PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ
CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

GIULIANA SANTOS

FTWEB: UMA INFRA-ESTRUTURA FLEXÍVEL PARA TOLERÂNCIA A FALTAS EM ARQUITETURAS ORIENTADAS A SERVIÇOS

CURITIBA
2005

GIULIANA SANTOS

FTWeb: Uma Infra-Estrutura Flexível para Tolerância a Faltas em Arquiteturas Orientadas a Serviços

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

Área de Concentração: *Sistemas Distribuídos*

Orientador: Prof. Dr. Lau Cheuk Lung

CURITIBA

2005

SANTOS Giuliana
FTWEB: Uma infra-estrutura flexível para tolerância a faltas em arquiteturas orientadas a serviços. Curitiba, 2005.

Dissertação de Mestrado – Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática Aplicada.

1. Arquitetura orientada a serviços 2. Web Services 3.Tolerância a faltas 4. CORBA.
I. Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e de Tecnologia. Programa de Pós-Graduação em Informática Aplicada II-t

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

ATA DA SESSÃO PÚBLICA DE DEFESA DE DISSERTAÇÃO DE MESTRADO
DO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA APLICADA
DA PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ

DEFESA DE DISSERTAÇÃO Nº 02/2005

Aos 30 dias do mês de maio de 2005 realizou-se a sessão pública de defesa da dissertação **"FTWeb: Uma Infra-Estrutura Flexível para Tolerância a Fltas em Arquiteturas Orientadas a Serviços"**, apresentada por **Giuliana Teixeira dos Santos** como requisito parcial para a obtenção do título de **Mestre em Informática Aplicada**, perante uma Banca Examinadora composta pelos seguintes membros:

	assinatura	parecer (aprov/ reprov.)
Prof. Dr. Lau Cheuk Lung PUCPR (orientador)	Lau Cheuk Lung	aprovado
Prof. Dr. Carlos Alberto Maziero PUCPR	Carlos Maziero	aprovado
Prof. Dr. Joni da Silva Fraga UFSC	Joni da Silva Fraga	aprovado
Prof. Dr. Luiz Nacamura Junior CEFET - PR	Luiz Nacamura Jr	aprovado

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

Prof. Dr. Díbio Leandro Borges,
Diretor do PPGIA PUCPR

Programa de Pós-Graduação em Informática Aplicada
PPGIA
PUCPR

Aos meus pais e ao meu marido, vocês são a razão de tudo.

Agradecimentos

Aos meus pais, Rozemeri dos Santos e Laurentino dos Santos, que foram os meus primeiros professores e que com muito empenho e carinho me concederam muito além do que tiveram.

Ao meu marido, Marcello Veronese, cujo apoio, amizade e amor foram essenciais para a conclusão deste trabalho.

A todos os professores principalmente ao meu orientador, Lau Cheuk Lung, por estar sempre presente e disposto a apontar novos horizontes, visando apenas o sucesso desta empreitada.

Aos colegas do HSBC, do mestrado e a todos aqueles que compartilharam deste sonho e me incentivaram a seguir em frente.

SUMÁRIO

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

RESUMO

A arquitetura de serviços web surgiu como uma resposta à busca da interoperabilidade entre aplicações. Nos últimos anos existe um interesse crescente em executar na Internet aplicações com requisitos de alta disponibilidade e confiabilidade, contudo as tecnologias associadas a essa arquitetura ainda não oferecem suporte adequado a esses requisitos. Atualmente não existem especificações padrão que tratem a tolerância a faltas nos serviços web.

O presente trabalho se situa neste contexto, provê uma nova camada de software que atua como um *proxy* entre as requisições do cliente e os serviços nos provedores. O foco principal é a utilização da técnica de replicação ativa para alcançar a tolerância a faltas em arquiteturas orientadas a serviços. O modelo proposto é baseado em uma infra-estrutura denominada de *FTWeb* que permite que os serviços estejam replicados e distribuídos sobre a Internet. Esta infra-estrutura possui componentes responsáveis por invocar concorrentemente as réplicas do serviço, aguardar o processamento, analisar as respostas processadas, e retornar a resposta ao cliente. Estes componentes são baseados nos modelos e conceitos do padrão FT-CORBA da OMG para o desenvolvimento de aplicações distribuídas e tolerantes a faltas. O objetivo desta abordagem é prover tolerância nas seguintes classes de faltas: parada, omissão e valor. Devido a flexibilidade da infra-estrutura *FTWeb* é possível a sua utilização na implementação de diferentes técnicas de replicação, tais como passiva (quente e fria) e semi-ativa. A implementação da infra-estrutura proposta e os testes realizados demonstraram a viabilidade desta solução.

Palavras chave: Serviços Web, Tolerância a Faltas, FT-CORBA.

ABSTRACT

The web services architecture came as an answer to applications interoperability problem. In recent years, there has been a growing interest in deploying on the Internet applications with high availability and reliability requirements. However, the technologies associated to this architecture still do not deliver adequate support to this requirement**.** Currently there are no standard specifications dealing with fault tolerance in web services.

The present work is inserted in this context and provides a new software layer that acts as a proxy between the client request and suppliers services. The main goal is to guarantee transparent fault tolerance for the customer through active response technique. We are proposing the *FTWeb* infrastructure for tolerance of faults in web services. This infrastructure features a set of components and services, some based on OMG's FT-CORBA standard's models and concepts, for the development of fault tolerant web applications. The *FTWeb* infrastructure has components responsible for concurrently executing requests to service replicas, wait for processing, analyze the responses, and return them to the client. FT-Web supports the use of the active replication technique in order to obtain fault tolerance in service-oriented architectures. The objective of this approach is to provide tolerance in the following kind of faults: stop, omission and value. Due the *FTWeb* flexibility it is possible to implement different replication techniques such as: passive (cold and hot) and semi-active. The infrastructure implementation and the tests performed show the viability of this solution.

Keywords: Web Services, Fault Tolerance, FT-CORBA.

Capítulo 1: Introdução

1.1. Motivação

Com a popularização da Internet surgiram várias tecnologias para o desenvolvimento de aplicações que oferecem serviços dinâmicos e interativos, dando origem aos *e-services*, tais como: comércio eletrônico (*e-commerce*), governo eletrônico (*e-gov*), computação em grade, biblioteca digital, dentre outros. Entretanto, estas tecnologias possuem ambientes operacionais específicos tornando difícil a integração entre aplicações. Para facilitar esta integração, com objetivo de definir padrões abertos, grupos de empresas especializadas se uniram em consórcios e definiram o SOAP (*Simple Object Access Protocol*) [SOAP, 2003], o WSDL (*Web Services Description Language*) [WSDL, 2001] e o UDDI (*Universal Description, Discovery and Integration*) [UDDI, 2002]. Esse conjunto de protocolos e padrões, juntamente com outros relacionados que estão sendo definidos por esses consórcios e pela academia, caracterizam um novo paradigma no desenvolvimento de aplicativos: os serviços web (ou *web services*).

A palavra-chave dos serviços web é interoperabilidade. Componentes de software podem ser acessados através de protocolos consolidados e amplamente utilizados como o HTTP (*Hypertext Transfer Protocol*) e o XML (*Extensible Markup Language*)[XML, 2000]. A sua principal vantagem está em permitir a integração de componentes já desenvolvidos, esta flexibilidade permite explorar as melhores características de cada tecnologia envolvida no processo de desenvolvimento de uma aplicação.

Por possuir uma arquitetura aberta os serviços Web têm se mostrado uma excelente opção para o desenvolvimento de sistemas distribuídos permitindo o desenvolvimento de soluções adaptadas a heterogeneidade e complexidade destes ambientes.

Contudo, para que todo o potencial dos serviços Web possa ser explorado faz-se necessário a definição de um modelo de desenvolvimento que atenda os requisitos de confiabilidade e alta disponibilidade. Este modelo deve ser flexível o suficiente para que todas as características dos serviços Web sejam mantidas. Confiabilidade e disponibilidade são atributos da área de pesquisa de tolerância à faltas. Esta área tem como objetivo primário propor soluções que permitam aplicações terem seus serviços executados mesmo na presença de falhas. Ainda são poucos os trabalhos que endereçam os requisitos de tolerância a faltas para os serviços web.

O principal problema encontrado para a proposição de um modelo tolerante a faltas nesta área se concentra no fato dos servidores web serem *stateless,* ou seja, eles não mantêm uma conexão ativa durante todas as requisições do cliente. Por este motivo aplicações críticas construídas sobre os protocolos Internet utilizam técnicas simplificadas que detectam a falta e direcionam futuras requisições para servidores redundantes. Estes métodos não são capazes de tolerar a falta durante o processamento de uma requisição.

Tanto o modelo de serviços Web quanto as especificações CORBA foram idealizados para o desenvolvimento de componentes de sistemas distribuídos, entretanto estas tecnologias não devem ser vistas como mutuamente excludentes mas complementares. A OMG em novembro de 2003 definiu uma especificação para a definição de objetos CORBA como serviços Web [OMG WSDL, 2003]. Porém, esta especificação não define como devem ser implementados os *gateways* que permitem a exposição de um objeto CORBA como um serviço Web.

1.2. Proposta

Os trabalhos relacionados a tolerância a faltas em serviços web utilizam técnica de replicação passiva e implementam mecanismos que detectam a falha no serviço e iniciam processos de recuperação do estado da aplicação em um servidor redundante.

A proposta deste trabalho é definir e implementar uma infra-estrutura tolerante a faltas para sistemas orientados a serviços. O foco principal deste trabalho será a utilização da técnica de replicação ativa, entretanto a infra-estrutura proposta deve ser flexível o suficiente para suportar diferentes técnicas de replicação, tais como passiva e semi-ativa.

A principal justificativa para utilização da técnica de replicação ativa é a possibilidade de implementação de um modelo que tolere as seguintes classes de faltas: parada, omissão e valor. Este modelo deve tolerar faltas que ocorrem durante o processamento das requisições e deve garantir o determinismo entre as réplicas, ou seja, assegurar que as réplicas livres de falhas recebam e executem as requisições na mesma ordem relativa.

Através deste modelo deve ser possível a replicação de um serviço em um conjunto de servidores dispersos geograficamente aumentando a disponibilidade e a confiabilidade do serviço. Entretanto, o usuário do serviço web deve ter a percepção do serviço como um componente único, quando na realidade este serviço é composto por um conjunto de componentes replicados. Este modelo deve ser capaz de interagir com diferentes tipos de serviços: *stateless, stateful,* síncronos e assíncronos.

Este trabalho tem ainda como objetivo explorar a integração entre as tecnologias CORBA e serviços web através da definição e implementação de componentes que permitam a exposição de objetos CORBA como serviços Web. Estes objetos são

implementados sob as especificações que tratam os aspectos relacionados a tolerância a faltas na arquitetura CORBA (FT-CORBA).

1.3. Organização

Este trabalho é composto por 8 capítulos que seguem a seguinte abordagem:

Capítulo 2: Tolerância a Faltas
É abordado neste capítulo os conceitos fundamentais relacionados a tolerância a faltas e as diferentes técnicas aplicadas em sistemas distribuídos.

Capítulo 3: CORBA
Este capítulo envolve as características da arquitetura CORBA e as especificações relacionadas a implementação de segurança e tolerância a faltas em sistemas distribuídos.

Capítulo 4: Serviços Web
Descreve o modelo conceitual dos serviços Web, a arquitetura e as tecnologias utilizadas no seu desenvolvimento.

Capítulo 5: Serviços Web Tolerantes a Faltas
Apresenta uma análise dos trabalhos encontrados na literatura que propõem modelos tolerantes a faltas para arquiteturas orientadas a serviços.

Capítulo 6: *FTWeb* – Uma infra-estrutura para tolerância a faltas em Serviços Web.
Este capítulo apresenta a arquitetura do *FTWeb* e as funcionalidades providas pelos componentes que formam esta arquitetura.

Capítulo 7: Implementação e Avaliação dos Resultados
No decorrer deste capítulo são apresentados os detalhes de implementação e os resultados obtidos com a avaliação de desempenho da infra-estrutura *FTWEB*.

Capítulo 8: Conclusão
Conclui a dissertação, apontando os benefícios alcançados e os trabalhos futuros relacionados ao projeto.

Capítulo 2: Tolerância a Faltas

2.1. Introdução

Analisando a evolução dos computadores é possível perceber o quanto os componentes de hardware cresceram em confiabilidade. O desenvolvimento da indústria de hardware tem proporcionado um crescimento significativo na confiabilidade dos equipamentos a partir de uma evolução tecnológica crescente e acelerada vista nos poucos anos da história da informática que compreendem os primeiros computadores a válvula até os modernos *notebooks*.

Entretanto, o software e os procedimentos de projeto estão tornando-se cada vez mais complexos e por isso, mais suscetíveis a falhas. Apenas a confiabilidade nos componentes de hardware não garante a qualidade e segurança desejada aos sistemas de computação. Para desenvolver um sistema com os atributos de tolerância a faltas desejados, um conjunto de métodos e técnicas deve ser empregado.

Essas técnicas garantem funcionamento correto do sistema mesmo na ocorrência de falhas e são em sua maioria baseadas em redundância, exigindo componentes adicionais ou algoritmos especiais. Este capítulo tem por objetivo apresentar conceitos e técnicas de tolerância a faltas.

2.2. Conceitos de tolerância a faltas

Com a crescente demanda de processos automatizados e informatizados a confiabilidade e disponibilidade dos sistemas se tornam requisitos indispensáveis. Em um ambiente distribuído suportado por infra-estrutura de rede de computadores, supõe-se que o sistema computacional opere apropriadamente, sem interrupção no seu serviço e sem perda de dados ou mensagens.

Confiabilidade e a disponibilidade são atributos da área de pesquisa de tolerância a faltas. Está área tem como objetivo primário propor soluções que permitam que aplicações terem seus serviços executados mesmo na presença de falhas.

2.2.1 Falta, erro ou falha

As faltas aparecem geralmente classificadas na literatura como físicas e humanas. As faltas físicas compreendem a qualquer componente (módulo, objeto ou processo) de

hardware ou software que opere diferentemente do especificado. As faltas humanas compreendem a faltas de projeto, de interação e de implementação, que podem ser acidentais ou intencionais.

Uma **falha** (*failure*) é definida como um desvio da especificação, um defeito do sistema e está relacionada ao universo do usuário, ele é quem percebe que o sistema não está funcionando conforme o especificado. O **erro** está relacionado ao universo da informação, ou seja, um sistema está em estado errôneo se o processamento após este estado possa levar a uma falha. A **falta** (*fault*) pode ser definida como a causa física ou algorítmica do erro (Figura 2.1) [Pradhan, 1996], [Anderson, Lee 1981], [Jalote 1994].

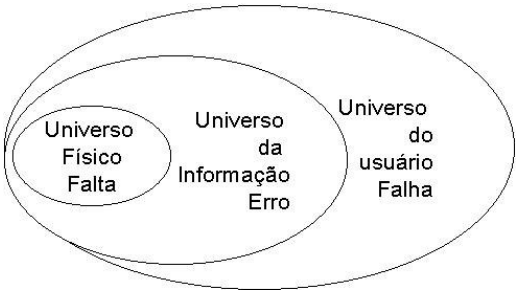

Figura 2.1 - Modelo de três universos

Considerando um servidor que hospeda um sistema de controle de estoque e que possui uma memória com defeito em alguns de seus bits. Esta **falta** pode provocar **erro** na interpretação de alguma informação armazenada. E como resultado o sistema pode negar a um usuário autorizado o acesso a alguma funcionalidade (**falha**).

É importante ressaltar que uma falta não necessariamente leva a um erro (aquela porção da memória pode nunca ser usada) e um erro não necessariamente conduz a uma falha (no exemplo, o usuário poderia obter acesso às informações através de outros dados redundantes).

