0,9 * 100 + (1-0,9) * 100 \tag{6.7}$$

$$T_{timeout_{(1)}} = 100ms; \tag{6.8}$$

$$T_{monitor_{(1)}} = T_{timeout_{(1)}} + \theta \tag{6.9}$$

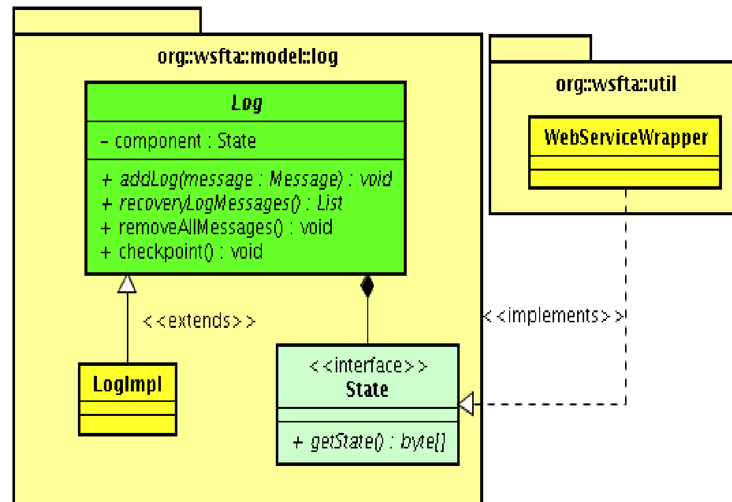



**Figura 6.5:** Principais classes do componente LOG

$$T_{monitor_{(1)}} = 100 + 200 \tag{6.10}$$

$$T_{monitor_{(1)}} = 300ms \tag{6.11}$$

Supondo $T_{resp_{(1)}} = 50ms$, $T_{resp_{(2)}} = 102ms$ e a falha do serviço após a segunda resposta, o tempo para detecção da falha é igual a:

$$T_{falha} = \beta T_{monitor_{(3)}} + T_{timeout_{(4)}} + c \tag{6.12}$$

$$T_{falha} = 1 * 295,7 + 115,7 + 0 \tag{6.13}$$

$$T_{falha} = 411,4ms \tag{6.14}$$

## 6.4 Armazenamento e recuperação de LOG

O componente para armazenamento e recuperação de log foi criado com o objetivo de armazenar informações sobre o estado do serviço que utiliza a WSFTA. Como informações de estado são muito específicas para cada serviço, o serviço precisa implementar alguma operação que permite ao componente de LOG obter o seu estado. Nessa implementação de referência, essa operação deve ser chamada de *getState* e retornar um *array* de *bytes* contendo o estado do serviço. As classes que formam o componente LOG podem ser visualizadas na figura 6.5.

A classe abstrata "Log"é a principal classe desse componente. Nessa classe são definidas as operações que podem ser invocadas, possuindo também uma referência para objetos de clas-

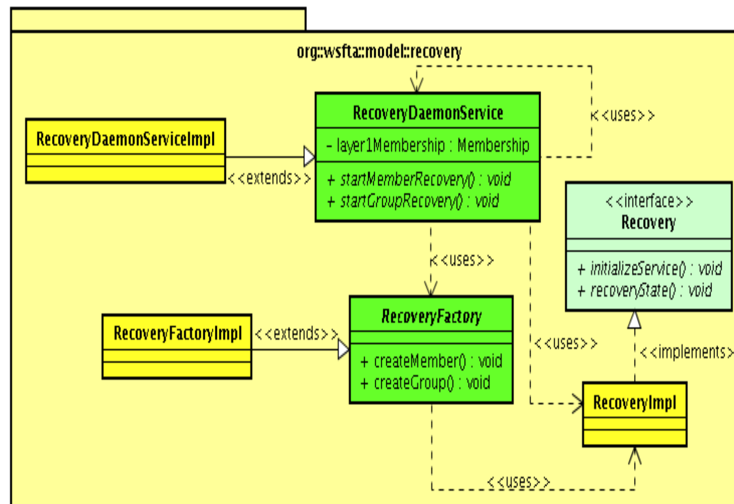



**Figura 6.6:** Principais classes do componente REC

ses que implementam a interface "State". Essa interface é a responsável por definir a operação que será invocada para o armazenamento de estado do serviço. Para que a implementação do componente LOG não dependa diretamente da implementação do serviço, a classe "WebService-Wrapper"realiza a ligação entre o componente e o serviço. Sendo assim, uma invocação no método *getState*, presente na interface "State", resulta na invocação da operação *getState* presente no serviço. Como a classe "Log"é uma classe abstrata, a classe "LogImpl"estende suas funcionalidades para que o componente LOG possa funcionar, armazenando informações que podem ser utilizadas posteriormente.

## 6.5 Recuperação de Falha

O componente de recuperação é outro componente da WSFTA no qual sua implementação depende do serviço. Essa dependência é derivada da especificidade das rotinas de recuperação exigidas por cada serviço. Porém, para padronizar os métodos e a forma de implementação dessa recuperação, este componente define as classes e interfaces que precisam ser implementadas. Essas classes e interfaces podem ser visualizadas na figura 6.6.

A interface "Recovery"define as operações básicas que precisam ser implementadas pelo desenvolvedor do serviço. Essas operações permitem que o estado do serviço possa ser inicializado e recuperado. Como "Recovery"é uma interface, a implementação concreta dessas operações deve ser realizada na classe "RecoveryImpl". As demais classes "RecoveryDaemonService", "Reco-

veryFactory"e suas respectivas implementações, devem ser especificadas para que o processo de recuperação de estado das réplicas do serviço possa ser realizado de maneira automatizada. É importante destacar que a implementação de referência não provê mecanismos para recuperação automatizada, por se tratar de uma atividade dependente da implementação do serviço que será recuperado.

## 6.6 Experimentos

Esta seção apresenta a descrição de alguns experimentos realizados com o objetivo de avaliar o desempenho da implementação de referência da WSFTA, descrita durante todo o capítulo 6. Assim como a implementação de referência, os serviços utilizados nesses testes foram implementados na linguagem Java com auxílio das APIs do JAX-WS e do JDK 6.0. Os experimentos tem o objetivo de avaliar o desempenho da arquitetura comparando diferentes configurações e estilos de replicação, tendo sido executados em quatro computadores com processadores AMD $1.83GHz$, $512MB$ de memória RAM e sistema operacional Windows XP SP2 e um computador com processador AMD $1.8GHz$, $512MB$ de memória RAM e sistema operacional GNU/Linux Slackware 12.0, executando a ferramenta para testes de Serviços Web chamada SOAPUI [SOAPUI, 2008]. O serviço utilizado tinha a característica de devolver mensagens SOAP com o tamanho solicitado pelo cliente. Além disso, para todos os testes foram atribuídos os valores de $10$ mensagens para a janela de ordenação, $200ms$ para o *timeout* de ordenação, $2s$ para o intervalo de monitoração e $1,8s$ para o *timeout* inicial dos detectores de falha.

Esses computadores estavam interconectados por uma rede local de $100Mbps$ e cada um deles executava uma réplica do serviço. O objetivo principal desses experimentos foi realizar a avaliação de desempenho da WSFTA utilizando diferentes configurações e técnicas de replicação. Para obtenção dos resultados que expressam o *overhead* adicionado por uma configuração específica da WSFTA, os mesmos experimentos foram executados sobre um serviço não replicado e que não utilizasse qualquer *framework* ou arquitetura de TF.

Foram realizados ao todo cinco experimentos. O primeiro experimento, representado pelo gráfico da figura 6.7, teve o objetivo de mensurar o *overhead* da WSFTA sobre o serviço, utilizando em cada um dos grupos a técnica de replicação semi-ativa. A obtenção desse *overhead* levou em consideração o tamanho das mensagens variando de $2KB$ a $128KB$, com $1000$ invocações consecutivas. Para o segundo experimento, representado pelo gráfico da figura 6.8, o número de usuários simultâneos enviando mensagens para o serviço variou entre $2$ e $20$. Esse experimento também

tinha o objetivo de verificar o impacto do *overhead* da WSFTA sobre o tempo de resposta do serviço. É importante destacar que o tamanho de mensagem utilizado para a realização desse teste foi de $4$*KB* e que a ferramenta de testes SOAPUI [SOAPUI, 2008] foi responsável pela simulação do número de usuários simultâneos acessando o serviço.

Já no terceiro experimento, representado pelo gráfico da figura 6.9, é possível verificar a vazão do serviço que utiliza a WSFTA com replicação semi-ativa, variando também o tamanho de 2 a $128$*KB*, com $1000$ invocações consecutivas. No quarto experimento, foram comparados os *overheads* das técnicas de replicação passiva e semi-ativa utilizando três diferentes configurações de grupo e tamanho de mensagem de $8$*KB*. E por fim, no quinto e último experimento, foi verificado o comportamento do serviço para uma carga de usuários simultâneos variando dinamicamente entre os valores de $5$ e $15$, para mensagens com $4$*KB*.

## 6.7 Análise dos Resultados

Analisando o gráfico da figura 6.7, é possível perceber que para mensagens entre $2$ e $16$*KB* não há uma diferença de *overhead* muito acentuada entre todas as configurações de grupo. Para mensagens de $32$*KB* pode-se perceber uma diferença de aproximadamente $68$*ms* entre as configurações de um grupo com três réplicas e um grupo com quatro réplicas. Essa diferença é mais acentuada que nas mensagens que variam de $2$*KB* a $16$*KB*. Diversas variáveis tais como latência da rede, escolha da resposta que será devolvida ao consumidor do serviço e até mesmo o tamanho da mensagem influenciam nessa diferença. Nessa implementação de referência, para comparações entre as configurações que utilizam apenas um grupo, os indícios apontam que o aumento do *overhead* do grupo com quatro réplicas é derivado da união do tamanho da mensagem com a execução do algoritmo de votação da resposta que será enviada ao consumidor do serviço. Como nesse algoritmo as respostas são comparadas uma a uma, o crescimento do tamanho da mensagem juntamente com a adição de mais uma réplica no grupo, contribuem significativamente para o aumento do *overhead* entre essas configurações.