As faltas são consideradas inevitáveis. Os componentes físicos sofrem interferências internas e externas, sejam ambientais ou humanas. Os projetos de software e hardware são complexos devido ao grande volume de detalhes. As falhas podem ser evitadas através de técnicas de tolerância à faltas. Além da causa para definir uma falha, considera-se ainda:

- Natureza: falha de hardware, falha de software, de projeto, de operação, etc
- Duração ou persistência: permanente ou temporária (intermitente ou transitória)
- Extensão: local a um módulo, global
- Valor: determinado ou indeterminado no tempo

2.2.2 Latência

Latência de falta é o período de tempo desde a ocorrência da falta até a manifestação do erro devido àquela falta. Define-se latência de erro como o período de tempo desde a ocorrência do erro até a manifestação da falha devido aquele erro.

Baseando-se no modelo de 3 universos apresentado na seção 2.2.1, o tempo total desde a ocorrência da falha até o aparecimento do defeito é a soma da latência de falhas e da latência de erro. [Weber 1999].

2.3. Tolerância à faltas

Tolerância a faltas é a qualidade de um determinado serviço fornecido por um dado sistema. É a propriedade que garante que o serviço continuará disponível mesmo na presença de falhas em algum de seus componentes. Os principais atributos de tolerância a faltas são [Pradhan 1996]:

- **Confiabilidade**: capacidade de atender a especificação, dentro de condições definidas, durante certo período de funcionamento e condicionado a estar operacional no início do período.
- **Disponibilidade:** probabilidade do sistema estar operacional num instante de tempo determinado; alternância de períodos de funcionamento e reparo.
- **Segurança:** probabilidade do sistema estar operacional e executar sua função corretamente ou descontinuar suas funções de forma a não provocar dano a outros sistemas ou pessoas que dependam dele.

Define-se também como **segurança** o atributo que provê a proteção contra falhas maliciosas, visando a privacidade e autenticidade, integridade e irrepudiabilidade dos dados. Porém esta definição esta relacionada a área de segurança computacional e esta fora do escopo deste trabalho.

2.3.1 Técnicas de tolerância a faltas

O desenvolvimento de sistemas tolerantes a faltas prevê a implementação de um conjunto de métodos e técnicas. Geralmente são utilizadas as técnicas de prevenção, tolerância, validação e previsão de falhas.

- **Prevenção de falhas:** envolve a seleção de métodos para a especificação de projetos e de tecnologias adequadas para a implementação de seus componentes. Impede a ocorrência ou introdução de falhas
- **Tolerância a falhas:** permite que um determinado serviço continue operante mesmo na presença de falhas. As técnicas empregadas são detecção de falhas, localização, reconfiguração e tratamento.
- **Validação:** remoção da falhas e verificação da presença de falhas.
- **Previsão de falhas:** consiste em estimar a presença e a conseqüência de falhas.

Em sistemas que exigem alta confiabilidade e disponibilidade apenas técnicas de prevenção e remoção de faltas não são suficientes. Para sistemas que possuem estes requisitos o ideal é a utilização de técnicas de tolerância a faltas.

2.3.2 Fases de aplicação das técnicas de tolerância a faltas

A implementação de técnicas de tolerância a faltas identifica 4 fases [Anderson, Lee, 1981] :

Detecção de erro: esta é a primeira fase e é responsável por detectar erros no sistema. Antes da manifestação da falta como erro, se a falta esta latente não pode ser detectada. Esta fase é considerada a mais importante porque somente após a manifestação da falta como erro é que os mecanismos de tolerância a faltas podem ser acionados. Um exemplo de mecanismo de detecção é o esquema de duplicação e comparação (Figura 2.2). Componentes idênticos (hardware ou software) realizam o mesmo processo sobre o mesmo conjunto de dados. Após o processamento os resultados fornecidos pelos componentes são comparados havendo diferença o erro é detectado. É importante ressaltar que o elemento votador que realiza a análise do resultado também deve ser tolerante a faltas, a fim de evitar um ponto crítico de falha.

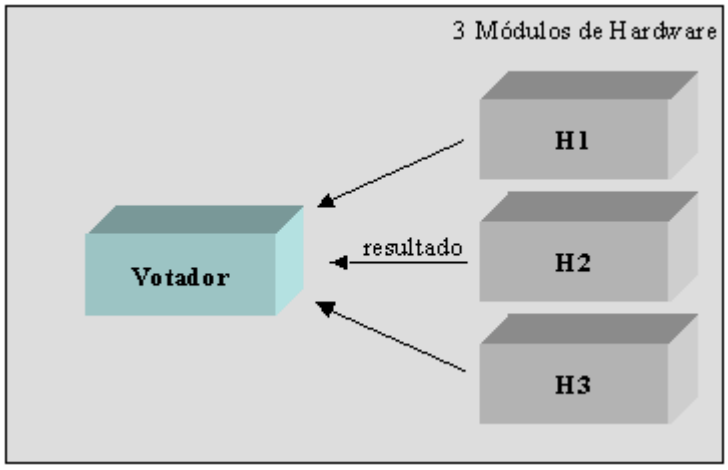

Figura 2.2 – Mecanismo de detecção através de comparação

Confinamento: devido a latência, ou seja, o período entre a manifestação do erro e a sua detecção, outros componentes do sistema podem ser impactados. A fase de confinamento tem por objetivo estabelecer limites para a propagação do erro impedindo assim que outros componentes sejam influenciados. A fase de confinamento deve ser prevista na especificação do projeto definindo interfaces que restrinjam o fluxo de informações na presença de falhas.

Processamento de erros: após um erro ter sido detectado é necessário removê-lo e retornar o sistema a um estado sem erros. Duas técnicas são utilizadas na fase de processamento de erros:

- **Compensação de erros**: consiste em mascarar os erros a fim de garantir a resposta correta mesmo na presença de falhas. Os mecanismos usuais para implementação de compensação de erros são replicação de componentes e código de correção de erros.
- **Recuperação de erros:** envolve a substituição do estado errôneo por um estado sem erros. Os mecanismos para a implementação de recuperação de erros são recuperação por retorno e recuperação por avanço. Os dois tipos de recuperação podem ser visualizados na Figura 2.3

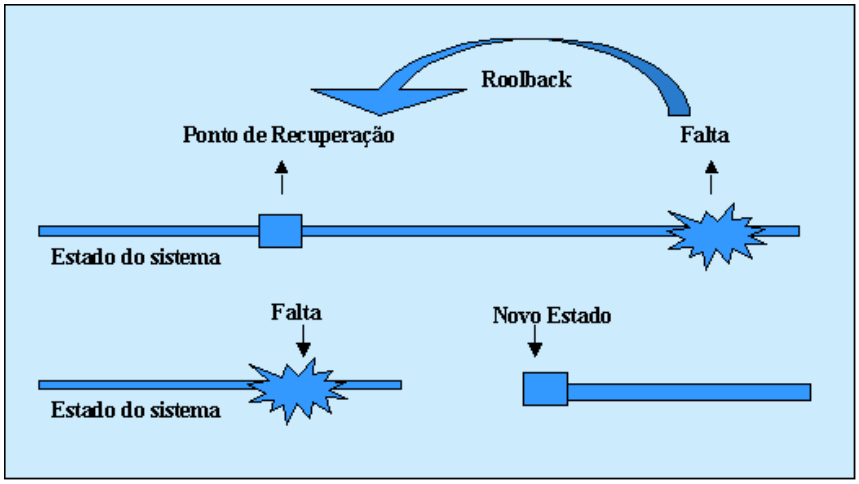

Figura 2.3 - Recuperação por retorno e por avanço

A técnica de recuperação por retorno exige que informações sobre o estado do sistema sejam armazenadas regularmente em um repositório de memória estável. Permitindo que, após a detecção de uma falha, o sistema seja retornado ao último estado armazenado. A recuperação por retorno pode ser simples para um sistema com um único processo, mas em sistemas com processamento distribuído torna-se complexa devido a dependência de informações entre os processos. Técnicas de recuperação por retorno não são adequadas a sistemas de tempo real. Nesses sistemas deve ser usada recuperação por avanço [Jalote 1994].

A recuperação por avanço consiste em colocar o sistema em sua operação normal a partir do estado atual sem a recuperação do estado anterior.

Tratamento: A fase de tratamento impede que faltas detectadas e recuperadas sejam reativadas colocando o sistema em estado errôneo novamente. Esta fase consiste em:

- localizar o erro,
- localizar a falha,
- reparar a falha,
- recuperar o restante do sistema.

Para os dois tipos de localização é usado diagnóstico. O diagnóstico é um teste com comparação dos resultados gerados com os resultados previstos. Após a localização, a falha é reparada através da remoção do componente danificado.

As fases envolvem o conceito de uma seqüência complementar de atividades, que devem ser executadas após a ocorrência de uma ou mais faltas. O diagnóstico, pode ser usado tanto como um mecanismo nas fases de detecção de erros e de localização de falhas e como uma técnica isolada conduzida periodicamente para diminuir a latência.

2.4. Tolerância a Faltas em Sistemas Distribuídos

Pode-se definir Sistemas Distribuídos como sendo "uma coleção de computadores autônomos ligados através de uma rede contendo softwares projetados para produzir uma facilidade computacional integrada" – [Colouris 1994]. Analisando as características de um sistema distribuído pode-se considerar a existência de um modelo lógico e um modelo físico. O modelo físico consiste em muitos computadores (frequentemente chamados de nodos), a rede de comunicação e os elementos que compõe os nodos: processador, relógio local, memória local volátil, armazenamento não volátil, interface de rede e software. Resumindo pode-se considerar que o modelo físico é composto por um conjunto de computadores autônomos interconectados por uma rede de comunicação [Jalote 1994].

O modelo lógico consiste na aplicação distribuída. Um conjunto de processos concorrentes interage para a execução de determinada tarefa. Neste ponto a rede é considerada completamente conectada e os canais entregam mensagens na ordem que foram enviadas, mas não existe ordenação total de mensagens, apenas ordenação parcial.

Devido à ausência de um relógio global, a ordenação de eventos não ocorre em uma base de tempo comum, a ordenação ocorre em nodos diferentes medidos por relógios independentes. Relógios lógicos [Lamport 1978] são um meio de assinalar um número a um evento. Não possuem nenhuma relação com o tempo físico. Podem ser implementados através de *timestamps*. O relógio lógico pode ser usado para ordenação total de eventos e é suficiente para a maior parte das aplicações que não exigem respostas críticas de tempo.

Os sistemas distribuídos podem ser síncronos ou assíncronos, dependendo da existência de um tempo definido e conhecido para troca de mensagens. É usual também o conceito de tempo limite (*timeout)* associado aos sistemas síncronos.

2.4.1 Faltas em sistemas distribuídos

A implementação de mecanismos tolerantes a faltas esta relacionado a classificação das faltas. Após a correta identificação das faltas em um sistema é possível dimensionar os mecanismos necessários para tolerar e as tratar as suas faltas. Uma das possíveis formas de classificar as faltas em sistemas distribuídos é baseada no comportamento do componente quando este sofre uma falta.[Coulouris et. al. 2001]:

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

- Faltas por omissão: fazem com que um serviço não responda a algumas requisições de serviço. Comportamento intermitente ou transitório.

- Faltas por temporização: são aquelas que fazem com que um componente responda a uma requisição de serviço fora do intervalo

de tempo especificado. Também conhecidas como faltas de desempenho.

- Faltas por valor: são aquelas que ocorrem quando uma resposta é devolvida com o valor fora do especificado, porém dentro do intervalo de tempo especificado.

- Faltas bizantinas: esta categoria de faltas engloba todas as classes de faltas citadas acima. Uma sub-categoria interessante da falta bizantina é a falta de processamento incorreto, onde o componente não apresenta falta porém produz respostas incorretas para um conjunto de entradas.

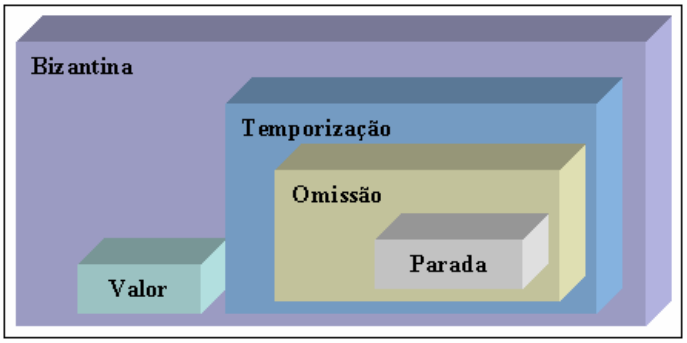

Figura 2.4 - Faltas em sistemas distribuídos

2.4.2 Técnicas de replicação

A tolerância a faltas em sistemas distribuídos é alcançada através da replicação de processos em nodos distintos. A utilização de técnicas de replicação aumenta a confiabilidade e disponibilidade permitindo que o sistema continue operante mesmo na presença de falhas. Para o usuário que possui a percepção do serviço, o componente continua sendo único, na realidade o serviço é formado por um conjunto de componentes replicados, que permitem a execução continuada mesmo na presença de falhas em um componente.

A administração das réplicas prevê a recuperação em situação de falha parcial e total, a consistência de estado entre as réplicas e a transparência do conjunto. Estes requisitos são assegurados através da implementação de protocolos de coordenação.

As técnicas de replicação podem ser implementadas através das abordagens Réplicas Passivas, Réplicas Ativas e Réplicas Semi-ativas. Estas abordagens são capazes de mascarar falhas individuais de réplicas membros do conjunto. Entretanto na escolha de uma dessas abordagens devem ser considerados todas as características da aplicação e os seus requisitos de tolerância a faltas [Budhiraja et. al., 1993] [Fraga et. al. 2001] [Favarim 2003].

2.4.2.1 Replicação passiva

Esta abordagem prevê a existência de um membro principal ou primário que recebe e executa as requisições do cliente (Figura 2.5). As réplicas que compõem o grupo têm a função de *backup*, ou seja, elas substituem o membro principal em caso de falhas.

O membro primário envia periodicamente mensagens de *checkpoint*, estas mensagens têm a finalidade de assegurar que os estados dos membros se mantenham mutuamente consistentes. Esse mecanismo de *checkpoint* faz com que esta abordagem apresente a vantagem de não precisar de determinismo[1] entre as réplicas.

Se freqüência no envio de mensagens de *checkpoint* for elevada, o sistema pode ter o seu desempenho impactado. O número de mensagens de *checkpoint* pode ser limitado a um determinado número de requisições. Esta estratégia é ideal para sistemas onde falhas no membro primário não ocorram com freqüência. Entretanto se após uma requisição o primário falhar sem realizar o *checkpoint*, o cliente terá que reenviar a sua requisição.

Mecanismos de log são utilizados para evitar que o cliente necessite reenviar requisições após a falha do primário, mantendo registradas em discos todas as requisições durante os intervalos de *checkpoint*.

Quando há falha no membro primário, um protocolo de eleição, seleciona entre as réplicas, qual irá substituir o membro primário. A réplica selecionada assume a execução das requisições dos clientes a partir do último *checkpoint*. Nenhuma informação pode ser processada até que o novo primário seja eleito.

Uma réplica só poderá participar da eleição, se ela recebeu a última mensagem de *checkpoint*, um protocolo deve reconhecer um backup desatualizado e somente após a sua atualização permitir que a réplica seja elegível a membro primário. A interação entre cliente e servidor é facilitada uma vez que o cliente envia as requisições somente ao membro primário.

Nesta abordagem somente é possível detectar faltas de parada e omissão. Os mecanismos utilizados para detectar estas faltas são *timeout* e *keepalive.* O mecanismo *keepalive* é responsável pela rapidez na recuperação do erro entretanto se este mecanismo apresentar uma freqüência elevada pode influenciar no desempenho do sistema.

Mecanismos de retenção de resultado devem ser implementados para evitar situações em que o membro primário falhe logo após a execução de uma requisição, neste caso como a mensagem não foi enviada ao cliente a réplica substituta poderá

[1] O determinismo de réplicas introduzido por [Schneider 90] implica que réplicas corretas, partindo do mesmo estado inicial e processando o mesmo conjunto de entradas, na mesma ordem relativa, devem produzir as mesmas saídas. O determinismo de réplicas é uma condição para a consistência de estados entre réplicas ativas.

repetir o processamento da requisição. Com a implementação de mecanismo de retenção de resultado o novo membro primário apenas retransmitirá o resultado processado anteriormente.

Através desta abordagem é possível mascarar falhas, uma vez que o serviço ficaria indisponível apenas se todas as réplicas apresentassem falhas. As características da aplicação e do sistema distribuído determinam qual será o número de *backups* que recebem as mensagens de *checkpoint*.

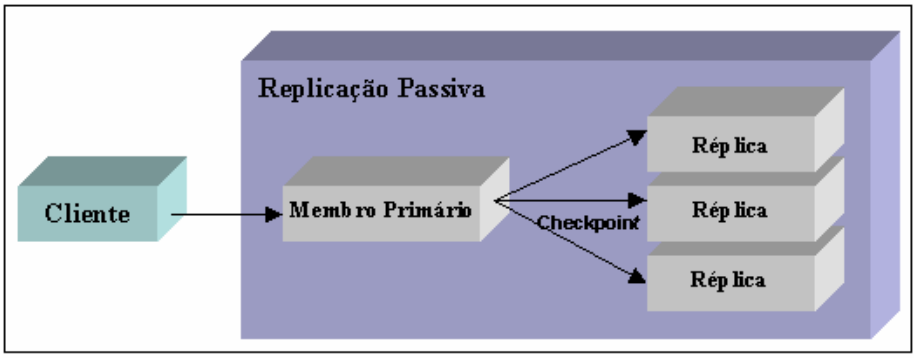

Figura 2.5 - Replicação passiva

A replicação passiva pode seguir duas diferentes abordagens:

- **Passiva Fria**: Apenas o membro primário executa as requisições do cliente. As demais réplicas (secundárias) ficam em modo standby (reserva). O primário grava periodicamente informações de estado (*checkpoint*). Em caso de falha do objeto primário uma das réplicas secundárias assume o papel de novo primário, atualizando seu estado a partir do último *checkpoint* gravado.

- **Passiva Quente**: Este abordagem funciona de maneira semelhante a anterior, entretanto os *checkpoints* realizados pelo membro primário são difundidos para as réplicas periodicamente, otimizando o desempenho da atualização de uma réplica em caso de falha do membro primário;

2.4.2.2 Replicação ativa

Na replicação ativa todas as réplicas livres de falhas recebem, executam e respondem todas as requisições enviadas pelo cliente (Figura 2.6). Este modelo deve prover determinismo de réplicas para que não haja inconsistência de estados entre as réplicas. Para garantir o determinismo a replicação ativa necessita de difusão atômica, este tema é abordado na seção 2.4.3 onde são discutidas as primitivas do suporte de comunicação de grupo.

Mecanismos de entrega de repostas ao cliente devem ser implementados, uma vez que todas as réplicas executam a mesma requisição. As possíveis formas de implementação destes mecanismos são:

- A primeira resposta processada é enviada ao cliente
- As respostas são concatenadas na seqüência em que são processadas e enviadas ao cliente
- As respostas são enviadas a um elemento votador que analisa as respostas e entrega ao cliente a resposta mais freqüente.

A replicação ativa é ideal para aplicações de tempo real, que possuam requisitos de disponibilidade altos com sobrecarga mínima em situações de falha. Esta abordagem cobre todas as classes de falhas (faltas por parada, omissão, valor e bizantina)

O custo desta abordagem é maior quando comparada com a técnica de replicação passiva, devido a maior necessidade de recursos como memória e processador. Como o fluxo de mensagens entre os componentes tende a ser maior o canal de comunicação é mais utilizado.

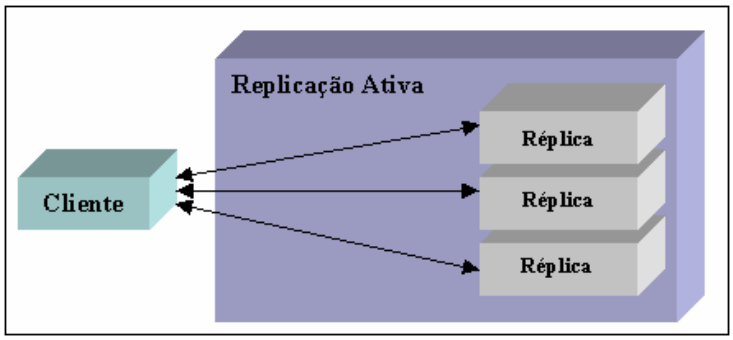

Figura 2.6 - Replicação ativa

2.4.2.3 Replicação semi-ativa

Esta abordagem de replicação é uma combinação das características replicação ativa e passiva. Assim como na replicação ativa, nesta abordagem todos os componentes recebem e executam as requisições dos clientes, contudo existe a figura da réplica líder que é responsável pela definição da ordem de execução das requisições e também responsável pela entrega da resposta ao cliente (Figura 2.7).

Assim como na abordagem da replicação passiva na falha da réplica líder é executado uma votação para substituir o líder por outra réplica livre de falha. Esta abordagem não apresenta o requisito de recuperação de estado em retorno devido a todas as réplicas estarem ativas e, portanto, evoluindo no mesmo estado.

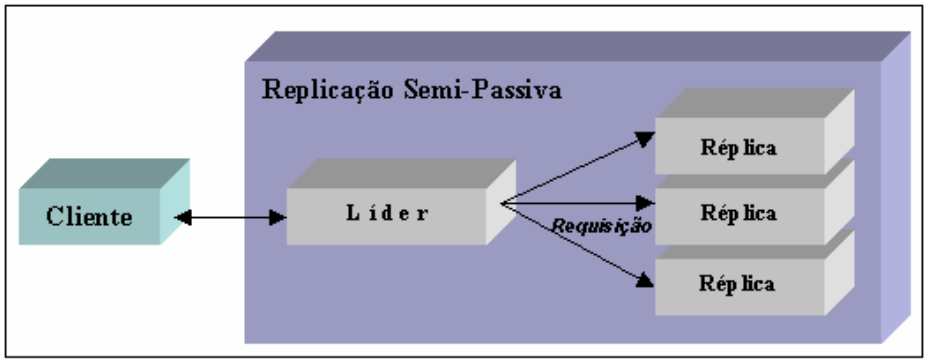

Figura 2.7 - Replicação semi-ativa

2.4.3 Suporte de Comunicação de Grupo

Para tratar a comunicação entre componentes de sistemas distribuídos são utilizadas soluções de comunicação em grupo. A comunicação em grupo é uma das abstrações mais relevantes na área de tolerância a faltas, pois permite a concretização das técnicas de replicação. Define-se grupo como uma coleção de componentes (membros) que interagem e que compartilham do mesmo objetivo. Um problema fundamental relacionado a sistemas distribuídos, em especial em suportes de comunicação de grupo, diz respeito à determinação, em tempo de execução, de quais processos são membros de um grupo. Esta lista de membros pertencentes ao grupo é chamada comumente de *membership* (pertinência) ou visão.

O *membership* de um grupo pode ser estático ou dinâmico. Em um *membership* estático os membros do grupo são conhecidos previamente, em tempo de compilação. Em *memberships* dinâmicos, processos entram e saem de grupos, possuindo um número de membros variável em função do tempo. O *membership* normalmente é dinâmico por questões de flexibilidade. O conjunto de membros de um grupo pode mudar por pelo menos três razões:

- **Requisitos da Aplicação:** Algum requisito funcional da aplicação deve implicar nas alterações de *membership*. Por exemplo, em uma vídeo-conferência, algumas pessoas podem entrar na sessão (grupo) após o início desta, e ainda algumas podem deixá-la antes de seu fim;
- **Falhas de Processos:** Processo falhos devem ser removidos de grupos;
- **Redimensionamento do Sistema:** Grupos podem crescer ou diminuir segundo a demanda.

Quando o suporte de comunicação de grupo fornece o *membership*, a lista de membros é disponibilizada através do serviço de *membership* do grupo. Este serviço controla as alterações no grupo através de operações específicas oferecidas aos processos (exemplo: *join* e *leave*).

O serviço de *membership* também é responsável em disponibilizar um mecanismo de detecção de falhas a fim de descobrir processos faltosos e excluí-los dos grupos. Desta forma o serviço de *membership* assegura a consistência e o gerenciamento eficiente do grupo.

2.4.3.1 Primitivas de Comunicação de Grupo

O paradigma da comunicação em grupo provê a comunicação confiável entre os membros. Sua principal propriedade é a troca de mensagens através de difusão. Suportes de comunicação de grupo disponibilizam várias primitivas de comunicação com diferentes garantias de confiabilidade e ordenação, atendendo diferentes requisitos de aplicações. Estas garantias são definidas através das propriedades implementadas pelos serviços que oferecem estas primitivas no suporte.

Em um sistema distribuído onde os processos se comunicam por difusão, a presença de faltas pode ocasionar perdas de mensagens, que levam a inconsistência nos estados dos membros dos grupos. Assim sendo, as primitivas utilizadas para comunicação devem oferecer pelo menos confiabilidade e um nível de garantia em relação a ordenação das mensagens. Os quatro tipos básicos de primitivas de difusão são detalhados a seguir:

Difusão Confiável

Um serviço de difusão confiável garante que todas as mensagens enviadas a um grupo de processos serão recebidas por todos os membros não faltosos do grupo. A difusão confiável é implementada através de duas primitivas:

- *R*-multicast (G;m): A mensagem *m* é difundida para todos os processos pertencentes ao grupo *G*.
- *R*-deliver (*m*): A mensagem *m* é liberada para a aplicação.

Estas primitivas devem satisfazer as seguintes propriedades:

1. **Validade:** Se um processo correto difundir um *m* em G, então algum processo correto pertencente a *G* entregará *m* ou nenhum processo do grupo está correto;

2. **Acordo:** Se um processo correto pertencente a *G* entrega a mensagem *m*, então todos os processo corretos pertencentes a *G* entregarão *m*;

3. **Integridade:** Para qualquer mensagem *m*, cada processo correto pertencente a *G* entrega *m* no máximo uma vez e apenas se *m* foi previamente difundida em *G*.

Qualquer protocolo de difusão confiável deve prover estas duas primitivas e satisfazer essas três propriedades.

Difusão FIFO

A difusão seletiva FIFO (*First-In-First-Out*) é uma difusão confiável com a propriedade de ordenação FIFO:

- **Ordenação FIFO Local:** Se um processo difunde uma mensagem *m* em *G* antes de difundir *m0* em *G*, então todos os processos corretos em *G* não entregam *m0* antes de entregar *m*.

A ordenação FIFO garante que as mensagens difundidas por um processo serão entregues pelos receptores na mesma ordem em que foram realizadas. A implementação dessa primitiva é feita através de um número de sequência, inserida no cabeçalho da mensagem, indicando a ordem de entrega de cada mensagem.

Difusão Causal

A difusão causal é necessária quando existe inter-dependência entre as mensagens. Nestes casos, faz-se necessário um tipo de ordenação que leve em consideração a precedência causal de eventos [Lamport, 1978]. O evento *e* precede causalmente o evento *f* (denotado $e \Rightarrow f$) se e somente se:

1. o mesmo processo executa e e depois executa f, ou;
2. *e* é a difusão de uma mensagem e *f* é a entrega desta mensagem, ou;
3. existe um evento *h*, tal que $e \Rightarrow h$ e $h \Rightarrow f$.

A ordenação causal generaliza a noção de dependência entre mensagens e garante que uma mensagem só será entregue à aplicação se a mensagem que a causou tiver sido entregue antes. Formalmente, uma difusão causal é caracterizada por uma difusão confiável que satisfaz a propriedade de ordenação causal, de acordo com as seguintes propriedades de ordenação:

- Ordenação Causal Local: Se a difusão de uma mensagem *m* em *G* precede causalmente a difusão de uma mensagem *m0* em *G*, então nenhum processo correto em *G* entrega *m0* antes de entregar *m*.

A implementação desta primitiva pode ser realizada através da utilização de um vetor histórico, inserido no cabeçalho da mensagem, indicando as mensagens que a precedem. O receptor ao receber essa mensagem apenas a entregará para a aplicação se já tiver recebido e entregue todas as mensagens precedentes indicadas no vetor.

Difusão Atômica

A difusão atômica garante que todos os processos corretos entregarão todas as mensagens e na mesma ordem. Desta forma, todos os processos tem a mesma visão [Birman, 1996] do sistema e podem agir de maneira consistente sem comunicações adicionais. Formalmente, uma difusão atômica é uma difusão confiável que satisfaz a seguinte propriedade:

- **Ordenação Total Local:** Se dois processos corretos *p* e *q* entregam as mensagens *m* e *m0* endereçadas ao grupo *G*, então *p* entrega *m* antes de *m0* se e somente se *q* entregar *m* antes de *m0*. A difusão atômica é considerada um mecanismo fundamental para a implementação de replicação ativa (ou por máquina de estados) para tolerância a faltas [Schneider, 1990].

Existem diversas formas de implementar a difusão atômica [Défago et al., 2002]:

- **Hitórico de Comunicação:** Essa abordagem é baseada no algoritmo de ordenação de eventos de Lamport [Lamport, 1978]. O algoritmos baseados em histórico de comunicação entregam as mensagens em uma ordem total que é compatível com a relação de precedência das mensagens;

- **Baseada em privilégio:** Nesta abordagem se assume a existência de um grupo de emissores formando um anel vitual em que um bastão (*token*) circula dentro do grupo. Apenas o processo emissor de posse do bastão tem o privilégio de enviar mensagens ao grupo. Uma vez que cada processo receptor conhece a sequência de emissores no anel lógico, os receptores do grupo podem, desta forma, estabelecer uma ordem total determinística para as mensagens enviadas ao grupo (Figura 2.8).