Se ainda forem comparadas as configurações de um grupo com quatro réplicas e dois grupos com duas réplicas cada, percebe-se que a adição de mais um grupo de réplicas, para mensagens maiores que $32$*KB*, diminui o *overhead* do tempo de resposta, além de aumentar a disponibilidade do serviço. Isso é derivado da paralelização do algoritmo de votação. Com mais de um grupo, cada líder executa esse algoritmo para selecionar uma resposta dentro de seu grupo que será marcada com um número que indique a quantidade de ocorrências da mesma. Isso é realizado em paralelo

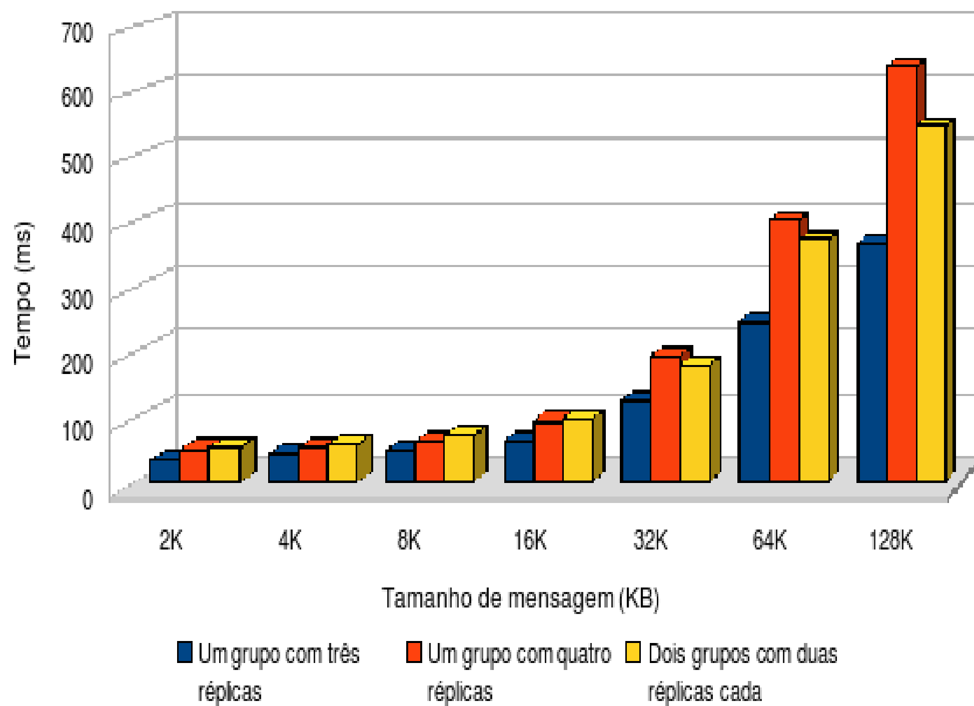



**Figura 6.7:** Overhead da replicação semi-ativa quanto ao tamanho das mensagens

por todos os líderes. Feito isso, o líder que responderá para o consumidor realiza uma votação contabilizando o número de ocorrências de respostas encaminhadas pelos outros líderes juntamente com as respostas originadas pelas réplicas de seu grupo. Logo, esse processo de votação é menos custoso nas configurações com mais de um grupo. Portanto, a análise do gráfico da figura 6.7 indica que o aumento da disponibilidade do serviço, adicionando mais um grupo de réplicas, possibilita ainda uma redução do *overhead* do serviço. É importante deixar claro que esses testes foram realizados em uma LAN, o que não permite elaborar conclusões a respeito do comportamento da arquitetura, em relação ao *overhead* do serviço, em uma WAN.

Para avaliar o impacto do acesso simultâneo ao serviço, foram elaborados testes que variavam a quantidade de usuários acessando o serviço simultaneamente. O gráfico da figura 6.8 exibe o impacto da WSFTA no tempo de resposta do serviço, quando o mesmo é acessado simultaneamente por um conjunto que varia de $2$ a $40$ usuários. Como já citado anteriormente, esse teste utilizou mensagens com tamanho fixo de $4KB$. Pode-se notar no gráfico da figura 6.8 que a divisão das réplicas em mais de um grupo proporciona uma maior disponibilidade do serviço e uma diminuição do *overhead*, se comparado a configurações que possuam o mesmo número de réplicas organizadas em apenas um grupo.

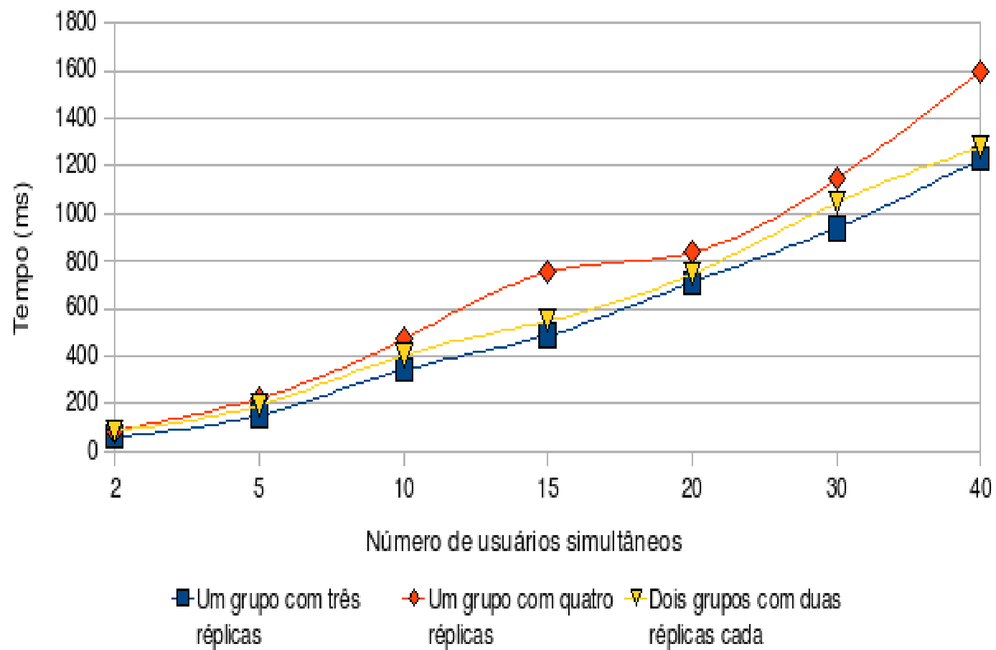



**Figura 6.8:** Overhead da replicação semi-ativa quanto ao número de usuários simultâneos

Nesse gráfico ainda é possível notar que a a carga imposta pelo número de usuários simultâneos acessando o serviço aproxima os *overheads* das configurações com quatro réplicas em dois grupos e três réplicas em apenas um grupo. Um dos principais motivos para isso, além da votação paralela, é a divisão entre os dois líderes do consumo de memória e do processamento necessários para tratar as conexões dos consumidores do serviço. Outra informação importante é referente a configuração com quatro réplicas organizadas em apenas um grupo.

No gráfico da figura 6.9 é possível visualizar a comparação da vazão obtida pelo serviço nas três configurações testadas. Essa vazão foi testada com mensagens variando de $2KB$ a $128KB$. Já no gráfico da figura 6.10 foi realizada uma comparação entre as técnicas de replicação semi-ativa e passiva, utilizando três configurações diferentes. Nesse gráfico nota-se que a utilização da replicação passiva diminui consideravelmente o *overhead* adicionado ao tempo de resposta do serviço. Essa diminuição é característica da técnica de replicação passiva onde, nesse caso, somente os líderes executavam as requisições enviadas pelos consumidores do serviço, deixando as demais réplicas somente como *backups*.

Dando continuidade a observação do gráfico da figura 6.10, é possível perceber que utilizando replicação passiva, as duas configurações que utilizam apenas um grupo tem *overheads*

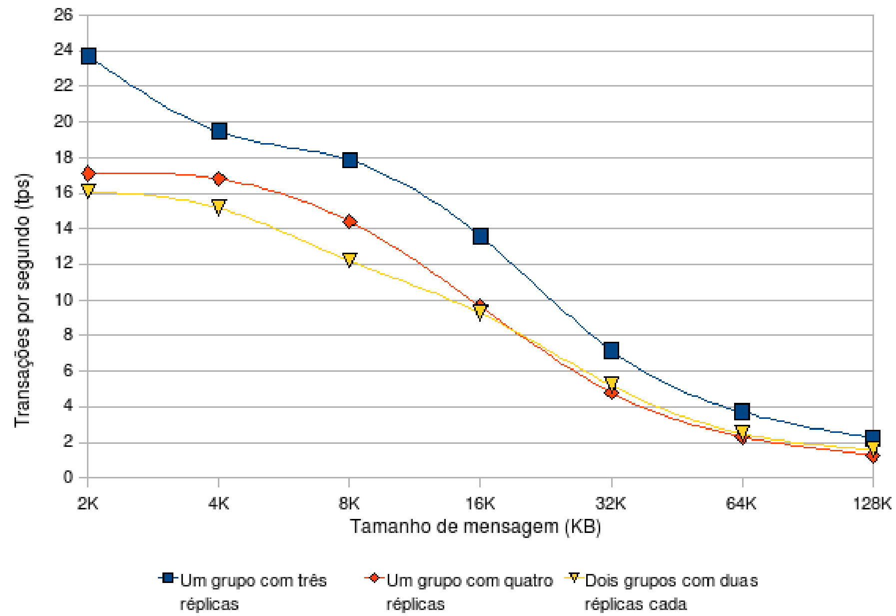



**Figura 6.9:** Vazão do serviço com a WSFTA utilizando replicação semi-ativa em diferentes configurações

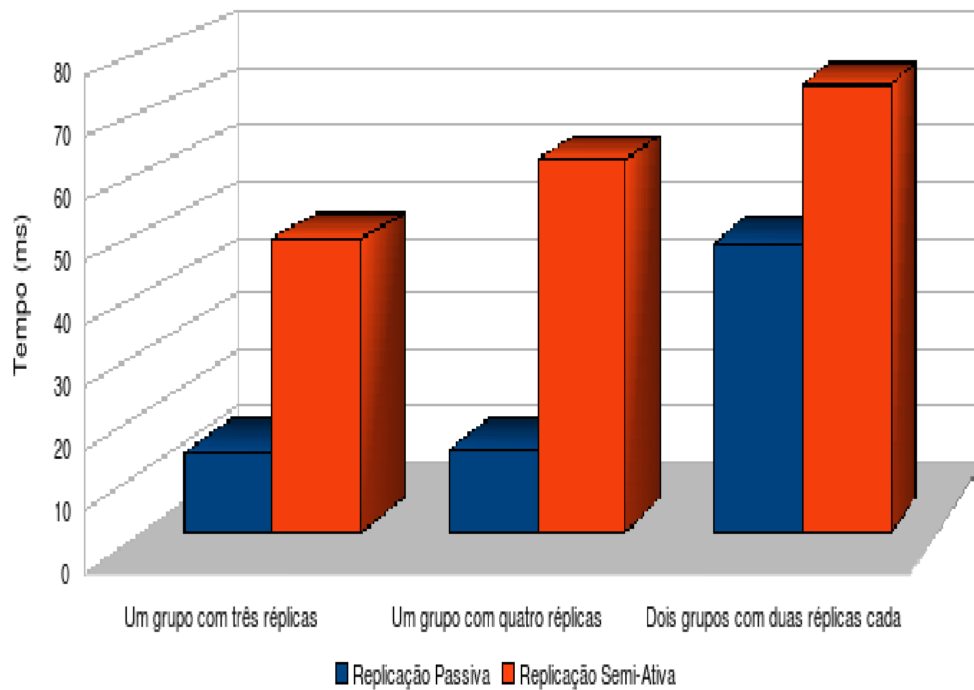



**Figura 6.10:** Comparação do overhead do serviço utilizando a WSFTA com replicação passiva e semi-ativa

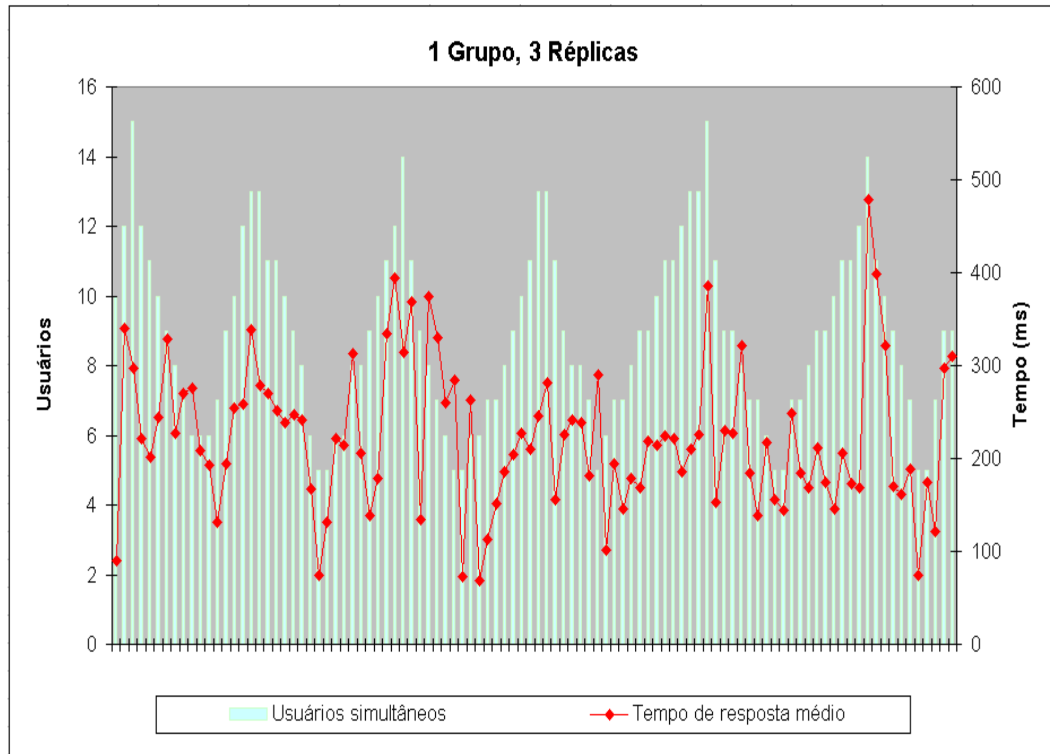



**Figura 6.11:** Variação do tempo de resposta em função da variação dinâmica do número de usuários simultâneos

menores que a configuração que utiliza réplicas organizadas em dois grupos. Isso é proveniente da diferença da técnica de replicação utilizada pelo líderes e por seus grupos isoladamente. Em outras palavras, os líderes recebem e executam as requisições, além de realizar a votação que elege a resposta que deve ser enviada ao consumidor, enquanto as demais réplicas somente recebem essas mensagens.

Já no gráfico da figura 6.11 é possível observar a variação no tempo de resposta em relação à variação no número de usuários simultâneos acessando o serviço. Nesse experimento foi utilizada a configuração de três réplicas organizadas em apenas um grupo. Nota-se que o gráfico da figura 6.11 que na medida que o número de usuários cresce o tempo de resposta médio aumenta e na media que esse número de usuários decresce o tempo de resposta também é reduzido. É possível observar que a implementação da WSFTA pode ser utilizada em cenários onde existe variações dinâmicas de carga do número de usuários simultâneos, possibilitando sua utilização em serviços que recebem cargas com essas características.

## 6.8 Comparação com Trabalhos Relacionados

O *middleware* para concepção de Serviços Web confiáveis proposto por [Ye and Shen, 2005], descrito na seção 4.1, não suporta consumidores legados, ou seja, consumidores legados não podem usufruir das características de TF presentes nesse trabalho porque não dispõe de *proxy*. Em serviços que utilizem a WSFTA, consumidores legados podem usufruir das características de TF fornecidas pela arquitetura, mesmo que não utilizem o WSFTA-Proxy. Outro ponto importante é em relação à ordenação das requisições recebidas por consumidores. Como não existe uma réplica central que define a ordem de execução das requisições para as outras réplicas, o aumento do tempo de definição da ordem de execução das requisições é diretamente proporcional ao aumento do número de réplicas do serviço. Para diminuir o impacto da ordenação no tempo de resposta do serviço, [Ye and Shen, 2005] definiram que somente operações que realizam alteração de estado devem ser ordenadas. Porém, cada uma dessas operações é ordenada em uma fila individual, ou seja, operações distintas podem ser executadas simultaneamente. Isso significa que caso essas operações alterem um mesmo conjunto de dados, essa execução simultânea deixaria o serviço em um estado inconsistente.

Na WSFTA, cada grupo possui um líder, sendo que somente esses líderes definem a ordem de execução das mensagens para todas as réplicas. Isso permite que o número de réplicas aumente, sem que o tempo para definição da ordem de execução das requisições aumente de forma proporcional, além de não possuir problemas de inconsistência derivados da execução simultânea de operações.

Consumidores de serviços que utilizem o SWS [Li et al., 2005], descrito na seção 4.2, devem ser implementados com auxílio de um pacote específico que possibilita o conhecimento dos grupos de replicação, podendo assim usufruir das vantagens da tolerância a faltas do serviço, ou seja, não possui suporte a consumidores legados. A WSFTA fornece tolerância a faltas para o serviço em questão, mesmo que o consumidor do serviço seja um consumidor legado. O SWS também utiliza a idéia de grupos, possuindo protocolos específicos de comunicação e ordenação de mensagens. O protocolo de ordenação SWS-MO utiliza acordos bizantinos para prover consenso entre todas as réplicas do grupo. Uma estratégia utilizada é a execução desse algoritmo somente quando existe diferença entre a requisição que deve ser executada em cada réplica. Logo, quanto maior o número de réplicas no grupo, maior será o *overhead* desse algoritmo no tempo de resposta do serviço. A vantagem da WSFTA é que a definição da ordem é realizada somente pelos líderes de cada grupo, possibilitando um menor tempo de processamento e reduzindo a utilização da rede para realizar

essa tarefa.

No FTWEB [Santos et al., 2005], descrito na seção 4.3, consumidores do serviços interagem com um componente denominado *WSDispatcher*. Esse componente é responsável pelo gerenciamento das réplicas e pelo seqüenciamento das requisições, fornecendo assim características de ordenação total na execução das requisições por parte das réplicas do serviço. No FTWEB um componente chamado *WSClientDriver* é responsável por detectar a falha do componente *WSDispatcherEngine* e transferir a requisição para um réplica *backup* desse componente. Caso o serviço esteja sendo acessado por vários usuários ao mesmo tempo, um problema na rede pode fazer com que um desses clientes considere que o *WSDispatcher* falhou transferindo suas requisições para a réplica *backup*. Logo, uma parte dos clientes estariam enviando requisições para o *WSDispatcher* e outros enviando requisições para a réplica de *backup* desse componente. Nessa situação, onde o *WSDispatcherEngine* e sua réplica *backup* estão executando requisições de seus consumidores simultaneamente, o serviço pode entrar em um estado inconsistente.

Como na WSFTA a detecção da falha das réplicas lideres, que podem ser acessadas diretamente pelo serviço, é realizada pelos componentes de detecção de falha que são executados juntamente com o serviço, a arquitetura não sofre do mesmo problema encontrado no FTWEB. Além disso, a WSFTA não possui dependência com nenhum *framework* ou tecnologia, diferentemente do FTWEB que possui forte acoplamento com o FT-CORBA [OMG, 2002] e o Apache Axis [APACHE, 2008], que foram utilizados em seu desenvolvimento e são utilizados na execução de seus componentes.

Fang, C. et al. [Fang et al., 2007] definiram uma arquitetura de tolerância a faltas denominada FT-SOAP, descrita na seção 4.4. Assim como o FTWEB, o FT-SOAP possui forte acoplamento com o FT-CORBA[OMG, 2002]. O FT-SOAP utiliza replicação passiva para fornecer tolerância a faltas ao serviço e faz uma alteração na especificação WSDL para adicionar um novo elemento que permite referenciar Serviços Web que formam um grupo de tolerância a faltas do serviço. Em contrapartida, a WSFTA não possui dependência com nenhum *framework* ou plataforma e também não tem a necessidade de alterar as especificações utilizadas em Serviços Web. Outro ponto importante que deve ser levado em consideração no FT-SOAP, é o fato de haver a dependência de um componente, denominado *FT-SOAP-Engine*, para que consumidores possam usufruir das características de tolerância a faltas providas pelo FT-SOAP. Logo, consumidores legados não tem a possibilidade de usufruir das características de TF fornecidas pelo FT-SOAP. Na WSFTA, consumidores legados podem usufruir das características de TF fornecidas ao serviço de forma transparente.

| | Middleware | SWS | WS-Replication | FTWEB | FT-SOAP | WSFTA |
|---|---|---|---|---|---|---|
| **Técnicas de replicação** | Ativa | Ativa | Ativa | Várias | Passiva | Várias |
| **Clientes legados** | Não | Não | Sim | Não | Não | Sim |
| **Vários grupos do mesmo serviço** | Não | NE | Não | Não | Não | Sim |
| **TF configurável** | NE | Sim | NE | Não | Não | Sim |
| **Recuperação de componentes** | NE | NE | NE | NE | NE | Sim |
| **Detecção de Falhas** | NE | NE | NE | Sim | Sim | Sim |
| **Recuperação de Falhas** | NE | Sim | NE | Sim | Sim | Sim |
| **Ordenação de mensagens** | Sim | Sim | Sim | Sim | Sim | Sim |

**Tabela 6.1:** Comparativo entre as diferentes infra-estruturas e *frameworks* de TF para Serviços Web

Por fim, Salas, J. et al. [Salas et al., 2006] especificam um *framework* para alta disponibilidade de Serviços Web denominado *WS-Replication*. O principal objetivo desse *framework* é a replicação de Serviços Web em uma WAN (*Wide Area Network*). Porém, os autores exibem alguns testes que descrevem que o framework também pode ser executado em uma LAN (*Local Area Network*). Esse *framework* utiliza replicação ativa para organizar seu grupo de réplicas do serviço ao longo de uma rede. Consumidores legados têm conhecimento de apenas uma das réplicas do WS-Replication, portanto, em caso de falha dessa réplica, o *framework* não consegue tolerar essa falha, justamente pelo fato das réplicas formarem um grupo único distribuído na rede. A WS-FTA pode ser utilizada perfeitamente tanto em uma LAN quanto em uma WAN. Em WANs, cada um dos grupos é instanciado em uma rede distinta, sendo que somente os líderes se comunicam pela WAN. Isso permite que a arquitetura ofereça tolerância a faltas para consumidores legados. Além disso, os consumidores podem ter falhas de grupo toleradas utilizando o WSFTA-Proxy.

Apesar das comparações realizadas entre a WSFTA e seus trabalhos relacionados, a tabela

6.1 permite visualizar aspectos que possibilitam uma melhor comparação entre a arquitetura e esses trabalhos.

## 6.9 Considerações sobre o capitulo

Nesse capítulo foi apresentada a implementação de referência da WSFTA com o objetivo de avaliar o impacto da utilização dessa arquitetura no tempo de resposta do serviço. Pode-se verificar que os resultados obtidos pela implementação foram satisfatórios, possibilitando o fornecimento de TF aos Serviços Web sem adição de um *overhead* elevado. Além disso, nesse capítulo foi realizada uma comparação com os trabalhos relacionados utilizando critérios que são desejáveis em um arquitetura de TF. Nessa comparação foi possível perceber que a WSFTA atende todos os requisitos, sendo mais flexível e robusta que os outros trabalhos presentes na literatura.