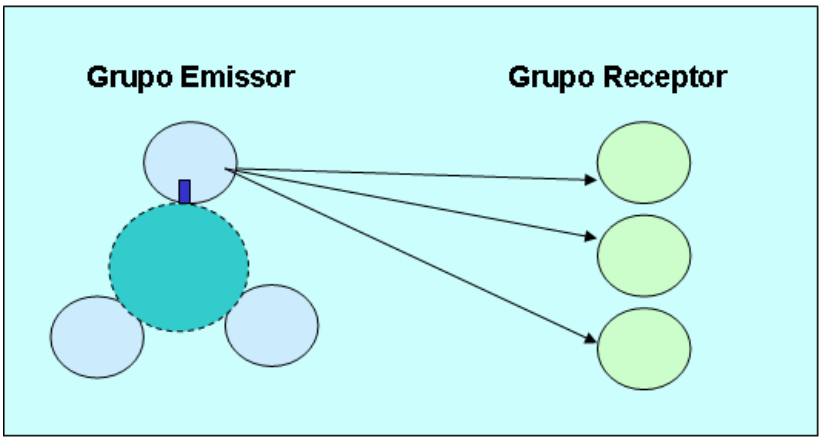

Figura 2.8 – Difusão Atômica Baseada em privilégio

- **Sequenciador fixo:** Nesta abordagem se assume a existência de um processo sequenciador que recebe todas as mensagens do grupo emissor e as redireciona ao grupo receptor em uma ordem especifica. (Figura 2.9)

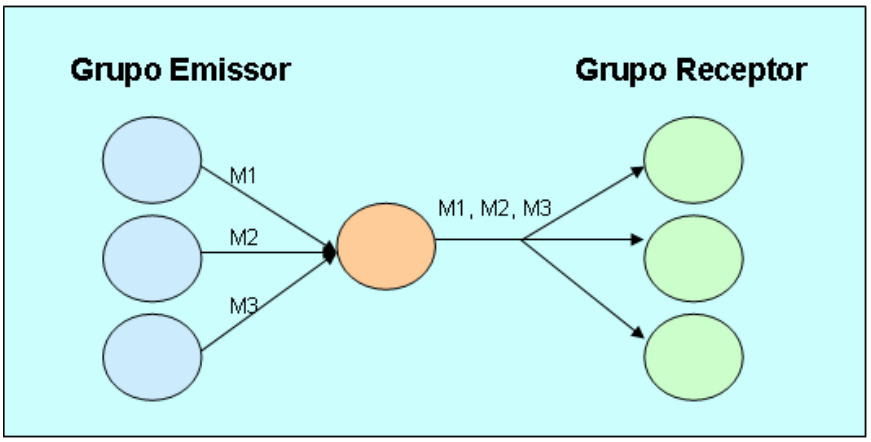

Figura 2.9 – Difusão Atômica Seqüenciador Fixo

- **Sequenciador móvel:** Para tolerar a falta do seqüenciador fixo é proposta então, a abordagem do sequenciador móvel. Nesta abordagem é definido um grupo de processos sequenciadores, formando um anel lógico, que recebe todas as mensagens do grupo emissor. Mas, apenas um processo sequenciador, possuidor do bastão, tem o privilégio de repassar as mensagens ao grupo receptor, em uma ordem especifica. Como cada processo receptor conhece a ordem de sequenciadores no anel lógico, os receptores do grupo conseguem estabelecer uma ordem total. (Figura 2.10)

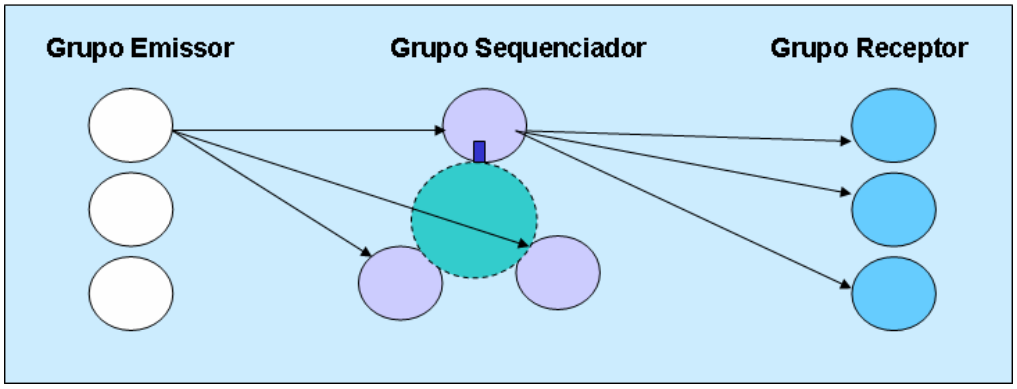

Figura 2.10 – Difusão Atômica Seqüenciador Móvel

- **Acordo no destino:** Neste caso, a mensagem de um emissor é enviada para todos os membros do grupo receptor. No entanto, para definir a ordem de entrega dessa mensagem, cada receptor participa de um protocolo de acordo (ou consenso distribuído) no sentido de definir a ordem total que essa mensagem deve ser entregue. (Figura 2.11)

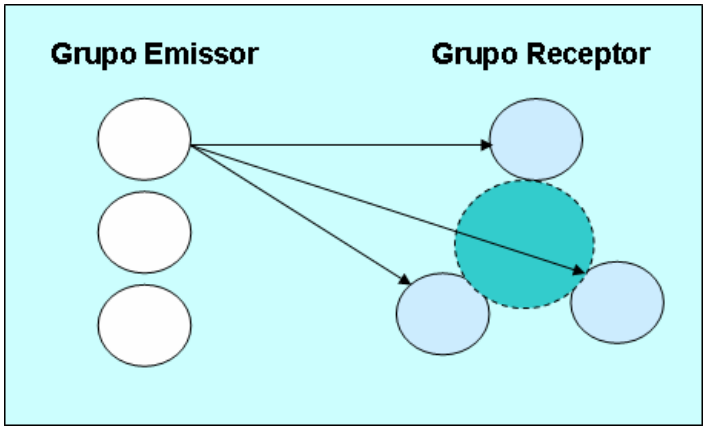

Figura 2.11 – Difusão Atômica Acordo no Destino

2.5. Conclusões do capítulo

Para desenvolver um sistema com os atributos de tolerância a faltas desejados, um conjunto de métodos e técnicas deve ser empregado. As técnicas apresentadas neste capítulo aumentam a confiabilidade e a disponibilidade do sistema.

Entretanto é importante ressaltar que o desenvolvedor deve reconhecer as exigências da aplicação quanto a confiabilidade e a disponibilidade, para escolher as técnicas de tolerância a faltas que supra os requisitos da aplicação e também ter condições de desenvolver os mecanismos complementares para alcançar a confiabilidade desejada.

Este capítulo apresentou um estudo sobre os requisitos de tolerância a faltas em sistemas distribuídos, as técnicas de replicação utilizadas e as vantagens e desvantagens de cada abordagem. Por fim foi apresentado um estudo sobre suporte de comunicação em grupo, uma das abstrações mais relevantes na área de tolerância a faltas, pois permite a concretização das técnicas de replicação.

Capítulo 3: CORBA

3.1. Introdução

A arquitetura CORBA foi publicada pela primeira vez em 1990 pela OMG *(Object Management Grou*p)[2], sua especificação aberta assegura a flexibilidade necessária aos sistemas distribuídos. Reusabilidade, portabilidade e interoperabilidade são palavras chaves desta especificação que tem como principal finalidade prover a padronização de uma arquitetura distribuída e orientada a objetos.

O componente central desta arquitetura é o ORB *(Object Request Broke*r), que pode ser definido primariamente como um canal de comunicação permitindo o acesso a objetos distribuídos de forma transparente. Todos os objetos deste modelo são descritos através de uma linguagem de definição de interface IDL(*Interface Definition Language*). A IDL descreve um conjunto de operações que podem ser requisitadas pelos clientes e permite ocultar do cliente aspectos relacionados à implementação do serviço. Desta forma, mudanças na implementação do serviço não impactam à aplicação cliente. O uso de interfaces assegura a interoperabilidade de objetos construídos em ambientes completamente heterogêneos.

Este capítulo apresenta as terminologias e os aspectos conceituais do modelo de objetos definidos pelas especificações da arquitetura CORBA. Também é apresentada a extensões CORBA que tratam as propriedades de tolerância a faltas em aplicações distribuídas.

3.2. Arquitetura OMA

Todas as especificações da OMG estão baseadas na arquitetura OMA *(Object Management Archictecture)* que define um modelo para a comunicação entre objetos distribuídos. Esta arquitetura é composta de um Modelo de Objeto e de um Modelo de Referência. O Modelo de objetos descreve como os objetos são descritos e o Modelo de

[2] É uma organização formada por mais de 800 empresas com o objetivo de especificar um conjunto de padrões e conceitos para a programação orientada a objetos em ambientes distribuídos abertos.

referência caracteriza as interações destes objetos. A Figura 3.1 representa os elementos desta arquitetura [Westphall 1998][Wangham 2000][Fraga et.al 2001].

Figura 3.1 – Arquitetura OMA

Aplicações – Os objetos de aplicação são desenvolvidos especificamente para uma dada aplicação. Como estas interfaces consistem na aplicação propriamente dita, as interfaces não são padronizadas pela OMG.

Facilidades CORBA: Provê um conjunto de serviços relacionados a várias aplicações. As facilidades comuns podem ser divididas de acordo com o seu escopo de atuação: horizontais e verticais. As facilidades horizontais independem do domínio da aplicação e são divididas segundo quatro categorias: interface de usuário, gerenciamento de informação, gerenciamento de sistema e gerenciamento de tarefa. As facilidades verticais estão relacionadas ao domínio da aplicação, exemplos: telecomunicações, finanças, controle de estoque etc.

Serviços de objetos (COSS): São serviços (interfaces e objetos), independentes do domínio da aplicação, de propósitos gerais que são fundamentais para o desenvolvimento de aplicações formadas por objetos distribuídos, ou que fornecem uma base universal para a interoperabilidade das aplicações. Em conjunto com o ORB, os serviços de objeto (ou serviços CORBA) estão relacionados a aspectos de infra-estrutura do sistema distribuído. A OMG tem definido até agora vários serviços nesta camada, tais como: *Life Cicle Service, Persistence Service*, *Naming Service, Event Service, Concurrency Control Service, Transaction Service, Relationship Service, Externalization Service, Query Service, Licensing Service* e *Properties Service.*

ORB: O ORB provê um canal de comunicação entre os objetos distribuídos, fornecendo os mecanismos necessários para a invocação de objetos de forma transparente. As interações entre cliente e servidor seguem o modelo de comunicação RPC *(Remote Procedure Call).* Todas as requisições realizadas pelos clientes são enviadas em forma de mensagens contendo o endereço do objeto e os parâmetros necessários para realizar a operação requisitada.

Um cliente realiza uma requisição através da interface do objeto disponibilizada ao cliente ***(IDL stubs)***. A mensagem é serializada e o ORB se encarrega de localizar o objeto no servidor e transportar os dados de acordo com a sintaxe de transferência. No servidor, a requisição é passada ao **Adaptador de Objeto Portável** *(Portable Object*

Adaptor - POA) que tem a responsabilidade de ativar a implementação de objeto correspondente. O processo de deserialização da mensagem é feito pelo ORB e a interface do objeto correspondente no servidor (***IDL Skeleton***) realiza a execução da operação solicitada.

O cliente pode realizar a invocação de objetos no servidor de duas formas: através da interfaces estática (IDL *Stubs*), ou através da interface de invocação dinâmica (***Dynamic Invocation Interface - DII*).** Para permitir invocações dinâmicas, as interfaces de objetos CORBA devem ser armazenadas no **repositório de interfaces**. Uma invocação dinâmica permite ao cliente construir e invocar métodos no servidor em tempo de execução. O **repositório de Implementações** contém informações que permitem ao ORB localizar e ativar implementações de objetos. A Figura 3.2 apresenta os componentes da arquitetura CORBA

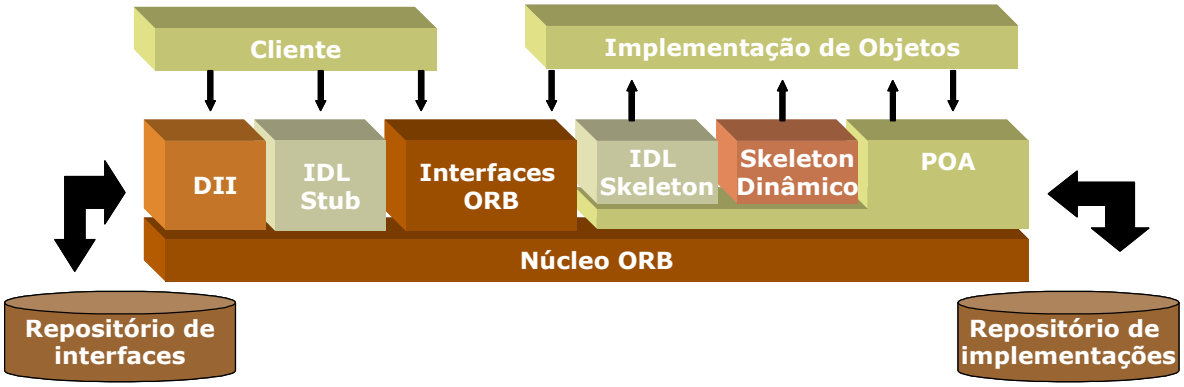

Figura 3.2 – Arquitetura CORBA

3.3. FT-CORBA

A especificação FT-CORBA (*Fault-Tolerance CORBA*) define um conjunto de interfaces de serviços para a implementação de técnicas de replicação em ambientes distribuídos e heterogêneos. Este modelo segue a abordagem de objetos de serviços COSS, ou seja, o ORB se abstrai da responsabilidade de prover os mecanismos para a concretização da tolerância a faltas. A abordagem de interceptação faz com que as mensagens enviadas aos objetos sejam capturadas e redirecionadas para os objetos de serviços apropriados, permitindo que o serviço de tolerância a faltas seja transparente para a aplicação. A abordagem de integração assegura ao FT-CORBA o suporte a comunicação em grupo. As referências de objeto passam a poder identificar tanto um objeto único como um grupo de objetos e o ORB é responsável por distinguir estas referências .

A arquitetura de tolerância a faltas no CORBA é apresentada na Figura 3.3. Os objetos de serviço que fornecem as funcionalidades básicas para a construção de aplicações tolerantes a faltas são [Fraga et. al., 2001][OMG, 2002]:

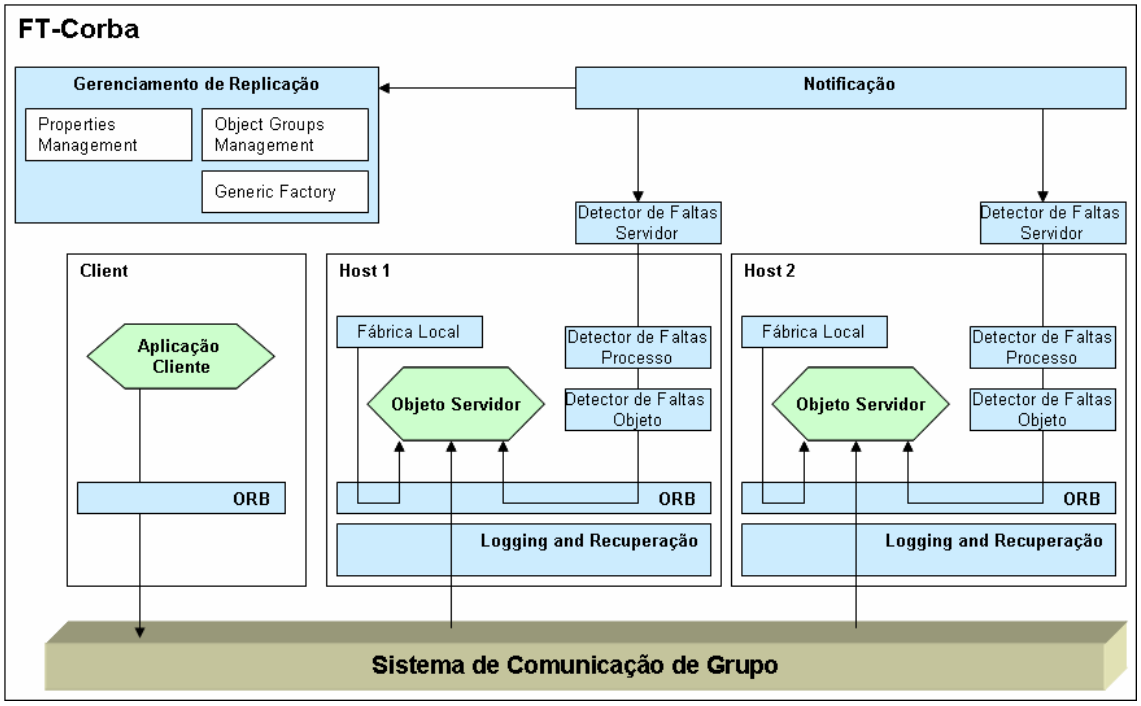

Figura 3.3 – Arquitetura FT-CORBA

3.3.1 Serviço de Gerenciamento de Replicação (SGR)

É um importante componente da arquitetura de tolerância a falta, interage com os objetos de Serviço de Gerenciamento de Propriedades, Serviço de Gerenciamento de Grupo de Objetos e Serviço Fábrica Genérica para oferecer um completo suporte para gerenciamento de replicação de objeto. No entanto, o SGR, em si, oferece apenas operações relacionadas com o notificador de falha. Basicamente, gerencia a referência de grupo de objetos (IOGR) do notificador de falha para que esta possa ser obtida pela aplicação ou pelos outros serviços da arquitetura. Contudo, as especificações permitem que a interface do SGR possa ser estendida com soluções proprietárias, com métodos similares para os outros componentes da arquitetura FT-CORBA.