**Algoritmo 6.4** Adicionando mensagem para ordenação

**Require:** *message* {Requisição que será adicionada na janela de ordenação.}

**Require:** *timeoutOrder* {Se mais nenhuma requisição for recebida. o timeout é usado para iniciar a execução do algoritmo de ordenação}

**Require:** *nrMsgWindow* {Número de requisições da janela de ordenação. Se o timeout expirar, esse número é ignorado}

**Require:** *orderer* {Se a réplica é o sequenciador atual: *true*; senão: *false*}

**Require:** *checker* {Objeto responsável por disparar o algoritmo de ordenação quando o timeout expira}

1: **if** $verifyMessage(message) == true$ **then**
2: $idMessage = extractId(message);$
3: $idMember = extractIdMember(message);$
4: **if** $idMessage == (lastInsertedId(idMember) + 1)$ **then**
5: $insertMsgToOrder(message);$
6: $insertLastId(idMessage, idMember);$
7: **if** $checkThread! = null \,\&\&\, orderer$ **then**
8: $checkThread.cancel(true);$
9: **end if**
10: $verifyBuffer(idMember);$
11: **if** $(nrMsg >= nrMsgWindow) \,\&\&\, orderer$ **then**
12: $defineOrder(null);$
13: **else**
14: **if** *orderer* **then**
15: $checkThread = schedule(checker, timeoutOrder);$
16: **end if**
17: **end if**
18: **else**
19: **if** $idMessage > (lastInsertedId(idMember) + 1)$ **then**
20: $insertBuffer(message, member);$
21: **end if**
22: **end if**
23: **else**
24: $defineOrder(extractArrayOfIds(message));$
25: **end if**

**Algoritmo 6.5** Define ordem de execução das mensagens

**Require:** *arg* {Lista de Ids ordenados. Essa lista não existe se a réplica for o sequenciador atual}

**Require:** *orderer* {Se a réplica é o sequenciador atual: *true*; senão: *false*.}

**Require:** *isLeader* {Se a réplica é um líder: *true*; senão: *false*.}

**Require:** *moreGroupsExist* {Se existirem mais grupos: *true*; senão:*false*.}

**Require:** *isNextOrderer* {Se a réplica é o próximo sequenciador: *true*; senão:*false*.}

```
1: if orderer then
2:    size = getBuffer().size();
3:    arrayOfIds = getElementOrder();
4:    message = createMessage(arrayOfIds)
5:    if moreGroupsExist then
6:       sendMsgLeaders(message);
7:       setOrderer(false);
8:    end if
9:    sendMsgGroup(message);
10:   for i = 0 to (size − 1) do
11:      message = removeMessage(0);
12:      dispatchMessage(message);
13:   end for
14: else
15:   arrayOfIds = arg;
16:   if isLeader then
17:      message = createMessage(arrayOfIds);
18:      sendMsgGroup(message);
19:      if isNextOrderer then
20:         setOrderer(true);
21:      end if
22:   end if
23:   for i = 0 to (arrayOfIds.size() − 1) do
24:      message = getMessage(arrayOfIds(i));
25:      dispatchMessage(message);
26:   end for
27: end if
```

# Capítulo 7

# Conclusões e Trabalhos Futuros

Este trabalho apresentou uma proposta de arquitetura de software para fornecimento de tolerância a faltas a aplicações construídas utilizando Serviços Web, chamada WSFTA. A especificação da WSFTA visa preencher uma lacuna existente nas especificações de Serviços Web, que não definem mecanismos para provimento de tolerância a faltas às aplicações construídas utilizando essa tecnologia. O provimento de tais mecanismos advém da necessidade apresentada por um amplo conjunto de aplicações distribuídas, que possuem como requisito a confiança no funcionamento.

Observando as possíveis formações de grupo suportadas pela arquitetura e os resultados apresentados pelos experimentos, é possível afirmar que a divisão de réplicas em diferentes grupos permite uma maior confinamento do tráfego da rede, possibilitando que um menor número de mensagens sejam trocadas entre as réplicas do serviço. Além disso, as formações de grupos hierárquicos, isto é, grupos que possuem líderes, permitem ainda um fornecimento mais adequado de tolerância a faltas para consumidores legados, fazendo com que serviços incorporem propriedades de tolerância a faltas e forneçam essa vantagem para esses consumidores já existentes.

Apesar de experimentos em uma rede WAN não terem sido realizados, a possibilidade de utilização de mais de um grupos de réplicas do serviço e a constituição hierárquica desses grupos, torna factível a utilização da WSFTA em uma rede desse tipo. Nesse caso, grupos poderiam estar distribuídos em diferentes redes, com a vantagem da comunicação entre esses grupos ser realizada somente através de seus líderes. Isso permitiria uma redução de mensagens trocadas em uma rede de larga escala, fornecendo uma maior disponibilidade ao serviço sem sacrificar o seu tempo de resposta.

Outros trabalhos descritos na literatura, que visam prover tolerância a faltas a aplicações

estruturadas na forma de Serviços Web, apesar de serem capazes de prover uma maior confiabilidade às aplicações, apresentam algumas limitações em sua utilização. Em especial, destacam-se a inexistência de suporte a consumidores legados e a necessidade de efetuar alterações nos padrões adotados por Serviços Web. Além disso, essas estratégias são fortemente acopladas à tecnologia na qual foram desenvolvidas. Sendo assim, a WSFTA foi desenvolvida para ser independente de tecnologia, suportar consumidores legados e fornecer TF de forma simples e eficaz.

Como já citado anteriormente no capítulo 5, a arquitetura proposta é uma descrição abstrata de componentes para fornecimento de TF, possibilitando a utilização de diferentes tecnologias para a implementação desses componentes. Sendo assim, as limitações da WSFTA são diretamente derivadas de sua implementação. A primeira limitação do protótipo implementado é a baixa diversidade de algoritmos de ordenação, detecção de falha, comunicação de grupo, entre outros que não possibilita uma cobertura total das classes de faltas presentes na literatura. Atualmente, esse protótipo somente cobre faltas de *crash* e de valor. É importante ressaltar que o suporte a faltas de valor está baseado na ausência de réplicas maliciosas. Outra limitação é a definição estática da formação inicial do(s) grupo(s) do serviço. Além disso, é necessário implementar o processo de recuperação automatizada, assim como foi citado na seção 6.5.

## 7.1 Trabalhos Futuros

Na tentativa de tentar solucionar as deficiências presentes na implementação de referência da WSFTA e adicionar novas características a especificação da arquitetura, tem-se diversos trabalhos futuros a serem realizados tais como:

- a inserção de características de qualidade de serviço (QoS) para solucionar problemas tais como balanceamento de carga e tempo de resposta;

- a descoberta e formação dinâmica de novos grupos do mesmo serviço;

- a adaptação da arquitetura para fornecimento de tolerância a faltas para Serviços Web em grids computacionais;

- a implementação de diversos algoritmos de ordenação, detecção de falha, comunicação de grupo e recuperação de falhas para que a arquitetura possa realizar o tratamento dos mais variados tipos de falhas que possam ocorrer em sistemas distribuídos;

- a extensão da WSFTA para dar suporte a prevenção, remoção e previsão de faltas, necessárias para a cobertura de todos os atributos presentes em confiança no funcionamento.

## 7.2 Considerações Finais

Por fim, este trabalho possibilitou um aprendizado abrangente em diversos aspectos referentes a sistemas distribuídos, principalmente nos tópicos relacionados a tolerância a faltas, permitindo que os conhecimentos adquiridos fossem repassados para o projeto e concepção da arquitetura proposta. Além disso, partes dos resultados obtidos neste trabalho foram publicados em [Souza and Siqueira, 2007] e [Souza and Siqueira, 2008].

# Referências Bibliográficas

[Aghdaie and Tamir, 2002] Aghdaie, N. and Tamir, Y. (2002). Implementation and Evaluation of Transparent Fault-Tolerant Web Service with Kernel-Level Support. In *11th IEEE International Conference on Computer Communications and Networks*, Miami, Florida, USA. IEEE Computer Society.

[APACHE, 2008] APACHE (2008). Apache Axis. http://ws.apache.org/axis2/ Acessado em 29 de Janeiro de 2008.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basics Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 01(1):11–33.

[Ayala et al., 2002] Ayala, D., Browne, C., Chopra, V., Sarang, P., Apshankar, K., and McAllister, T. (2002). *Professional Open Source Web Services*. Wrox Press Ltd.

[Babaoglu and Schiper, 1995] Babaoglu, O. and Schiper, A. (1995). On Group Communication in Large-Scale Distributed Systems. In *ACM Operating Systems Review*, volume 29, pages 62–67.

[Baldoni et al., 1998] Baldoni, R., Bonamoneta, S., and Marchetti, C. (1998). Implementing highly-available www servers based on passive object replication. In *Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Saint-Malo, France. IEEE Computer Society.

[Bernes-Lee and Cailliau, 1990] Bernes-Lee, T. and Cailliau, R. (1990). WorldWideWeb: Proposal for a HyperText Project. http://www.w3.org/Proposal.

[Cerami, 2002] Cerami, E. (2002). *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI and WSDL*. O'Reilly.

[Chandra and Toueg, 1996] Chandra, T. D. and Toueg, S. (1996). Unreliable Failure Detectors For Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267.

[Coulouris et al., 2005] Coulouris, G., Dollimore, J., and Kindberg, T. (2005). *Distributed Systems: Concepts and Design*. Addison-Wesley.

[Dialani et al., 2002] Dialani, V., Miles, S., Moreau, L., Roure, D. D., and Luck, M. (2002). Transparent Fault Tolerance for Web Services Based Architectures. In *8th International 12 Europar Conference (EURO-PAR'02)*, Padeborn, Germany.

[Défago et al., 2005] Défago, X., Úrban, P., Hayashibara, N., and Katayama, T. (28 June-1 July 2005). Definition and Specification of Accrual Failure Detectors. In *International Conference of Dependable Systems and Networks (DSN 2005)*, pages 206–215. IEEE Computer Society.

[Ekwall et al., 2004] Ekwall, R., Pleisch, S., and Schiper, A. (2004). Token-based atomic broadcast using unreliable failure detectors. In *23rd Symposium on Reliable Distributed Systems (SRDS 2004)*, Florianópolis, Brazil.

[Erl, 2005] Erl, T. (2005). *Service-Oriented Architecture: Concepts, Technologies and Design*. Prentice Hall.

[Fang et al., 2007] Fang, C.-L., Liang, D., Lin, F., and Lin, C.-C. (2007). Fault Tolerant Web Services. *Journal of System Architecture*, 53:21–38.

[Freeman et al., 2004] Freeman, E., Freeman, E., Sierra, K., and Bates, B. (2004). *Head First Design Patterns*. O'Reilly Media, Inc., first edition.

[Grosso, 2001] Grosso, W. (2001). *Java RMI*. O' Reilly, first edition.

[Hayashibara et al., 2004] Hayashibara, N., Défago, X., Yared, R., and Katayama, T. (2004). The $\varphi$ Accrual Failure Detector. In *23rd Symposium on Reliable Distributed Systems (SRDS 2004)*, Florianópolis, Brazil.

[Hayden and Birman, 1996] Hayden, M. and Birman, K. (1996). Probabilistic Broadcast. Technical report, Dept od Computer Science, Cornell University, Ithaca, NY, USA.

[He, 2004] He, W. (2004). Recovery in Web Service Applications. In *IEEE International Conference on e-Technology, e-Commerce and e-Service*, pages 25–28, Taipei, Taiwan. IEEE Computer Society.

[Hoare, 2004] Hoare, C. (2004). *Communicating Sequential Processes*. Prentice Hall Internatio-nal. Eletronic version of *Communicating Sequential Processes* first published in 1985.

[Hopkins, 1978] Hopkins, A.L., J. S. T. I. L. J. (Oct. 1978). FTMPA highly reliable fault-tolerant multiprocess for aircraft. *Proceedings of the IEEE*, 66(10):1221–1239.

[JGroups, 2008] JGroups (2008). JGroups: A Toolkit for Reliable Multicast Communication. http://www.jgroups.org Acessado em 29 de Janeiro de 2008.

[Koren and Krishna, 2007] Koren, I. and Krishna, C. M. (2007). *Fault Tolerant Systems*. Elsevier, Inc.

[Lamport, 1978] Lamport, L. (1978). Time, Clocks, and The Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565.

[Laprie, 1995] Laprie, J.-C. (24-27 Oct 1995). Dependability of computer systems: concepts, limits, improvements. *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 2–11.

[Larrea et al., 2004] Larrea, M., Fernández, A., and Arévalo, S. (2004). On the Implementation of Unreliable Failure Detectors in Partially Synchronous Systems. *IEEE Transactions on Computers*, 53:815–828.

[Li et al., 2005] Li, W., He, J., Ma, Q., Yen, I.-L., Bastani, F., and Paul, R. (2005). A Framework to Support Survivable Web Services. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, volume 01, page 93b, Denver, Colorado, USA. IEEE Computer Society.

[Newcomer, 2002] Newcomer, E. (2002). *Understanding Web Services: XML, WSDL, SOAP, and UDDI*. Addison-Wesley Professional.

[OASIS, 2004a] OASIS (2004a). *Universal Description Discovery and Integration Specification 3.0.2*. OASIS. http://uddi.org/pubs/uddi/_v3.htm.

[OASIS, 2004b] OASIS (2004b). *WEb Services Reliable Message Specification Version 1.1*. OASIS. http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws/_reliability-1.1-spec-os.pdf.

[OMG, 2002] OMG (2002). *Common Object Request Broker Architecture: Core Specification Chapter 23*. OMG. http://www.omg.org.

[Orfali and Harkey, 1998] Orfali, R. and Harkey, D. (1998). *Client-Server Programming With Java and CORBA*. John Wiley Sons, second edition.

[Pullum, 2001] Pullum, L. L. (2001). *Software Fault Tolerance Techniques and Implementation*. Artech House, Inc.

[Roscoe, 2005] Roscoe, A. W. (2005). *The Theory and Pratice of Concurrency*. Prentice-Hall.

[Rubin and Brain, 1998] Rubin, W. and Brain, M. (1998). *Understanding DCOM*. Prentice Hall.

[Salas et al., 2006] Salas, J., Perez-Sorrosal, F., Patiño-Martinez, M., and Jímenez-Peris, R. (2006). WS-Replication: A framework for Highly Available Web Services. In *15th International World Wide Web Conference*, Edinburgh, Scotland.

[Santos et al., 2005] Santos, G. T., Lung, L. C., and Montez, C. (2005). FTWEB: A Fault Tolerant Infrastructure for Web Services. In *9th IEEE International EDOC Enterprise Computing Conference (EDOC'05)*, pages 95–105, Enschede, Netherlands. IEEE Computer Society.

[Shooman, 2002] Shooman, M. L. (2002). *Realiability of Computer Systems and Networks*. JOHN WILEY & SONS, INC.

[SOAPUI, 2008] SOAPUI (2008). SOAPUI: Open Source Web Services Testing Tool. http://www.soapui.org Acessado em 29 de Janeiro de 2008.

[Sousa et al., 2002] Sousa, A., Pereira, J., Moura, F., and Oliveira, R. (2002). Optimistic Total Order in Wide Area Networks. In *21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, page 190, Suita, Japan. IEEE Computer Society.

[Souza and Siqueira, 2007] Souza, J. L. R. and Siqueira, F. (2007). Uma Arquitetura para Tolerância a Faltas em Serviços Web. In *VIII Workshop de Testes e Tolerância a Falhas*, pages 233–246, Belém,PA,Brasil.

[Souza and Siqueira, 2008] Souza, J. L. R. and Siqueira, F. (2008). Providing Dependability for Web Services. In *23rd ACM Symposium on Applied Computing*, Fortaleza,CE,Brazil.

[Sun, 2006] Sun (2006). *Java API for XML Web Services*. Sun Microsystems, Inc. https://jax-ws.dev.java.net/jax-ws-ea3/docs/.

[SUN, 2006] SUN (2006). *Java Development Kit 1.6*. Sun Microsystems, Inc. http://java.sun.com/javase/6/docs/.

[Tanenbaum, 1995] Tanenbaum, A. S. (1995). *Distributed Operating Systems*. Prentice-Hall, 2 edition.

[Tindwel et al., 2001] Tindwel, D., Snell, J., and pavel Kunchenko (2001). *Programming Web Services with SOAP*. O'Reilly, first edition.

[Veríssimo and Rodrigues, 2001] Veríssimo, P. and Rodrigues, L. (2001). *Distributed Systems for System Architects*. Kluwer Academic Publishers.

[W3C, 2007a] W3C (2007a). *SOAP Specification Version 1.2*. W3C. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/.

[W3C, 2007b] W3C (2007b). *Web Services Description Language (WSDL) 2.0*. W3C. http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/.

[Wolfson et al., 1997] Wolfson, O., Jajodia, S., and Huang, Y. (1997). An Adaptative Data Replication Algorithm. *ACM Transactions on Database Systems*, 22(2):255–314.

[Wu, 1999] Wu, J. (1999). *Distributed System Design*. CRC Press, Florida, US.

[Xu and Bruck, 1998] Xu, L. and Bruck, J. (1998). Deterministic Voting in Distributed Systems Using Error-Correcting Codes. In *IEEE Transactions on Parallel and Distributed Systems*, volume 09, pages 813–824. IEEE Computer Society.

[Ye and Shen, 2005] Ye, X. and Shen, Y. (2005). A Middleware for Replicated Web Services. In *IEEE International Conference on Web Services (ICWS'05)*, pages 631–638, Orlando, Florida, USA. IEEE Computer Society.

# Apêndice A

# Descrição em CSP dos componentes da WSFTA

## A.1 Componente de Proxy - WSFTA-Proxy

O processo *SEND* é responsável por enviar as mensagens do consumidor para o serviço. Para isso, primeiramente esse processo lê a mensagem do seu canal de entrada *in*. Após isso, caso não ocorra falha no líder do grupo, a mensagem é marcada, enviada a esse líder e o processo SEND retorna ao seu estado inicial. Em caso de falha no evento *sendMsgLeader*, o líder é marcado como suspeito, a mensagem é reinserida no canal de entrada *in* para tentar ser enviada novamente.

O processo *RECEIVE* tem a responsabilidade de receber todas as mensagens. Assim que uma mensagem é recebida, o evento *recvMsg* é disparado. Se a mensagem recebida for uma resposta de mensagens enviadas anteriormente, o identificador dessa mensagem é verificado, e caso seja o identificador da mensagem que deve ser a próxima a ser entregue, o consumidor é notificado, a mensagem é entregue e o processo CHECKBUFFER é inicializado. Caso o identificador não seja o próximo a ser entregue, a mensagem é inserida no *buffer* e o processo *RECEIVE* retorna ao seu estado inicial.

O componente de *proxy* pode receber mensagens para atualizar sua lista de líderes dos grupos ativos. Recebendo essas mensagens o processo *RECEIVE* atualiza essa lista e retorna ao seu estado inicial. Por fim, o processo *CHECKBUFFER* possibilita checar se mensagens inseridas no *buffer* precisam ser entregues ao consumidor do serviço. Em caso positivo, essas mensagens serão entregues até que o *buffer* esteja vazio, ou não existam mensagens no *buffer* que devem ser entregues.

**Descrição A.1.** *Comportamento do componente WSFTA-Proxy*

$$
\begin{aligned}
SEND &= in?data : \{msg\} \rightarrow (markMsg \rightarrow sendMsgLeader \rightarrow SEND \,\Box \\
&\quad markLeaderSuspect \rightarrow in!data \rightarrow SEND) \\
RECEIVE &= recvMsg \rightarrow (verifyIdMsg \rightarrow (notifyClient \rightarrow deliveryMsg \rightarrow \\
&\quad CHECKBUFFER \,\Box\, insertMsgBuffer \rightarrow RECEIVE) \not< msg\ is\ reply \not> \\
&\quad updateLeaderList \rightarrow RECEIVE \,\Box\, RECEIVE) \\
CHECKBUFFER &= readFirstMsg \rightarrow notifyClient \rightarrow deliveryMsg \rightarrow CHECKBUFFER \,\Box \\
&\quad RECEIVE
\end{aligned}
$$

## A.2 Componente de Comunicação de Grupo - CG

O processo *GC* inicialmente deve receber informações, em seu canal de comunicação *in*, que indicam qual a ação que deve ser tomada e, como conseqüência dessa ação, para qual(is) membro(s) do grupo uma mensagem deve ser enviada. Esses dados que são lidos pelo canal de comunicação *in* são armazenados nas variáveis *action* e *data*. Os possíveis valores que podem ser assumidos pela variável *action* são especificados pelo conjunto $\mathbb{A}$. O valor *all*, presente nesse conjunto, tem a finalidade de representar todos os membros do grupo, para que uma mensagem possa ser enviada para todos os membros ativos. Além disso, cada membro é representado separadamente, sendo esses membros elementos do conjunto $\mathbb{A}$ definidos pelo conjunto $\mathbb{M}$. Já a variável *data* representa o dado fornecido ao componente GC. Esse dado pode ser tanto uma mensagem (representada por *msg*) quanto um número do conjunto dos naturais ($\mathbb{N}$) que representa anterior de uma mensagem.