3.3.2 Serviço de gerenciamento de propriedades (SGP)

Este componente torna possível ao SGR definir as propriedades de tolerância a faltas dos grupos de objetos criados. Essas propriedades para grupo de objetos podem ser estabelecidas estaticamente para um domínio, ou estabelecidas e modificadas dinamicamente durante a execução. E são definidas como [Fraga et.al 2001][OMG, 2002]:

ReplicationStyle: define a técnica de replicação utilizada, classificadas em: Replicação passiva fria, Replicação passiva quente, Replicação ativa;

FaultMonitoringGranularity: define o tipo de monitoramento de falha:

(i) sobre cada membro individual de um grupo;

(ii) sobre um membro representativo de um *host* contendo mais de um objeto (a falha do membro representativo é considerada como a falha de todos

objetos pertencente ao *host*, independente se é membro do mesmo grupo ou não);

(iii) sobre o membro representativo de um grupo em um *host*;

***MembershipStyl*e**: define se a criação de novas réplicas é feita em nível de aplicação de forma não transparente ou em nível de suporte, usando os objetos fábrica;

***ConsistencyStyl*e**: define se os procedimentos de transferência de estado *(checkpoin*t), *logging* e recuperação são realizados em nível de aplicação ou através dos serviços oferecidos na arquitetura de tolerância a faltas no CORBA;

***FaultMonitoringStyl*e**: define dois atributos, o PULL em que o objeto detector de falhas envia periodicamente mensagens para o objeto monitorado para verificar se está ativo. E o PUSH em que o próprio objeto envia mensagens periodicamente ao detector de falhas para indicar que está ativo;

***Factorie*s**: fornecem as informações do objeto fábrica. Por exemplo, a referência do objeto (IOR), a localização onde deve criar remotamente um membro de um grupo de objeto, e os critérios usados para criar esse membro;

***MinimumNumberReplica*s**: define o número mínimo de réplicas em um grupo para manter o grau de tolerância a falta desejado;

***FaultMonitoringIntervalAndTimeou*t**: define o intervalo de monitoramento *(pin*g) e o tempo de resposta *(timeou*t) do objeto monitorado para determinar se está faltoso;

***CheckpointInterva*l**: determina o intervalo de tempo entre cada atualização de estados.

3.3.3 Serviço de gerenciamento de grupo de objetos (SGG)

O serviço de gerenciamento de grupo oferece para a aplicação (ou para outros serviços FT-CORBA) métodos que permitem registrar e obter as referências de grupos de objetos (IOGR). Seu funcionamento é similar a um serviço de nomes, entretanto, o que o distingue é o gerenciamento de referências para grupos dinâmicos, gerencia diferentes versões de uma referência, modificadas a cada entrada ou saída de um membro no grupo. Fornece também métodos para a localização de membros em um grupo de objetos.

3.3.4 Fábrica genérica

Provê a criação e remoção de grupos de objetos, réplicas e objetos não replicados. A interface da fábrica genérica é implementada pelos objetos fábricas locais situados um em cada servidor. Desta forma, o SGR pode invocar os objetos fábricas locais para criar membros de um grupo de objetos e, também, permitir à aplicação invocar os objetos fábricas locais para criar objetos não replicados.

3.3.5 Serviço de gerenciamento de falha (SGF)

Neste serviço são definidas as interfaces dos serviços de detecção, notificação e análise de falhas. Os detectores são responsáveis por reportar em forma de registro as falhas detectadas ao notificador. O notificador recepciona estes registros, filtra as informações e envia ao analisador. O analisador desempenha uma importante função, ele é responsável por descartar registros duplicados ou desnecessários e verificar se há correlação entre os registros recepcionados. Os detectores utilizam mecanismos de *timeou*t e podem detectar uma falha em diferentes níveis :detecção de falha do objeto, detecção de falha no processo, detecção de falha no servidor.

3.3.6 Gerenciamento de *logging* e recuperação (SLR)

Este serviço tem a função de registrar o estado das réplicas, recuperar o estado de objetos faltosos, atualizar o estado de objetos backup e transferir o estado do grupo para novos membros que são inseridos no modelo de replicação. Na abordagem replicação passiva o mecanismo de *logging* registra em um dispositivo de armazenamento (*log*) o estado das réplicas e as ações da réplica primária para que em caso de falha o estado da réplica possa ser recuperado.

As requisições são registradas pelo mecanismo de *log* na ordem em que foram recebidas pela réplica primária. Em caso de falha o mecanismo de recuperação restaura o estado de um objeto *backup* para substituir o objeto primário que apresentou a falta e permite a re-execução das requisições na ordem correta. O armazenamento das *logs* pode ser implementado de forma centralizada ou distribuída.

O serviço de gerenciamento de *logging* e recuperação, especifica duas interfaces *Checkpointable* e *Updateabl*e. A transferência do estado do grupo é realizada pela interface *Checkpointable* e é utilizada nas abordagens de replicação passiva quente e ativa quando ocorre a entrada de um novo membro no grupo. A interface *Updateable* é utilizada para transferência de estado parcial na replicação passiva quente – em situações em que um *backup* já possui um estado inicial e é atualizado periodicamente, por exemplo, a cada mudança de estado do objeto primário [Fraga et. al.,. 2001].

3.4. Conclusões do capítulo

As arquiteturas baseadas em padrões abertos envolvem uma série de tecnologias e especificações que se mostram ideais para a implementação de sistemas distribuídos. O padrão CORBA possui uma arquitetura aberta e suporta sistemas com alta complexidade e heterogeneidade.

A especificação FT-CORBA trata as propriedades de tolerância a faltas definindo um conjunto de interfaces para as aplicações que possuem requisitos de alta disponibilidade e confiabilidade. Assim como os sistemas estão em constante evolução, as especificações CORBA passam por uma série de revisões, para que seja possível contemplar todos os requisitos apresentados pelos sistemas suportados por esta arquitetura.

Este capítulo apresentou os conceitos da arquitetura CORBA e o modelo estrutural da extensão FT-CORBA.

Capítulo 4: Serviços Web

4.1. Introdução

Quando a Internet começou a se popularizar, por volta dos anos 90, as tecnologias presentes permitiam a conexão com *websites*[3]. O HTML (Hiper Text Markup Language) era a linguagem que permitia a apresentação da informação presente na rede. Nos últimos anos, porém, novas tecnologias e padrões de desenvolvimento estão surgindo, permitindo uma maior integração entre os diversos aplicativos e serviços disponíveis na Internet.

Um dos modelos em crescimento são os serviços web que permitem a interoperabilidade entre aplicações heterogêneas através da utilização de protocolos amplamente utilizados e consolidados. Entre as principais vantagens na implementação dos serviços web pode-se destacar:

- Menor custo no desenvolvimento: permite a reutilização de componentes de software.
- Integração com sistemas legados: permite a integração com sistemas estabelecidos e operacionais.
- Melhores interfaces com parceiros comerciais: através de intercâmbio eletrônico de dados com baixo custo.

Neste capítulo serão apresentados o modelo conceitual e os padrões que compõem a arquitetura dos serviços web.

4.2. Modelo Conceitual

Os serviços Web são identificados por um URI (*Unique Resource Identifier*), e são descritos e definidos usando XML[4] (*Extensible Markup Language)*. Um dos motivos que os tornam tão atrativos é o fato deste modelo ser baseado em tecnologias padrão, em particular XML e HTTP. Os serviços web são usados para disponibilizar

[3] Conjunto de documentos estaticamente disponíveis na Internet pertencentes ao mesmo endereço (URL). URL é o endereço que permite a localização de informações na Internet.

[4] especificação técnica desenvolvida pela W3C, é definida como o formato universal para dados estruturados. W3C World Wide Web Consortium - entidade responsável pela definição de padrões relacionados Internet.

serviços interativos na WEB, podendo ser acessados por outras aplicações.

Entre as vantagens deste serviço pode-se destacar a interoperabilidade entre plataformas distintas e a capacidade de chamar um serviço web por intermédio de tecnologias presentes em toda a parte. Entretanto, para entender como o serviço web funciona faz-se necessário apresentar o seu modelo conceitual.

O modelo conceitual dos serviços web destaca dois elementos importantes: papéis e operações. Papéis são os diferentes tipos de entidades e as operações representam as funções executadas por essas entidades. [Hendricks 2002] [Potts, Kopack 2002][Chappel, Jewel, 2002][Shuping, 2003]

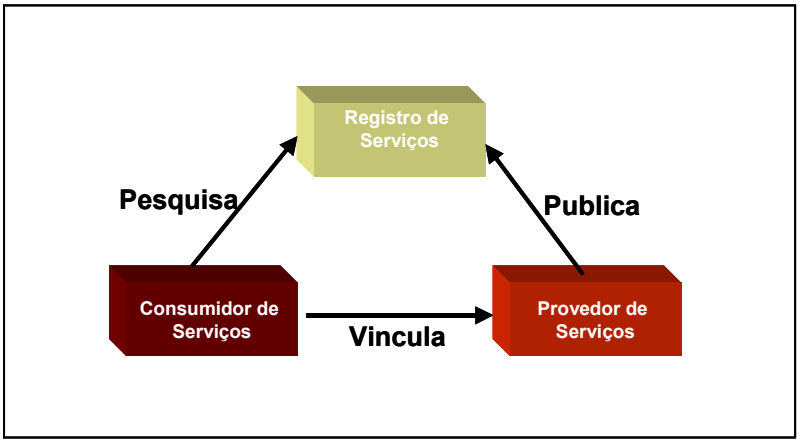

Figura 4.1 – Modelo Conceitual

A Figura 4.1 apresenta um modelo comum de serviços web identificando os três tipos de papéis e as operações que eles executam. Os papéis mostrados no diagrama são:

O Provedor de serviços — O provedor de serviços é a entidade que cria o serviço web. O provedor apresenta alguma funcionalidade comercial em sua empresa que será utilizada por outras empresas. Considerando o caso de uma empresa que comercializa livros através da Internet e que deseja apresentar seu serviço de pedidos como um serviço web. Primeiramente esta empresa precisa descrever o serviço em um formato padrão, que seja compreensível por qualquer outra empresa que possa usar esse serviço. Em segundo lugar, para alcançar um grande público, o provedor serviços deve publicar os detalhes sobre seu serviço em um registro central que esteja publicamente disponível para os interessados.

O Consumidor de serviços — Qualquer empresa que utilize um serviço web criado por um provedor é chamada de consumidor de serviços. O consumidor de serviço pode conhecer a funcionalidade de um serviço a partir da descrição disponibilizada pelo provedor. Para recuperar os detalhes, o consumidor realiza uma pesquisa sobre o registro onde o provedor publicou a descrição do seu serviço web. O mais importante é que o consumidor, a partir da descrição do serviço, pode obter do servidor o mecanismo para vínculo, podendo então chamar esse serviço web.

O Registro dos serviços — Um registro de serviços é a localização central onde o provedor pode relacionar seus serviços web.. Através do registro central, consumidores

podem encontrá-los e em seguida, usá-los. Tipicamente, informações como detalhes da empresa, os serviços por ela fornecidos e detalhes sobre cada serviço, inclusive detalhes técnicos, são armazenadas no registro do serviço.

No modelo de serviços web, é possível identificar três operações que são fundamentais — localização, vínculo e publicação. Para obter a comunicação entre as aplicações sem considerar detalhes de sua implementação é necessário que cada uma das operações realizadas pelas entidades sejam padronizadas. Visando obter a interoperabilidade foram criados os seguintes padrões:

- A *Web Service Description Language* (WSDL) é um padrão que utiliza o formato XML para descrever serviços web. Basicamente o documento WSDL define os métodos que estão presentes no serviço, os parâmetros de entrada/saída para cada um dos métodos, os tipos de dados, o protocolo de transporte usado e a URL da extremidade onde o serviço web está hospedado.

- O padrão *Universal Description, Discover*, and Integration (UDDI) permite que os provedores de serviços publiquem detalhes sobre suas empresas e os serviços web fornecidos em um registro central. Esta é a parte relativa à descrição (Description) do UDDI. Também fornece um padrão para permitir que os consumidores localizem os provedores e detalhes sobre seus serviços web. Esta é a parte relativa à descoberta (Discovery) do UDDI.

- O *Simple Object Access Protocol* (SOAP) é usado para trocar informações entre aplicações, independentemente do sistema operacional, da linguagem de programação ou do modelo de objetos.

4.3. Arquitetura para Serviços WEB

Para que seja possível a execução das operações fundamentais (publicação, localização e vínculo) em um ambiente de serviços web é necessário uma arquitetura básica que será apresentada em camadas [Hendricks 2002] [Potts, Kopack 2002][Chappel, Jewel 2002].

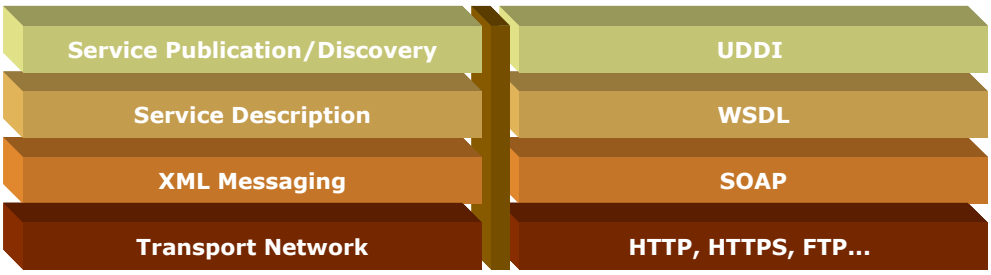

Figura 4.2 – Arquitetura de Serviços Web

Na Figura 4.2 os blocos à esquerda do diagrama são chamados de conceituais, enquanto os rótulos mostrados à direita correspondem às tecnologias reais que estão presentes em cada camada.

4.3.1 Camada de Transporte

Essa camada é responsável por tornar os serviços web acessíveis por intermédio de algum dos protocolos de transporte disponíveis, como HTTP, SMTP, FTP e outros. Os serviços web são construídos com base em padrões de comunicação existentes, que os torna independentes do transporte. No cenário atual, o HTTP é o protocolo de comunicação mais amplamente utilizado.

É importante observar que os serviços web podem ser distribuídos tanto pela Internet quanto internamente, dentro de uma empresa. Caso sejam implementados para acessos pela Internet, é recomendado escolher o HTTP como o principal protocolo de rede. Por outro lado, implementando serviços web dentro de uma empresa, é possível selecionar tecnologias de rede particulares como Messaging Standards (MS).

4.3.2 Troca de mensagens XML – SOAP

A próxima camada na pilha básica de serviços web se refere à troca de mensagens XML. Essa camada define o formato de mensagem usado na comunicação entre aplicações. O padrão usado com regularidade pelos serviços web é o SOAP, um padrão com base em XML para as informações entre aplicações independentemente do sistema operacional, do ambiente de programação e do modelo do objeto. Algumas das características do SOAP:

- O SOAP é um padrão superficial e simples, as informações são transmitidas no formato XML pelo HTTP. Este padrão não define um modelo de programação.

- SOAP não é vinculado a nenhum protocolo de transporte em particular. É possível deduzir que o SOAP equivale a um XML que opera por intermédio do HTTP, o que é perfeitamente razoável. Mas, ao mesmo tempo, pode ser vinculado a qualquer outro protocolo de transporte como SMTP, FTP e outros.

- SOAP é facilmente extensível por meio da XML.