A primeira verificação feita pelo processo *GC* é a identificação do dado fornecido a ele. Caso esse dado seja uma mensagem, a próxima ação a ser tomada deve ser a identificação do valor em *action* que define se a mensagem deve ser enviada para todo o grupo ou só para um membro específico. Se for para enviar uma mensagem ao grupo, primeiramente o evento *sendMsgGroup* é disparado. Em seguida, é escrito no canal de comunicação *out* um número de identificação que representa unicamente o envio dessa mensagem ao grupo. Após isso, o processo *GC* retorna automaticamente ao seu estado inicial. Caso a mensagem deva ser enviada somente para um membro

**Descrição A.2.** *Comportamento do componente GC*

$$
\begin{aligned}
GC &= in?action : \mathbb{A} \rightarrow in?data : \mathbb{S} \rightarrow ((sendMsgGroup \rightarrow out!genNumber() : \mathbb{N} \rightarrow \\
&\quad GC) \not< action\, is\, all \not> (sendMsgMember \rightarrow out!genNumber() : \mathbb{N} \rightarrow GC)) \\
&\quad \not< data\, is\, msg \not> (searchMsgBuffer \rightarrow retrieveMsgs \rightarrow out!msgArray \rightarrow \\
&\quad GC \rightarrow out!null \rightarrow GC) \\
HANDLER &= recvMsg \rightarrow (insertReplyBuffer \rightarrow HANDLER \,\Box\, notifyReceive \rightarrow out!msg \rightarrow \\
&\quad HANDLER) \\
Onde &: \\
\mathbb{M} &= \{member_n \mid n \in \mathbb{N}\} \\
\mathbb{A} &= \{all\} \cup \mathbb{M} \\
\mathbb{S} &= \{msg\} \cup \mathbb{N}
\end{aligned}
$$

específico, o evento inicial *sendMsgMember* é disparado, escrevendo, em seguida, um identificador no canal de comunicação *out* que represente o envio dessa mensagem. Posteriormente, o processo *GC* retorna ao seu estado inicial.

Nota-se no comportamento descrito até o presente momento, que o componente *GC* não fica bloqueado à espera das respostas dos demais membros do grupo. Para que o sistema possa tomar conhecimento da(s) resposta(s) de uma mensagem enviada anteriormente, o número de identificação que representa essa mensagem deve ser fornecido ao componente. Nesse caso, após obter esse número, que será utilizado para resgate das respostas, o primeiro evento que é disparado é o *searchMsgBuffer* que irá verificar se existe(m) mensagem(ns) no *buffer* para aquele número de identificação. Caso não exista(m) mensagem(ns) disponível(is), o processo *GC* escreve *null* no canal de comunicação *out* e retorna ao seu estado inicial. Caso exista(m), o evento *retrieveMsgs* é disparado. Se o identificador for de uma mensagem enviada para o grupo, esse evento verifica no *buffer* de mensagens se todos os membros ativos do grupo já enviaram suas respostas e, caso essas respostas estejam disponíveis, o processo *GC* as escreve no canal *out* antes de retornar ao seu estado inicial. Esse processo é similar para uma mensagem enviada a um membro específico do grupo.

Durante a descrição do processo *GC*, instâncias do processo *HANDLER* são executadas para receber as mensagens enviadas pelas demais réplicas. Essas mensagens podem ser de dois tipos: respostas de mensagens síncronas ou mensagens assíncronas enviadas pelas demais réplicas do

grupo. Assim que uma mensagem é recebida, o evento *recvMsg* é disparado. Caso a mensagem recebida seja uma resposta, a instância do processo *HANDLER* insere essa mensagem no *buffer* e retorna ao seu estado inicial. Caso contrário, o processo *HANDLER* notifica a recepção da mensagem (*notifyReceive*), escreve a mensagem em seu canal de comunicação *out* e retorna ao seu estado inicial.

## A.3 Componente de Ordenação de Mensagens - MO

O processo *ORDER* é o que inicia a execução desse componente. Para um líder, o processo *ORDER* deve executar o processo *LHANDLER*. Para isso, o canal de comunicação *in* é utilizado na recepção de um número natural que representa o total de mensagens que precisam ser agrupadas para ordenação. Nota-se no processo *ORDER* que esse número é passado como parâmetro ao processo *LHANDLER*. Esse parâmetro é identificado pela variável *nrMsg*. Após isso, o processo *LHANDLER* verifica se a variável *n*, que identifica o número de vezes que o processo *LHANDLER* foi executado, é menor que o *nrMsg*. Se *nrMsg* for maior que *n*, o processo *LHANDLER* lê a mensagem que precisa ser ordenada do canal de comunicação *in*, sendo essa mensagem representada pela variável *data*. A variável *data* pode assumir dois tipos de mensagens. Esses tipos são definidos pelo conjunto de entrada do canal *in*. Quando *data* é do tipo *msg*, significa que essa mensagem foi recebida diretamente por um consumidor do serviço. Então o evento *markMsg* é disparado. Após marcar a mensagem o evento *fwdLeaders* encaminha essa mensagem para os outros líderes. Já quando é do tipo $msg'$ significa que foi encaminhada por outro líder e já possui uma marcação. Após enviar a mensagem para os demais líderes, ou receber uma mensagem marcada previamente por um outro líder, o processo *LHANDLER* executa o processo *LORDER*.

Quando *nrMsg* é menor que *n*, o processo *LORDER* é executado com parâmetros que indicam que a ordem das mensagens deve ser definida. É importante notar que o processo *LHANDLER* pode executar o processo *LORDER* para definir a ordem das mensagens no início de sua execução. Essa característica é necessária quando o *nrMsg* ainda não foi alcançado e os consumidores do serviço não estão enviando mais mensagens, aguardando somente a execução das mensagens que foram enviadas anteriormente. Assim que o processo *LORDER* é executado, é verificado se o valor da variável *data*, passada como parâmetro, é *true*. Em caso positivo, esse processo deve definir a ordem de execução das mensagens já armazenadas em seu *buffer*. Para isso, primeiramente o evento *retrieveMsg* é executado. Esse evento resgata todas as mensagens armazenadas no *buffer*

**Descrição A.3.** *Comportamento do componente MO*

$$
\begin{aligned}
ORDER &= LHANDLER_{(0)}(in?nrMsg : \mathbb{N}) \,\square \\
&\quad RHANDLER(in?data : \{msg, msg'\}) \\
LHANDLER_{(n)}(nrMsg) &= (((in?data : \{msg, msg'\} \rightarrow LORDER_{(n)}(nrMsg, data)) \\
&\quad \not\prec data\, is\, msg' \not\succ (markMsg \rightarrow fwdLeaders \rightarrow \\
&\quad LORDER_{(n)}(nrMsg, data))) \not\prec n < nrMsg \not\succ \\
&\quad LORDER_{(n)}(nrMsg, true)) \,\square\, LORDER_{(n)}(nrMsg, true) \\
LORDER_{(n)}(nrMsg, data) &= (saveMsg \rightarrow LHANDLER_{(n+1)}(nrMsg)) \not\prec data\, is\, not\, true \not\succ \\
&\quad (retrieveMsgs \rightarrow clearBuffer \rightarrow defineOrder \rightarrow fwdMsgGroup \rightarrow \\
&\quad dispatchMessages \rightarrow ORDER) \\
RHANDLER(data) &= verifyOrder \rightarrow ((insertMsgBuffer \rightarrow ORDER) \,\square \\
&\quad (dispatchMessage \rightarrow ORDER) \,\square \\
&\quad (removeFirstMsgBuffer \rightarrow dispatchThisMessage \\
&\quad \rightarrow RHANDLER(data))
\end{aligned}
$$

até o presente momento. Em seguida o evento *clearBuffer* limpa esse *buffer* e dispara o evento *defineOrder* responsável pela execução do algoritmo de ordenação que efetivamente definirá a ordem de execução das mensagens. Após a execução desse evento, todos os líderes tem conhecimento da seqüência de execução das mensagens e o evento *fwdGroup* é disparado. Nesse evento, as mensagens são encaminhadas para as demais réplicas do grupo, sendo entregue ao serviço na execução do evento *dispatchMessages*. Após serem entregues ao serviço, o processo *ORDER* é novamente executado para dar início a um novo ciclo de ordenação de mensagens.

Em réplicas que não fazem papel de líder de grupo, o processo *ORDER* executa o processo *RHANDLER*. Esse processo recebe como parâmetro a mensagem lida do canal de comunicação *in*. Essa mensagem é fruto da execução do evento *fwdGroup*, do processo *LORDER*, na réplica líder. Assim que o processo *RHANDLER* recebe a mensagem, representada pela variável *data*, o evento *verifyOrder* é disparado. Esse evento verifica em que ordem a mensagem recebida deve ser entregue ao serviço que utiliza a WSFTA. Após essa verificação, três situações podem ocorrer: a mensagem ser inserida no *buffer* para entrega posterior, a mensagem ser entregue imediadamente ao serviço, ou a primeira mensagem do *buffer* ser entregue ao serviço. Na primeira situação,

a mensagem representada pela variável *data* possui o número de identificação maior do que o número de identificação da próxima mensagem que deve ser entregue ao serviço. Logo, o evento *insertMsgBuffer* é disparado e o processo *ORDER* volta a ser executado. Na segunda situação, o número de in dentificação da mensagem representada por *data* é igual ao número de identificação da próxima mensagem que deve ser entregue ao serviço. Sendo assim, o evento *dispatchMessage* realiza essa entrega e em seguida dispara a execução do processo *ORDER*. Por fim, se o número de identificação da primeira mensagem que está no *buffer* for igual ao número de identificação da próxima mensagem que deve ser entregue ao sistema, essa mensagem é removida do *buffer* (evento *removeFirstBuffer*), entregue ao sistema (*dispatchThisMessage*) e a mensagem representada por *data* é passada novamente como parâmetro para uma nova execução do processo *RHANDLER*.

## A.4 Processo de detecção de falhas

O comportamento dos componentes de detecção de falha, RFD e GFD, possuem similaridades em alguns aspectos. Logo, foram elaborados três processos responsáveis pela descrição do comportamento desses dois componentes. No GFD, o método de detecção de falha entre os grupos do serviço é baseado na formação de um anel virtual de detecção. Se a descrição do componente RFD for verificada, será notada que a forma de detecção de falha de RFDs dentro de um mesmo grupo também é efetuada com base em um anel virtual de detecção. Logo, existe um processo chamado de *FDRING*, que especifica o comportamento da detecção imposta por esse anel virtual.

O processo *FDRING* recebe como parâmetro o número máximo de falhas que podem ocorrer com o detector vizinho que está sendo monitorado. O primeiro evento que ocorre nesse processo é o início do intervalo de monitoração, representado por *startMInterval*. Após esse evento o evento *sendMsgNextFD* é disparado. Esse evento envia uma mensagem, do tipo "Are you alive ?", para o detector de falha monitorado. Logo após esse envio o evento *startTimeout* é disparado, representando o início do *timeout* imposto para resposta do detector de falha monitorado. Caso o detector de falha monitorado responda antes do término do *timeout*, o evento *recvReplyNextFD* é disparado, simbolizando a chegada da mensagem de resposta. Logo após esse evento, o *timeout* é cancelado, aguardando em seguida o término do intervalo de monitoração (representado pelo evento *finishMInterval*). Assim que o intervalo de monitoração é finalizado, o processo *FDRING* retorna ao seu estado inicial, pronto para começar mais um ciclo de monitoração.

Se o detector de falha monitorado não enviar uma mensagem de resposta antes do *timeout* terminar, o evento *finishTimeout* é disparado. Logo após o disparo, é verificado se o número de

**Descrição A.4.** *Comportamento dos componentes de detecção de falha RFD e GFD*

$$
\begin{aligned}
FD(nrFails) &= startMInterval \rightarrow (fdMessage \rightarrow TIMEOUT(nrFails) \,\Box \\
&\quad componentMessage \rightarrow TIMEOUT(nrFails)) \\
TIMEOUT(nrFails) &= startTimeout \rightarrow (componentMessage \rightarrow cancelTimeout \rightarrow \\
&\quad FD(nrFails) \,\Box\, finishTimeout \rightarrow ((detect \rightarrow notify \rightarrow \\
&\quad cancelMInterval \rightarrow Skip) \not< (nrfails - 1) \leqslant 0 \not> \\
&\quad (finishMInterval \rightarrow FD(nrFails - 1))) \\
FDRING(nrFails) &= startMInterval \rightarrow sendMsgNextFD \rightarrow startTimeout \rightarrow \\
&\quad (recvReplyNextFD \rightarrow cancelTimeout \rightarrow finishMInterval \rightarrow \\
&\quad FDRING(nrFails) \,\Box\, finishTimeout \rightarrow ((detect \rightarrow notify \rightarrow \\
&\quad (cancelMInterval \rightarrow restoreRing \rightarrow FDRING(in?nrFails : \mathbb{N}) \\
&\quad \Box\, cancelMInterval \rightarrow Skip)) \not< (nrFails - 1) \leq 0 \not> \\
&\quad (finishMInterval \rightarrow FDRING(nrFails - 1))))
\end{aligned}
$$

falhas permitidas chegou a um valor igual ou inferior a zero. Em caso positivo, o evento *detect* é disparado. Esse evento simboliza a detecção de falha do detector monitorado. Após detectada a falha, o evento *notify* notifica as demais réplicas (líderes para GFDs e réplicas do grupo para RFDs) que a réplica que esse componente monitora falhou. A partir dessa notificação o mecanismo de recuperação inicia seu funcionamento, e o intervalo de monitoração é cancelado para que o anel virtual de detecção possa ser restabelecido. Essa restauração do anel virtual é realizada pelo evento *restoreRing*. Após reconstruir o anel, o número máximo de falhas é obtido com auxílio do canal de comunicação *in*, permitindo que o processo *FDRING* inicie novamente o ciclo de monitoração. Para GFDs, caso após a execução do evento *notify* não exista mais o anel de detecção, ou seja, no presente momento existe somente um grupo ativo, o intervalo de monitoração é cancelado e o processo *FDRING* é finalizado. Se o número de falhas permitidas não atingir o valor zero, somente é necessário aguardar o término do intervalo de monitoração e iniciar novamente o processo *FDRING* com o *nrFails* com decréscimo de uma unidade.

Existem outros dois processos que têm a tarefa de detectar falhas no serviço propriamente dito. Esses processos são o *FD* e o *TIMEOUT*. Esse tarefa de detecção de falha do serviço é realizado pelos componentes RFDs localizados em cada uma das réplicas. O processo *FD* é o

ponto de início de todos os eventos que envolvem a tarefa de detecção de falha do serviço. Nesse processo, o primeiro evento disparado é o *startMInterval*, que inicia o intervalo de monitoração do serviço. A partir desse início, o detector de falha pode operar em dois modos: o modo *PUSH* ou o modo *PULL*. No modo *PUSH*, o serviço monitorado envia uma mensagem, representada pelo evento *componentMessage*, ao detector de falha para logo após dar início à execução do processo *TIMEOUT*. Já no modo *PULL*, o detector envia uma mensagem ao serviço monitorado (evento *fdMessage*) e aguarda a resposta dessa mensagem, que é tratada na execução do processo *TIMEOUT*. Percebe-se que o processo *TIMEOUT* é executado tanto no modo *PUSH* quanto no modo *PULL*.

Assim que o processo *TIMEOUT* entra em execução, o evento *startTimeout* é disparado. Se o serviço enviar uma mensagem ao detector antes que o *timeout* termine, o evento *componentMessage* é disparado. Após esse evento, o *timeout* é cancelado (evento *cancelTimeout*), e é aguardado o término do intervalo de monitoração, representado pelo evento *finishMInterval*. Assim que esse intervalo termina, o processo *FD* é novamente executado, dando início a um novo ciclo de monitoração. Porém, se o *timeout* se esgotar e o serviço não enviar nenhuma mensagem ao detector, o evento *finishTimeout* é disparado. Se o número de falhas permitidas não atingir um valor igual ou inferior a zero, o término do intervalo de monitoração é aguardado para que, logo em seguida, o processo *FD* seja executado com o parâmetro $nrFails - 1$. Já se o número de falhas permitidas for menor ou igual a zero, o evento *detect* é disparado, simbolizando que a falha do serviço foi detectada. Em seguida, o evento *notify* notifica todas as outras réplicas e o mecanismo de recuperação que o serviço falhou, cancelando na seqüência o intervalo de monitoração e finalizando a execução do processo *TIMEOUT*. Nesse caso, a detecção de falha do serviço volta ao seu funcionamento normal assim que o serviço tiver sido recuperado.

## A.5 Componente de LOG - LOG

O comportamento do componente de LOG é descrito pelo processo *LOG*. Inicialmente esse processo recebe como parâmetro um número natural, representado pela variável *nrMsg*, que especifica o intervalo total de mensagens para que o evento de *checkpoint* seja disparado. Sendo assim, caso o processo de *LOG* tenha iniciado sua execução, ou seja, o valor de $n$ seja igual a $0$, o primeiro evento que deve ocorrer é o evento de *checkpoint*. Esse evento é representado pelo símbolo ⓒ. Após realizar o armazenamento de estado do serviço, o processo de *LOG* é novamente executado, somando uma unidade no valor de $n$ que indica o índice do processo *LOG* em sua pilha de execu-

**Descrição A.5.** *Comportamento do componente LOG*

$$LOG_{(n)}(nrMsg) = (© \rightarrow LOG_{(n)}(nrMsg)) \not< n == 0 \not> (in?data : msg \rightarrow (© \rightarrow remAllMsgBuffer \rightarrow LOG_{(n+1)}(nrMsg)) \not< (n\, div\, nrMsg) == 0 \not> (saveMsgInBuffer \rightarrow LOG_{(n+1)}(nrMsg)))$$

ção. Se o número representado por $n$ for dividido pelo número representado por *nrMsg* e, o resto dessa divisão for diferente de 0, o evento *saveMsgBuffer* é disparado. Esse evento salva a mensagem, lida no canal do canal de comunicação *in*, no *buffer* persistente que armazena as mensagens do intervalo representado pela variável *nrMsg*. Em seguida, o processo *LOG* é executado com seu índice $n$ acrescido de uma unidade. Caso o resto da divisão seja igual a 0, o processo *LOG* dispara o evento de *checkpoint*, remove todas as mensagens previamente armazenadas e inicia novamente sua execução.

## A.6 Componente de Recuperação de Falhas - REC

O comportamento do componente REC é descrito pelos processos *RECOVERY*, *RECMEMBER*, *RECSTATE*, *DISCOVERY*, *ACTIVATE*, *RECGROUP* e *FORMGROUP* descritos na definição A.6. Esses são os processos responsáveis pela recuperação tanto de réplicas individuais quanto de grupos inteiros dessas réplicas. Essa recuperação sempre é iniciada pelo processo *RECOVERY*. Assim que uma falha é detectada e notificada, o evento *detect* do processo *RECOVERY* é disparado. Após esse evento ter sido disparado, se essa falha for de réplicas individuais o processo *RECMEMBER* é executado. Caso a falha seja de um grupo de réplicas, o canal de comunicação *in* permite a leitura da quantidade de membros que precisam ser criados e passa esse valor como parâmetro do processo *RECGROUP* que será executado.

No processo *RECMEMBER* o primeiro evento disparado na sua execução é o *isolateMember*. Esse evento retira a réplica faltosa do grupo para que a mesma possa ser recuperada sem interferir no resto do grupo. Caso essa réplica seja o líder do grupo, uma eleição é realizada para escolher a nova líder. Quando a nova líder é escolhida, todos os membros do grupo são notificados e o evento que inicia a recuperação (*startRecovery*) é disparado. Já quando a réplica faltosa (que foi isolada do grupo) não é a líder, o evento *startRecovey* é automaticamente disparado. Após esse evento, o processo *RECSTATE* é executado para tentar recuperar o estado da réplica faltosa. Para serviços

**Descrição A.6.** *Comportamento do componente REC*

$$
\begin{aligned}
RECOVERY &= detect \rightarrow (RECMEMBER \,\Box\, RECGROUP(in?nrMembers))\\
RECMEMBER &= isolateMember \rightarrow (election \rightarrow notify \rightarrow startRecovery \rightarrow\\
&\quad RECSTATE \,\Box\, startRecovery \rightarrow RECSTATE)\\
RECGROUP(nrMembers) &= startRecovery \rightarrow initializeGroup \rightarrow\\
&\quad FORMGROUP(nrMembers)\\
FORMGROUP(nrMembers) &= (DISCOVERY;\ FORMGROUP(nrMembers - 1))\\
&\quad \not< nrMembers > 0 \not> (election \rightarrow (notify \rightarrow finishRecovery\\
&\quad \rightarrow RECOVERY \,\Box\, synchronizeState \rightarrow notify \rightarrow\\
&\quad finishRecovery \rightarrow RECOVERY))\\
RECSTATE &= (getLastState \rightarrow synchronizeState \rightarrow (DISCOVERY \,\Box\\
&\quad ACTIVATE)) \,\Box\, (restartMember \rightarrow (DISCOVERY \,\Box\\
&\quad ACTIVATE))\\
DISCOVERY &= searchPoints \rightarrow selectPoint \rightarrow createMember \rightarrow RECSTATE\\
ACTIVATE &= insertMember \rightarrow (Skip \,\Box\, finishRecovery \rightarrow RECOVERY)
\end{aligned}
$$

*stateful*, o processo *RECSTATE* resgata o último estado do serviço (*getLastState*) e tenta sincronizar esse estado com o estado das demais réplicas (*synchronizedState*). Se a sincronização for realizada com sucesso, o processo *ACTIVATE* é executado. Caso contrário, será necessária a criação de uma nova réplica e então o processo *DISCOVERY* precisará ser executado. Para serviços *stateless*, o processo *RECMEMBER* tenta inicializar novamente o serviço (*initializeMember*). Em caso de sucesso, o processo *ACTIVATE* é executado. Em caso de falha na inicialização, o processo *DISCOVERY* é o escolhido.

O processo *ACTIVATE* é responsável pela inserção da réplica no grupo. Seu evento inicial é o *insertMember* responsável justamente pela essa tarefa de inserção. Após essa inserção, o processo *ACTIVATE* pode finalizar sua execução com sucesso ou disparar o evento que finaliza o processo de recuperação (*finishRecovery*) e executar novamente o processo *RECOVERY* para que novas recuperações possam ser realizadas. Já o processo *DISCOVERY* é responsável por descobrir um local para criação de novas réplicas. Para isso, o evento *searchPoint* é disparado. Após encontrar

locais para recuperação, o evento *selectPoint* seleciona um desses locais para que a réplica possa ser criada. Essa criação é realizada pelo evento *createMember*, deixando a réplica pronta para a uma sincronização de estado ou uma inicialização que podem ser realizadas, dependendo do tipo de serviço, pelo processo *RECSTATE* que é executado na seqüência.