Um exemplo de mensagem no SOAP

Uma mensagem em SOAP é conceitualmente representada conforme a Figura 4.3 abaixo:

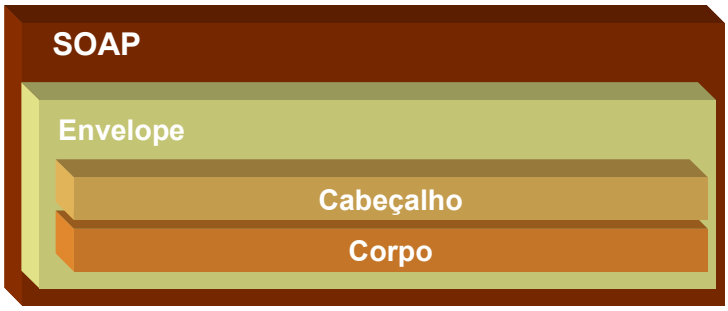

Figura 4.3. – Elementos de uma mensagem SOAP

Uma mensagem em SOAP consiste em um elemento envelope, que inclui um elemento cabeçalho (*Header*) opcional e um elemento corpo (*Body*) obrigatório. O envelope define um modelo para a descrição da composição da mensagem e como ela deve ser processada. O elemento cabeçalho, opcional, pode ser usado efetivamente pelos usuários para fornecer todos os dados adicionais necessários. O elemento corpo contém a carga útil da XML que codifica a chamada do procedimento, as respostas ou o relatório de falhas. O funcionamento da camada de transporte e a camada de troca de mensagens XML (XML Messaging) pode ser observado na Figura 4.4.:

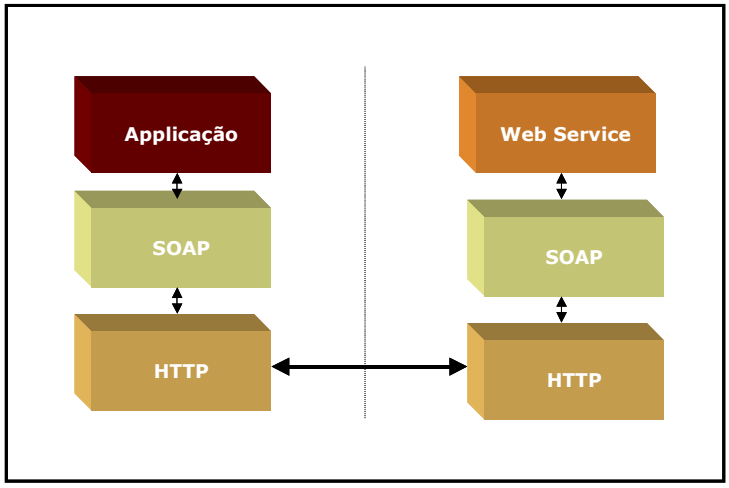

Figura 4.4 – Camada de transporte e troca de mensagens

A aplicação faz solicitações em SOAP. A solicitação em SOAP envolve a operação serviços web em uma carga útil de XML que é, então, transportada pelo protocolo HTTP. Na parte do serviço web, a camada de transporte passa a chamada para o servidor SOAP, que chama, a seguir, a funcionalidade apropriada, que foi exposta como um serviço web. Todas as respostas do serviço web são codificadas de volta para uma resposta do SOAP, que é transportada de volta, por meio do HTTP, para o cliente.

Usando a camada de troca de mensagens (SOAP) e a camada de rede de transporte (HTTP), é possível alcançar a comunicação entre as aplicações. O uso do SOAP concretiza, efetivamente, a comunicação entre aplicações, através de XML simples, com base em padrões abertos, sem estar preso a nenhum mecanismo proprietário. Entretanto, é possível observar que existe uma forte ligação entre as mensagens do SOAP, nas duas aplicações.

Para garantir a flexibilidade deste modelo é necessária uma descrição abstrata do serviço web, de maneira que tanto o consumidor quanto o provedor não precisem estar cientes de qual é o sistema operacional fundamental, qual a plataforma de programação e qual o modelo de objeto. Este é o principal objetivo da definição da próxima camada da arquitetura dos serviços web: a descrição do serviço [Hendricks 2002] [Potts, Kopack 2002][Chappel, Jewel 2002].

4.3.3 Descrição do serviço – WSDL

A camada Descrição do serviço, fornece um mecanismo ao provedor de serviços a

fim de descrever a funcionalidade proporcionada pelo Serviço web. A WSDL fornece o mecanismo, definindo a gramática de XML à descrição do serviço web como uma coleção de extremidades ou portas operando independentemente tanto nas mensagens orientadas a documento, quanto nas orientadas a procedimentos. A WSDL é para um serviço web o que o CORBA IDL é para o CORBA ou a Microsoft MIDL é para os componentes COM. A Figura 4.5 representa conceitualmente a descrição de um serviço web, de acordo com a especificação WSDL:

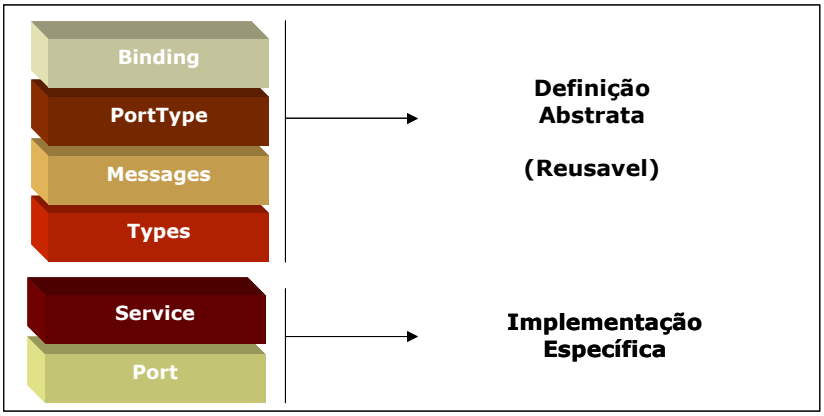

Figura 4.5 – Especificação WSDL

Para avaliar com mais precisão a WSDL, é importante notar que ela apresenta a definição de um serviço web em duas partes. A primeira representa uma definição abstrata independente do protocolo de transporte de alto-nível, de um serviço web, enquanto a segunda representa uma descrição de ligação específica para o transporte na rede.Em um alto nível de abstração, um serviço web contém os seguintes elementos:

Tipos de porta: Os elementos *portType* contêm um conjunto de operações representadas como elementos *operation*s, que estão presentes em um serviço web. É análogo a uma interface em Java ou C++. Uma operação pode portanto, ter uma mensagem de entrada e uma mensagem de saída. Além disso, ela pode ter apenas uma mensagem de entrada ou uma mensagem de saída, da mesma maneira que uma chamada de método normal.

Mensagens: Um elemento *message* contém uma definição dos dados a serem transmitidos. É semelhante ao parâmetro na chamada do método.

Tipos: O elemento *types* contém os tipos de dados que estão presentes na mensagem.

Vínculos: O elemento *binding* mapeia os elementos *operation*s em um elemento *potType*, para um protocolo específico.

Esta é a parte reutilizável da Definição do serviço web, pois representa apenas uma definição do Serviço. Para que uma operação possa ser executada em uma rede, deve-se usar um protocolo de transporte específico para enviar os dados pela rede. Pode-se portanto ligar a definição abstrata a vários protocolos de transporte, o que significa que uma definição de WSDL pode permitir que se defina um serviço web, independentemente do protocolo de transporte.

Resumindo, a descrição da WSDL de um serviço web contém uma coleção de

portas, cada porta se associa uma extremidade na qual o serviço web pode aceitar comunicações (como uma URL ou um endereço do e-mail) com um elemento *binding* particular que descreve as mensagens aceitas para esta porta. A figura 4.6 exibe como as três camadas da arquitetura de serviços web interagem:

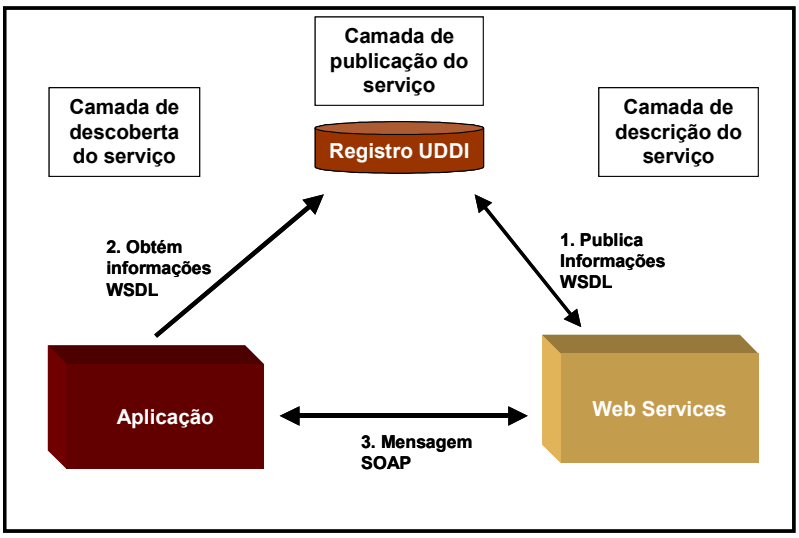

Figura 4.6 – Interação entre as camadas

A Figura 4.6 é uma extensão simples do diagrama de modelo de serviços web. No diagrama anterior (Figura 4.4), existia uma forte ligação entre as mensagens do SOAP que precisavam ser passadas entre as duas aplicações. Agora, com a camada de descrição do serviço web adicional, o provedor de serviços descreve o serviço web, através da criação e da publicação de um documento WSDL. Ele contém não apenas uma definição abstrata do serviço web, como também os detalhes de implementação (ligação) do serviço web. Isso significa que o consumidor do serviço, neste caso a aplicação do cliente, precisa apanhar o documento WSDL. A partir deste documento, não apenas as diferentes operações serão obtidas, que inclui mensagens e tipos de dados aceitos pelo serviço web, como também poderá ocorrer a recuperação da extremidade (por exemplo, URL) do serviço web pelo qual as mensagens do SOAP podem ser trocadas.

4.3.4 Publicação e descoberta do serviço – UDDI

A partir de um documento WSDL, um consumidor de serviço pode determinar os detalhes do serviço web, como as diferentes operações, tipos de dados, extremidades, protocolos de vínculo e outros.

Entretanto, existem outros fatores a serem, considerados. Esses fatores, devem ser examinados, sob o ponto de vista tanto do provedor de serviços quanto do consumidor de serviços.

Primeiramente, em vez de publicar o documento WSDL para cada possível cliente, um provedor de serviços estará bem servido se publicar as informações sobre seu serviço web em um registro central que esteja disponível publicamente para os consumidores interessados. Além de publicar apenas a descrição de seu serviço web, o provedor poderá publicar também, informações relacionadas ao negócio. Considerando que isso poderia ser publicado no mesmo registro central, a ajuda seria valiosa para a

disponibilização do seu serviço web para uma maior audiência.

Da mesma maneira, os consumidores de serviço podem querer encontrar os diferentes serviços Web que sejam disponibilizados pelos provedores de serviço. Eles podem querer avaliá-los independentemente, antes de integrar esses serviços Web às suas aplicações. Consequentemente, faz mais sentido, para eles, pesquisar o mesmo registro central, onde os provedores de serviço já publicaram seus detalhes. Os consumidores também podem realizar uma pesquisa de nível mais amplo. com base em categorias comerciais, para encontrar as empresas dessas categorias e os detalhes adicionais sobre os serviços web por elas oferecidos.

Existem muitas possibilidades além das apresentadas acima. Entretanto, o mais importante é a necessidade de um registro central, onde os provedores de serviços e os consumidores de serviços trabalham em conjunto para publicar e recuperar as informações apropriadas.

A especificação para a publicação e localização de informações comerciais é a especificação *Universal Discovery Description, and Integration* (UDDI). A especificação UDDI fornece uma estrutura comercial, usada para descrever um determinado negócio. A estrutura de negócios contém as seguintes informações:

- **Negócio**: contém as informações comerciais como o nome do negócio e o contato comercial. Contém uma ou mais instâncias da estrutura do serviço. A especificação UDDI também permite que uma definição comercial contenha um código de classificação que a identificará como pertencente a uma determinada categoria de negócios. como, por exemplo, instituições de empréstimo financeiras.

- **Serviço:** Uma estrutura de serviço captura os diferentes serviços Web fornecidos por este negócio. Cada serviço Web contém uma ou mais estruturas de especificação técnica. Uma empresa, pode ter, por exemplo, dois serviços Web; um serviço Web representa o estado do pedido de vendas e outro representando o estado do estoque. A especificação UDDI também permite que uma definição de serviço contenha um código de classificação que a identificará como pertencente a uma determinada categoria de serviços, como, por exemplo, a de verificação de crédito.

- **Especificação técnica:** As especificações técnicas contêm detalhes técnicos sobre um serviço Web. Uma das especificações técnicas é a especificação WSDL á qual um consumidor pode fazer referência, e determinar os detalhes técnicos sobre como invocar um serviço Web.

A Figura 4.6 mostra como as três tecnologias (UDDI, WSDL e SOAP) podem funcionar em conjunto, sob o porto de vista da arquitetura.

Considerando que o provedor de serviços decidiu apresentar determinadas funcionalidades comerciais como um serviço Web o mecanismo inteiro pode ser analisado através de uma sequência de etapas[Hendricks 2002] [Potts, Kopack 2002][Chappel, Jewel 2002]:

1. A primeira etapa do provedor de serviços envolve a codificação do arquivo WSDL. Existem diversas ferramentas disponíveis no mercado, atualmente, que auxiliam na geração do arquivo WSDL, a partir das definições do objeto existente. A seguir, é necessária a publicação de informações sobre si mesmo, que representem o negócio e a especificação técnica do serviço Web, como um arquivo WSDL, no registro UDDI central. Assim. descrição do serviço Web por meio da codificação do arquivo WSDL é capaz de capturar a camada Descrição do Serviço, enquanto a publicação de informações comerciais e o arquivo WSDL representam a camada publicação de serviço.

2. A aplicação do consumidor do serviço pode descobrir os serviços Web que ele esteja interessado em usar. A descoberta envolve não apenas a pesquisa de negócios e seus serviços, como também o carregamento das especificações técnicas mencionadas no arquivo WSDL. Essa etapa da corresponde à camada de Descoberta do Serviço.

3. Por último, a aplicação do consumidor de serviço utiliza o arquivo WSDL para determinar as mensagens que precisam ser passadas na comunicação com o serviço Web do provedor de serviços. Determina, também, as informações sobre o vínculo. A seguir, é feito o envio através de solicitações de SOAP e ocorre o recebimento das respostas de SOAP apropriadas. Esta etapa corresponde à camada de Mensagens e Transporte de XML na nossa pilha de serviços Web.

4.4. A Especificação *Web Services Reliable Messaging*

Em um ambiente ideal de execução de serviços web a rede jamais falha e todas as mensagens enviadas pelos clientes chegam ao destino sem problemas. Infelizmente em um ambiente real pode haver falhas de comunicação, perda e duplicação de mensagens, paradas de servidor entre outros problemas. Em uma transação negocial, o processo de transmissão de mensagens deve possuir a capacidade de reconhecer mensagens que foram recebidas pelo destino ou tomar as devidas providências quando a mensagem não é recebida corretamente. A especificação WS-Reliable Messaging (WS-RM) [WS-RM, 2004] endereça este problema através da definição de um protocolo para transmissão confiável de mensagens sobre o protocolo HTTP.

O objetivo da especificação WS-RM é permitir que aplicações enviem e recebam mensagem de forma simples confiável e eficiente mesmo na presença de falhas de rede. O WS-RM define um protocolo e um conjunto de mecanismos que permite os desenvolvedores de serviços web assegurar que mensagens são entregues de forma confiável entre dois pontos e suporta um conjunto de garantias na entrega de mensagens tornando a aplicação mais robusta.

Esta especificação atua no topo do protocolo SOAP e define meios do cliente reconhecer se as mensagens enviadas chegaram com sucesso ao seu destino. A idéia básica desta especificação é definir um meio de persistência de mensagens e continuamente tentar envia-las ao servidor. Uma vez que a mensagem é recebida pelo servidor, ela é retirada desta camada de persistência e o cliente é notificado que a mensagem foi entregue com sucesso.

O código implementado do lado do cliente para a transmissão da mensagem não difere muito de mensagens tradicionais SOAP, entretanto as diferenças ocorrem na comunicação entre o cliente e o servidor. Em vez de somente entregar a mensagem para o servidor, o cliente espera uma mensagem formal de reconhecimento. A especificação WS-RM introduz a noção de grupo de mensagens permitindo que o cliente agrupe uma série de mensagens e determine a ordem em que devem ser enviadas ao servidor. Existem quatro propriedades básicas de garantia de entrega de mensagens:

- AtMostOnce: As mensagens serão entregues no máximo uma vez, sem duplicação, ou uma mensagem de erro será retornada. Mensagens podem não ser recebidas.
- AtLeastOnce: Todas as mensagens enviadas serão entregues ou um erro será retornado. As mensagens podem ser recebidas mais de uma vez.
- ExactlyOnce: Todas as mensagens serão entregues, sem duplicação, ou um erro será retornado. (Esta propriedade é a combinação das duas propriedades acima)
- InOrder: Mensagens serão entregues na ordem em que foram enviadas. Esta propriedade pode ser combinada com qualquer uma das propriedades acima.

O WS-RM faz uso de outra especificação WS-Addressing [WS-ADD, 2002], que define como as informações referente a localização do serviço web deve ser apresentada dentro das mensagens SOAP. O WS-RM envia mensagens para o servidor através da localização do serviço contido nas mensagens SOAP. Além disso, o WS-RM utiliza os cabeçalhos do WS-Addressing para especificar a localização do cliente do serviço Web para que o servidor possa enviar as mensagens de confirmação ao cliente.

A Figura 4.7 ilustra as entidades e os eventos em uma troca confiável de mensagens entre um consumidor e um provedor de serviço. A camada *Reliable Messaging* localizada no consumidor é responsável por enviar as mensagens e receber a confirmação de recepção da camada *Reliable Messaging* localizada no provedor. A fim de facilitar a explanação dos eventos que ocorrem entre o servidor e o consumidor de serviços em uma troca confiável de mensagens, a camada *Reliable Messaging* localizada no cliente será denominada de RM origem e a camada *Reliable Messaging* localizada no provedor será denominada de RM destino.

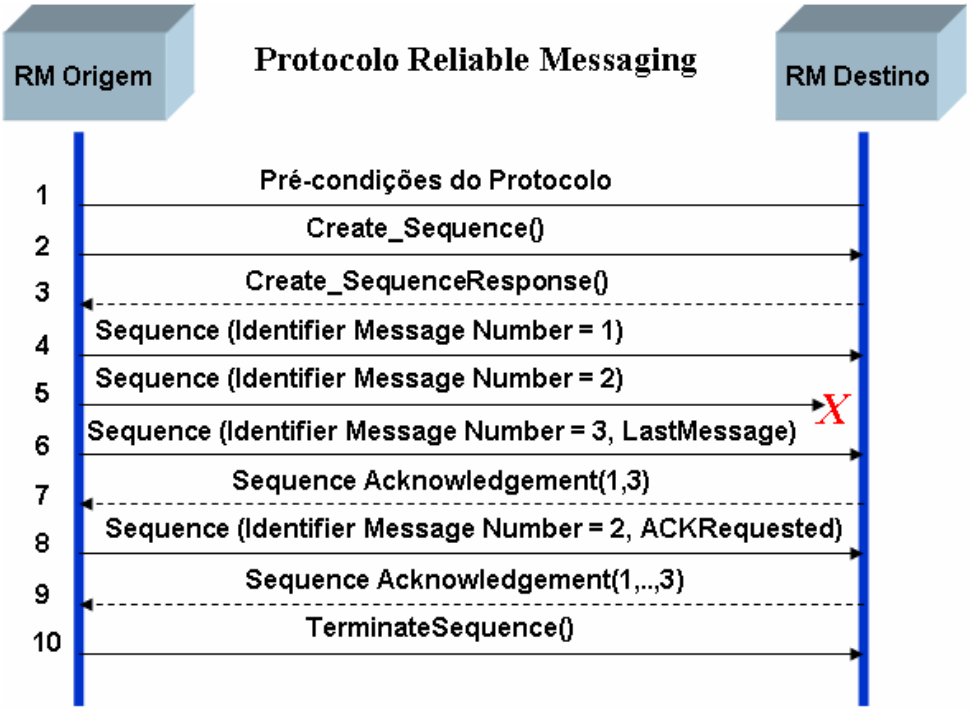

Figura 4.7 – Web Service Reliable Messaging

1. As pré-condições do protocolo são estabelecidas, incluindo a política de troca de mensagens entre a origem e o destino.

2. O RM origem requisita a criação de um elemento Sequence.

3. O RM destino cria o elemento Sequence a fim de prover ao conjunto de mensagens um identificador único.

4. O RM origem inicia o envio de mensagens começando com a mensagem número 1. Na Figura 4.7 a origem envia 3 mensagens.

5. A segunda mensagem é perdida durante a transição.

6. Na última mensagem o RM origem inclui um identificador notificando que é a última mensagem <LastMessage>.

7. O RM destino confirma a recepção da mensagem número 1 e número 3

8. O RM origem retransmite a segunda mensagem. Através do identificador da mensagem o RM destino reconhece a mensagem e a descarta em caso de recepção dupla da mensagem. O RM origem inclui um elemento <AckRequested> para que o RM destino envie a confirmação da recepção das mensagens.

9. O RM destino envia a confirmação da recepção das mensagens 1,2 e 3.

10. O RM origem finaliza a transmissão de mensagens através do <TerminateSequence>

O protocolo Web Service Reliable Messaging é formado por um conjunto de elementos que permitem identificar as mensagens, solicitar confirmação de respostas etc. Os principais elementos deste protocolo são descritos a seguir:

Elemento Sequence: o protocolo usa este elemento para administrar a entrega confiável de mensagens. Cada mensagem deve possuir um identicador único <Identifier> e cada mensagem dentro da seqüência deve possuir um número <MessageNumber>. O cabeçalho ainda pode conter um identificador <LastMessage> que sinaliza ao RM destino que a mensagem representa a última mensagem na seqüência. A sintaxe deste elemento é apresentada na Figura 4.8:

```
<wsrm:Sequence>
<wsrm:Identifier>http://fabrikam123.com/abc</wsrm:Identifier>
<wsrm:MessageNumber>10</wsrm:MessageNumber>
<wsrm:LastMessage/>
</wsrm:Sequence>
```

Figura 4.8 – WS-RM Elemento Sequence

Elemento Sequence Acknowlegedment: este elemento informa ao RM origem se a mensagem foi recebida com sucesso pelo RM destino. As mensagens de confirmação (Acknowlegedment) podem ser transmitidas individualmente ou incluídas nas mensagens de retorno. A Figura 4.9 apresenta exemplos de confirmação do recebimento de mensagens.

```
Exemplo 1: Confirmação das mensagens 1 a 10 (inclusive)

<wsrm:SequenceAcknowledgement>
<wsrm:Identifier>http://fabrikam123.com/abc</wsrm:Identifier>
<wsrm:AcknowledgementRange Upper="10" Lower="1"/>
</wsrm:SequenceAcknowledgement>

Exemplo 2: Confirmação das mensagens de 1,2,4,5,6,8,9 e 10. As
mensagens 3 e 7 não foram recebidas.

<wsrm:SequenceAcknowledgement>
<wsrm:Identifier>http://fabrikam123.com/abc</wsrm:Identifier>
<wsrm:AcknowledgementRange Upper="2" Lower="1"/>
<wsrm:AcknowledgementRange Upper="6" Lower="4"/>
<wsrm:AcknowledgementRange Upper="10" Lower="8"/>
</wsrm:SequenceAcknowledgement>

Exemplo 3: A mensagem 3 não foi recebida.

<wsrm:SequenceAcknowledgement>
<wsrm:Identifier>http://fabrikam123.com/abc</wsrm:Identifier>
<wsrm:Nack>3</wsrm:Nack>
</wsrm:SequenceAcknowledgement>
```

Figura 4.9 – WS-RM Elemento Sequence Acknowlegedment

Elemento Request Acknowlegedment: o propósito deste elemento é sinalizar ao RM destino que o RM origem esta requisitando uma confirmação de recepção de mensagens. Em qualquer momento o RM origem pode requisitar uma confirmação. A Figura 4.10 apresenta a sintaxe deste elemento.

```
<wsrm:AckRequested ...>
<wsrm:Identifier ...> xs:anyURI </wsrm:Identifier>
<wsrm:MessageNumber> xs:unsignedLong </wsrm:MessageNumber> ?
...
</wsrm:AckRequested>
```

Figura 4.10 – WS-RM Elemento Request Acknowlegedment

Elemento Sequence Creation: o RM origem deve requisitar ao RM destino o estabelecimento de uma nova Sequence. RM origem solicita a criação através da mensagem <CreateSequence>. O RM destino retorna a requisição através da mensagem <CreateSequenceResponse> ou através de uma mensagem <CreateSequenceRefused> em caso de falhas na criação da seqüência. A Figura 4.11 apresenta a sintaxe deste elemento.

```
<wsrm:CreateSequence ...>
<wsrm:AcksTo ...> wsa:EndpointReferenceType </wsrm:AcksTo>
<wsrm:Expires ...> xs:duration </wsrm:Expires> ?
<wsrm:Offer ...>
<wsrm:Identifier ...> xs:anyURI </wsrm:Identifier>
<wsrm:Expires ...> xs:duration </wsrm:Expires> ?
...
</wsrm:Offer> ?
</wsrm:CreateSequence>
```

Figura 4.11 – WS-RM Elemento CreateSequence

Elemento Sequence Termination: após o RM origem receber o <SequenceAcknowledgement> e completar o envio do conjunto de mensagens em uma seqüência, ele envia uma mensagem <TerminateSequence> para o RM destino indicando que a seqüência esta completa e que não serão enviadas mais mensagens relacionadas àquela seqüência. O RM destino pode seguramente reclamar por mensagens associadas com a seqüência ao receber uma mensagem <TerminateSequence>. A Figura 4.12 apresenta a sintaxe deste elemento.

```
<wsrm:TerminateSequence ...>
<wsrm:Identifier ...> xs:anyURI </wsrm:Identifier>
...
</wsrm:TerminateSequence>
```

Figura 4.12 – WS-RM Elemento SequenceTermination

4.5. Web Services x CORBA

As primeiras versões da especificação CORBA deixaram algumas áreas abertas à interpretação dos fornecedores. Como resultado, ORBS de fornecedores diferentes tiveram dificuldades de integração. Atualmente estas dificuldades foram superadas com as novas versões da especificação.

Para a implementação de sistemas baseados neste padrão faz-se necessário à configuração de portas especiais para permitir a comunicação entre cliente-servidor. Em muitos ambientes de rede os administradores relutam em abrir portas por representar brechas para possíveis intrusões. Para aqueles que interligam sistemas internos a Internet pode representar riscos, se as especificações de segurança deste padrão não forem implementas (CORBAsec).

Os serviços Web utilizam padrões baseados em XML, um padrão mais legível e extensível que a IDL utilizada pelo CORBA. Além disso os serviços web não são vinculados a nenhum protocolo de transporte em particular, podendo operar com mais de um protocolo simultaneamente e permitem o intercâmbio eletrônico de dados através de um padrão consolidado. Entretanto o CORBA pode manipular cargas de transação mais altas porque mantém uma conexão persistente entre clientes e servidores. Como é uma tecnologia mais madura, consequentemente, é mais padronizada. A Tabela 4.1 abaixo compara a arquitetura de serviços web e CORBA.

Item	Web Services	CORBA
Protocolo	SOAP, HTTP, XML Schema	IIOP, GIOP
Local de identificação	URLs	IORs, URLs
Interface	WSDL	IDL
Diretório de Nomes	UDDI	Naming Service, Interface Repository, Trader service

Tabela 4.1 – Arquitetura de serviços Web e CORBA

CORBA e serviços web não devem ser vistos como tecnologias mutuamente excludentes mas complementares. É possível extrair o melhor de ambos implementando sistemas com os requisitos desejáveis de distribuição, interoperabilidade, segurança e tolerância a faltas.

4.6. Conclusões do capítulo

Devido as suas principais características de independência de plataforma e interoperabilidade de aplicações, os serviços Web têm se mostrado como uma excelente alternativa para o intercâmbio eletrônico de informações entre aplicações.

Os padrões que formam o alicerce em que os serviços web estão fundamentados são definidos por grupos de consórcios a fim de garantir a interoperabilidade entre diferentes fornecedores. Através destes padrões abertos e tecnologias consolidadas estes serviços permitem a integração de processos de negócios completamente heterogêneos.

O principal objetivo deste capítulo foi apresentar o modelo conceitual e os padrões que compõem a arquitetura dos serviços web. Por fim, foi apresentada uma comparação entre a arquitetura CORBA e a arquitetura orientada a serviço.

Capítulo 5: Tolerância a Faltas em Serviços Web

5.1. Introdução

Atualmente não existem especificações padrão que tratem a tolerância a faltas nos serviços web, entretanto mecanismos que assegurem a confiabilidade e disponibilidade das informações são cruciais para a utilização desta tecnologia. Os esforços realizados nesta área, em uma visão geral, definem modelos para a detecção, notificação e tratamento de faltas transparentes para o usuário. Este capítulo tem por objetivo apresentar os trabalhos relacionados a implementação de serviços web tolerantes a faltas.

5.2. SOAP Tolerante a faltas

O modelo apresentado em *"FT-Soap: A Fault-Tolerant Web Service"* [Deron et. al., 2003] propõem uma extensão na arquitetura SOAP para atender os requisitos de tolerância a faltas. É baseado na especificação FT-CORBA e apresenta 4 funcionalidades principais:

- Gerenciamento de réplicas: realizado pelo componente ReplicationManager que gerência e constitui os grupos de serviços web protegidos.
- Gerenciamento de faltas: inclui os componentes FaultManager, FaultDetector e FaultNotifier responsáveis por monitorar os grupos de serviços protegidos.
- Mecanismos de recuperação e registros de eventos: inclui os componentes Recovery e Logging responsáveis por executar os procedimentos de recuperação em caso de faltas e registrar todos os eventos em um repositório centralizado.
- Mecanismos que asseguram a tolerância a faltas transparente ao consumidor do serviço.

As funcionalidades do SOAP foram estendidas e adicionados os seguintes conceitos: *Web Service Group(WSG)* e a figura do interceptador. O WSG é uma *tag* incluída no documento WSDL do serviço Web que define o serviço principal e as réplicas. O interceptador atua nas requisições de serviços, no consumidor e no provedor e é responsável por ativar as funções de tolerância à faltas.

Através do interceptador existente na camada SOAP do consumidor do serviço, faltas ocorridas no servidor primário não são percebidas pelo cliente. Quando ocorre a falta do servidor primário o interceptador redireciona a requisição para o servidor *backup,* baseado no grupo registrado no documento WSDL através da *tag* WSG. Esta abordagem provê ao modelo a tolerância a faltas transparente para o cliente. A Figura 5.1 apresenta os principais componentes deste modelo.