No processo *RECGROUP*, após iniciar o processo de recuperação (*startRecovery*), o grupo que será criado para substituir o grupo faltoso é inicializado (*initializeGroup*). Após essa inicialização o processo *FORMGROUP* é executado. Esse processo é um processo recursivo que é executado até que o grupo todo esteja em operação. Para criação de cada um dos membros do grupo, o processo *DISCOVERY* é executado. Assim que todos os membros do grupo forem criados, uma eleição que escolherá o líder do grupo é disparada. Caso o grupo seja de serviços *stateless*, os demais grupos ativos são notificados da presença desse novo grupo, a tarefa de recuperação é finalizada e o processo *RECOVERY* é executado novamente para que possa realizar novas recuperações.

# Apêndice B

# Classes da WSFTA

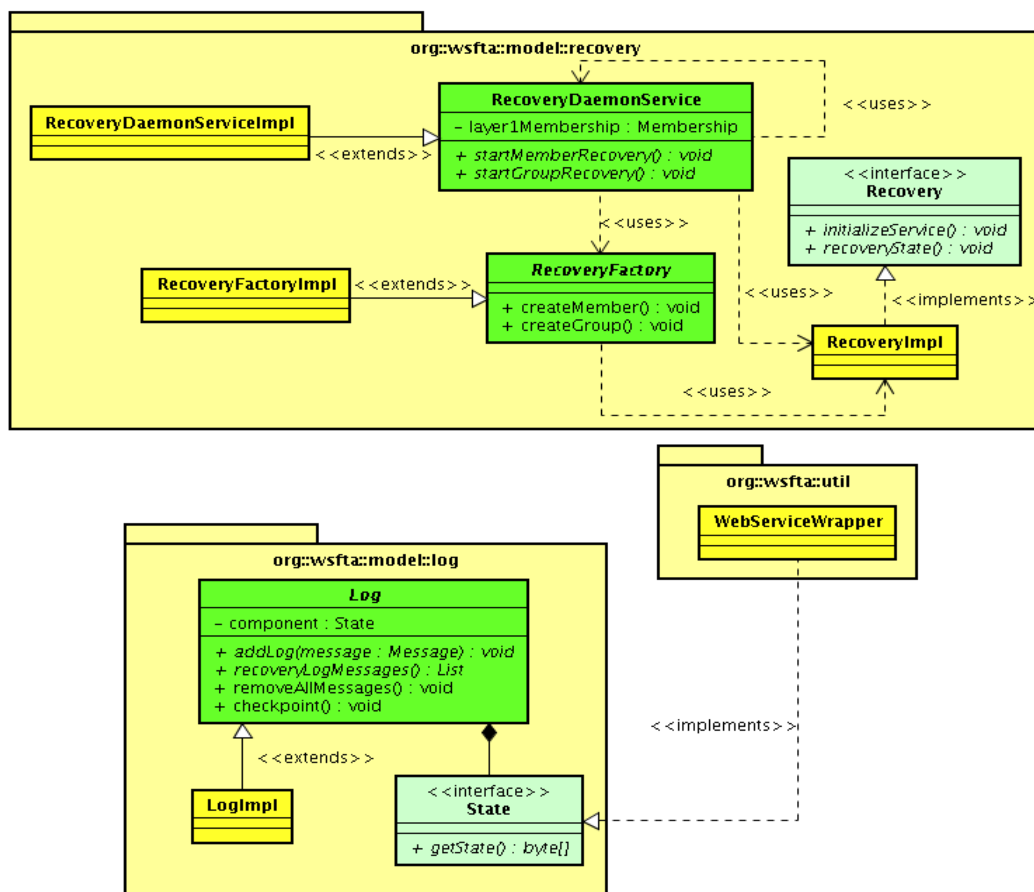



**Figura B.1:** Pacotes com as classes de recuperação e log

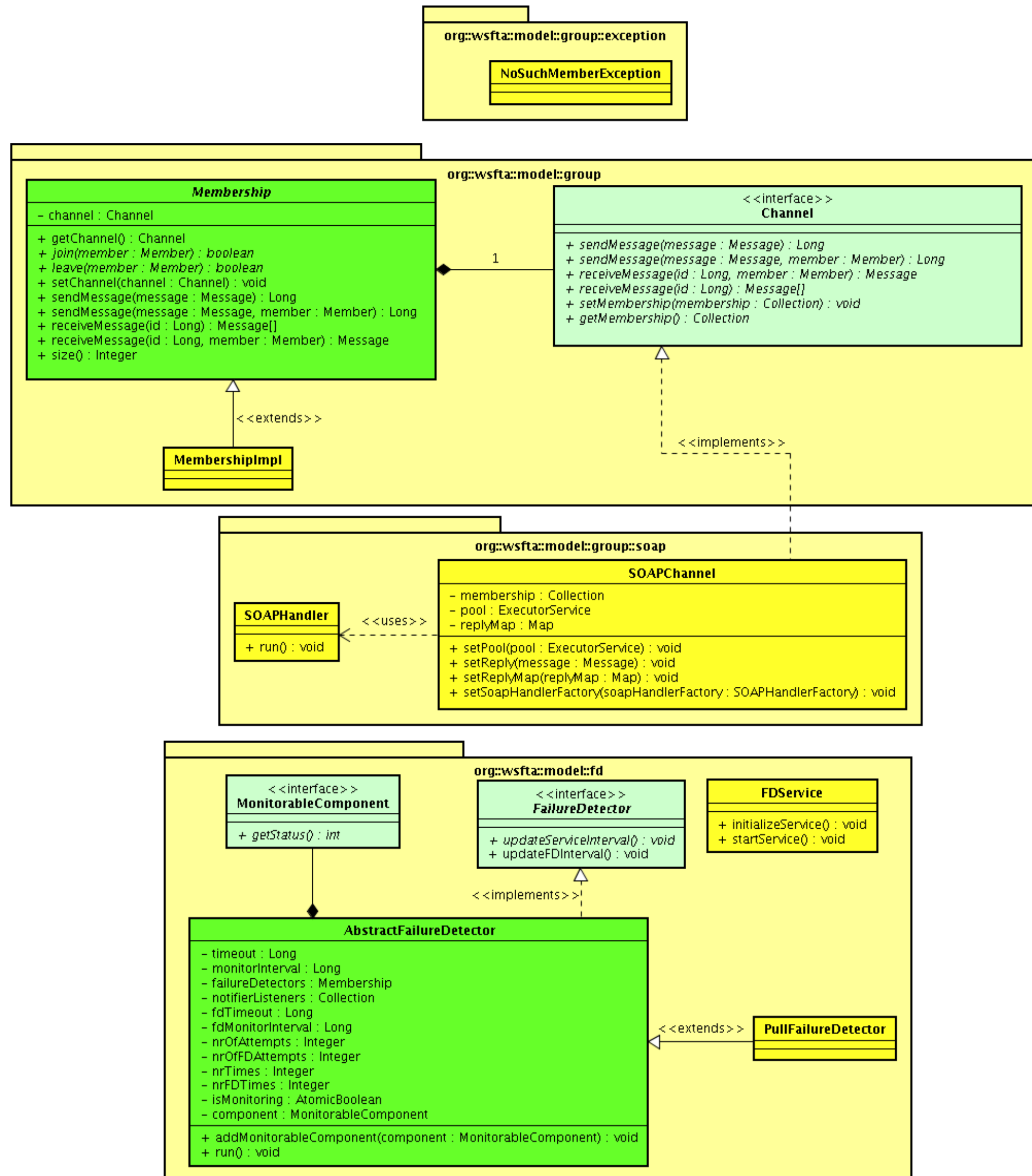



**Figura B.2:** Pacotes com as classes para comunicação de grupo e detecção de falha

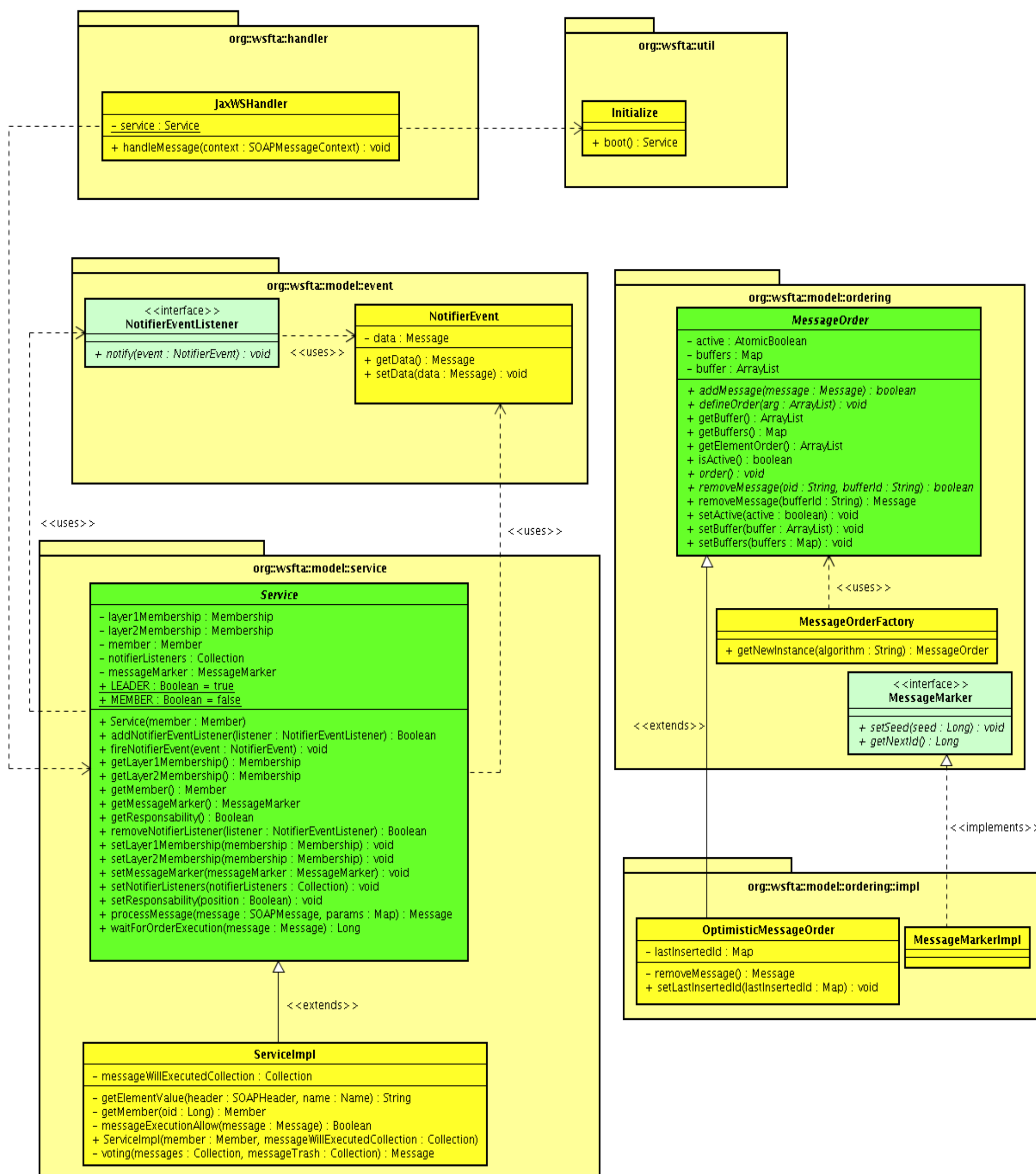



**Figura B.3:** Pacotes com as classes de ordenação, notificação de eventos e *handlers*