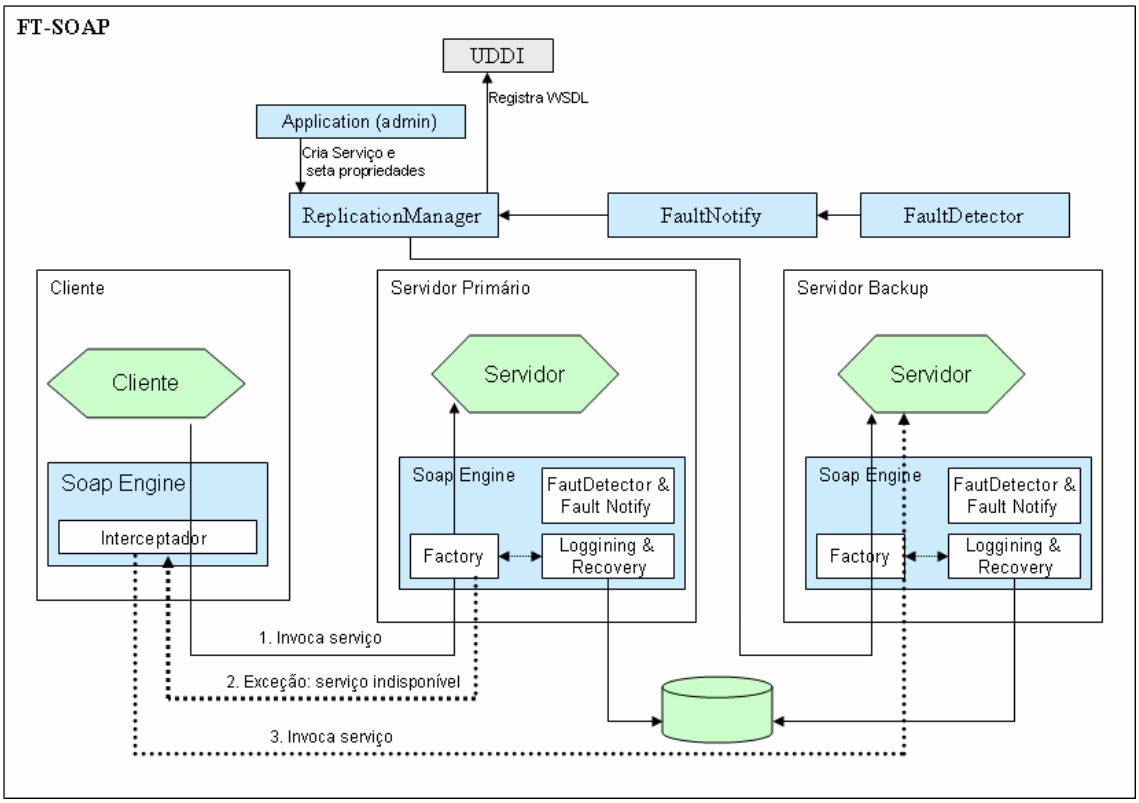

Figura 5.1 – FT-SOAP

Neste modelo o administrador do serviço web ao criar um serviço tolerante a faltas deve registrar o serviço em um grupo de replicação. O administrador deve registrar a sua política de tolerância a faltas, desta forma todos os procedimentos de recuperação estarão baseados na política registrada. Os procedimentos necessários para registrar um serviço Web tolerante a faltas são:

1. O administrador recupera o documento WSDL do gerenciador de replicação (RM) do UDDI.
2. O administrador registra as propriedades necessárias para a replicação do serviço tais como: propriedades da replicação e propriedades da monitoração do serviço.
3. O administrador requisita ao RM a criação do serviço.
4. O RM invoca a fabrica (factory) de cada membro do grupo para publicar o serviço baseado nas propriedades requeridas.
5. A fabrica registra todas as informações relativas ao serviço web para o FT-Soap e é responsável em retornar o documento WSDL do grupo para o RM.
6. O RM registra o WSDL do serviço no UDDI.

O mecanismo de *logging* registra todos os eventos em um sistema de arquivos centralizado. O funcionamento deste mecanismo pode ser descrito em 3 passos:

1. O cliente faz uma requisição para o servidor principal para a execução do serviço Web. Obtém as informações do servidor principal e das replicas através do WSG.
2. O interceptador SOAP localizado no servidor principal é invocado para completar a requisição. O interceptador invoca os mecanismos de log para registrar todas as informações necessárias.
3. O mecanismo de log recebe a notificação de log e escreve as informações em um sistema centralizado.

Quando ocorre uma falta no servidor principal o componente FaultDetector reporta ao componente FaultNotifier. O componente FaultNotifier é responsável em notificar a falta ao componente ReplicationManager definido para o grupo. Neste ponto o ReplicationManager inicia o processo de recuperação do serviço no servidor *backup,* modifica as informações referente ao grupo do serviço no documento WSDL e publica o WSDL modificado no UDDI.

A avaliação realizada com o protótipo mostrou que o *overhead* causado pelo mecanismo de log é insignificante, mensagens com até 512 KB causaram um *overhead* de 350 milisegundos. Utilizando o mesmo cenário para a avaliação de utilização de CPU foi verificado que o percentual de utilização foi de 0,25%.

O FT-SOAP implementa os requisitos de tolerância a faltas e estende todo as funcionalidades e vantagens da arquitetura SOAP, sendo completamente compatível com sistemas implementados puramente em SOAP. A avaliação de desempenho mostrou que o modelo não causa impacto no desempenho dos serviços quando comparado com serviços Web tradicionais (não tolerantes a faltas).

5.3. Detecção e recuperação de Faltas em serviços Web

A principal característica da arquitetura orientada a serviços é a integração dinâmica e flexível de aplicações. Esta arquitetura permite o desenvolvimento de uma nova classe de aplicações distribuídas e aplicações em *grid*. A computação em grid é caracterizada por aplicações de vida longa e envolve um grande número de recursos computacionais.

O modelo definido pela tecnologia de serviços em grid é flexível, seguro, coordenado e permite o compartilhamento de recursos através uma coleção de recursos individuais. As aplicações em grid possuem requisitos como: interoperabilidade, independência de plataforma, descoberta dinâmica e tolerância a faltas.

O modelo descrito em "Transparent Fault Tolerance for Web Services based Architectures" [Dialani et al 2002] tem como principal objetivo realizar a detecção e a recuperação em situações de falhas nos serviços que compõem o grid. Este modelo não trata a tolerância a faltas através da replicação de objetos, mas através de mecanismos de *checkpoint* e *rollback*. A área de sistemas distribuídos tem estudado vários algoritmos que tratam a disponibilidade e confiabilidade dos sistemas, no entanto ainda não há padrões definidos para o desenvolvimento de serviços web tolerantes a faltas.

A Figura 5.2 apresenta uma visão geral da arquitetura proposta. No topo da figura estão os vários componentes responsáveis por prover a tolerância a faltas. Estes

componentes são específicos para uma instância de aplicação e são independentes da localização. Esta camada é denominada de camada da aplicação. A camada intermediária apresenta as modificações realizadas no ambiente do provedor de serviços. As modificações podem ser categorizadas como um conjunto de modificações realizadas na camada de mensagens e um conjunto de interfaces suportadas pelos serviços individuais. Esta camada é denominada de camada de serviços.

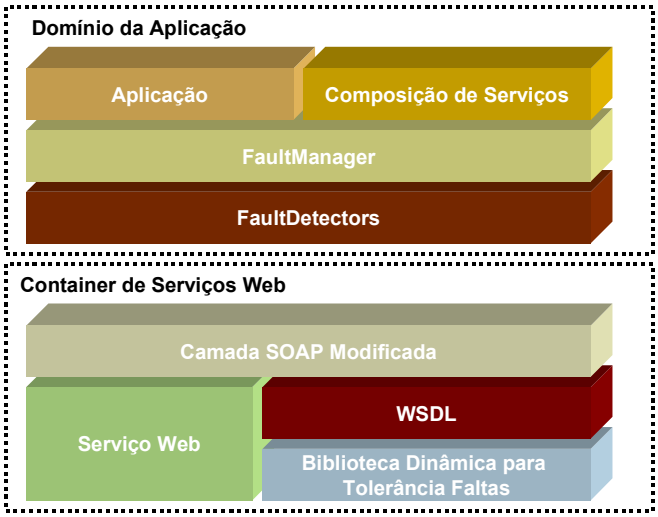

Figura 5.2 – Arquitetura de Tolerância a Faltas

Em linhas gerais as funcionalidades providas por este modelo são:

- Detectar falhas ou faltas
- Estimar o dano causado e decidir qual estratégia de recuperação utilizar.
- Reparar a falta.
- Restaurar o estado da aplicação.

A estratégia de recuperação utilizada por este modelo difere das estratégias utilizadas em modelos tradicionais de tolerância a faltas como o CORBA. Este modelo possui dois mecanismos de recuperação: global e local. O mecanismo local recupera uma instância de serviço sem intervenção do mecanismo global. O mecanismo global é responsável em recuperar o estado de toda a aplicação. Quando o mecanismo local não consegue recuperar a falta então o mecanismo global é notificado para prover a recuperação. Para coordenar as atividades entre as duas camadas o modelo provê um conjunto de interfaces. A camada de aplicação assume que uma aplicação agrega um conjunto de instâncias de serviços. O conceito de agregação de serviços é também conhecido como composição de serviços.

A camada de aplicação assume que existe uma descrição da composição do serviço que contém a lista de serviços colaboradores. Esta camada implementa um conjunto de componentes, tais como:

- *Application*: usa os serviços de uma composição. Uma aplicação pode acessar diretamente o componente de recuperação global ou permitir que o próprio *framework* interaja sobre o seu comportamento.

- *Fault Manager*: é um coordenador que interage com a aplicação ou com o *framework*. É responsável pela monitoração, diagnóstico e coordenação do *checkpoint* ou *rollback*. Este componente pode ser centralizado ou distribuído.
- *Service*: é a entidade que executa um processo dentro do ambiente de serviços Web.
- *Fault Detector:* detecta a mudança no ambiente e notifica o Fault Manager sobre a ocorrência da falha e fornece informações sobre o contexto da falha.

Um *Global Fault Manager* interage com um conjunto de serviços específicos na composição. Cada serviço necessita suportar um conjunto de interfaces para habilitar a comunicação entre o Local e o *Global Fault Manager*. Um *Local Fault Manager* coordena independentemente *checkpoints* e *rollbacks* de um serviço individual. Este componente monitora o serviços e suporta a interface *Fault Detector* para a notificação de faltas. O *Local Fault Manager* interage com a camada de mensagens para iniciar uma recuperação bloqueante ou não bloqueante, com ou sem resposta. O *Global Fault Manager* confia em um conjunto de detectores de faltas para enviar as notificações. Uma aplicação pode registrar uma lista customizada de detectores que suportam serviços individuais. A camada SOAP modificada provê o registro de mensagens *(log)* e também seletivamente suspende a comunicação entre os serviços para habilitar a recuperação local. A capacidade para suspender a comunicação facilita o processo de *rollback,* permitindo isolar um conjunto de serviços afetados por uma falta. Em resumo, as modificações realizadas na camada de serviços e na camada de aplicação permitem a tolerância a faltas baseadas nos mecanismos de *checkpoint* e *rollback.*

Este modelo provê modificações na camada SOAP, bibliotecas para iniciar o *framework* e um conjunto de componentes adaptáveis aos diferentes tipos de aplicações. A composição do serviço é realizada de maneira estática, entretanto o modelo pode ser alterado para permitir a composição e descoberta dinâmica. A camada de aplicação pode ser implementada para fazer parte do contexto de execução ou para ser uma instância de serviços web. Em qualquer um dos casos o *framework* inicia um *Global Fault Manager* para realizar as atividades de *checkpoint / rollback* entre as instâncias de serviços web. O Global Fault Manager utiliza a descrição da composição para localizar e se comunicar com o Local *Fault Manager*. O Local *Fault Manager* é implementado como uma biblioteca que pode ser adaptada dinamicamente ao código do serviço. Este componente interage com a camada SOAP modificada para controlar o fluxo de mensagens durante a recuperação, e deve suportar *checkpoint e rollback* bloqueante e não-bloqueante. Em caso de falta o *Local Manager* categoriza a falta e ativa os mecanismos de recuperação. Em certos casos é possível a recuperação do serviço individualmente através do retorno do seu estado. Em casos onde é necessária a recuperação completa dos serviços o *Local Manager* tenta recuperar o máximo de serviços possíveis e escala a falta para o Global Manager. Após receber a notificação o Global Manager inicia o *rollback* de todos os serviços afetados. O conjunto de dependências para recuperação pode ser provido pela aplicação. Adicionalmente os detectores de faltas podem prover o conjunto de serviços dependentes. O *rollback* também é realizado como uma operação *two-phase commit.*

Este *framework* assegura o baixo acoplamento e permite que outros mecanismos de tolerância à falta sejam adicionados aos componentes Global Fault Manager e Local Fault Manager. A utilização de componentes replicados como mecanismo de tolerância

a faltas não faz parte do escopo desta arquitetura. Informações referentes a avaliação de desempenho não foram fornecidas para este modelo.

5.4. Distribuição de carga centralizada no cliente

O modelo introduzido em "*Client-centered Load Distribution: A Mechanism for Constructing Responsive Web Services"* [Ghini et. al. 2001] propõe um mecanismo para a construção de serviços web com requisitos de alta disponibilidade e tempo de resposta. Esta solução é implantada junto a aplicação cliente. Especificamente o mecanismo provê o cliente de um conjunto de serviços web replicados e distribuídos sobre a Internet.

O principal objetivo desse mecanismo é minimizar tempo de resposta percebido pelo usuário (URT – User Response Time), ou seja, o tempo entre a geração da requisição no *browser*, a recuperação de uma página e a renderização da página pelo *browser*. Questões de consistência da réplica não fazem parte do escopo deste mecanismo.

Em resumo, em vez de ligar um cliente à replica mais conveniente o mecanismo intercepta as solicitações do cliente para um página web e fragmenta cada requisição em um número de sub-requisições. Cada sub-requisição é enviada a uma diferente replica disponível concorrentemente. A resposta recebida dos servidores é reagrupada no cliente para reconstruir a página requisitada e então é entregue ao *browser* do cliente.

O mecanismo foi modelado para se adaptar dinamicamente as mudanças de estado na rede (roteamento, congestionamento, falhas nos links) e nas replicas servidoras (sobrecarga na réplica e indisponibilidade). Para finalizar o mecanismo monitora periodicamente as réplicas e seleciona em tempo de execução quais réplicas deverão processar as sub-requisições. Estas réplicas podem prover fragmentos dentro de um intervalo de tempo determinado que permite o mecanismo minimizar o URT (tempo de resposta percebido pelo usuário). Este mecanismo implementa efetivamente a distribuição de carga dos clientes entre as réplicas e foi denominado de distribuição de carga centralizada no cliente (C2LD).

O modelo C2LD é baseado em um modelo analítico específico. Este modelo é essencial para determinar o tamanho do fragmento que deve ser requisitado para cada réplica. As informações de monitoração da réplica são usadas para determinar o tempo que levará o processamento da requisição.

O modelo foi implementado no topo do protocolo HTTP e opera transparentemente em software de camadas de alto nível (*browser*). Para este propósito a implementação provê aos usuários do mecanismo C2LD procedimentos de configuração que permitem configurar a variável USD (tempo limite de reposta especificado pelo usuário) antes de acessarem um serviço web. Caso o usuário não faça uso dessa configuração um valor é previamente estipulado.

Tipicamente para acessar um serviço web um usuário inicia um *browser* e informa a URL do serviço. O *browser* invoca um método HTTP Get, inicia a variável USD e usa a URL para interrogar o DNS. O DNS mantém o endereço IP das N réplicas que implementam o serviço. Quando C2LD submete uma requisição para o DNS resolver uma URL o DNS retorna o endereço IP associado para todas as réplicas. Após obter o endereço das réplicas o C2LD interage com cada replica como apresentado na Figura 5.3.

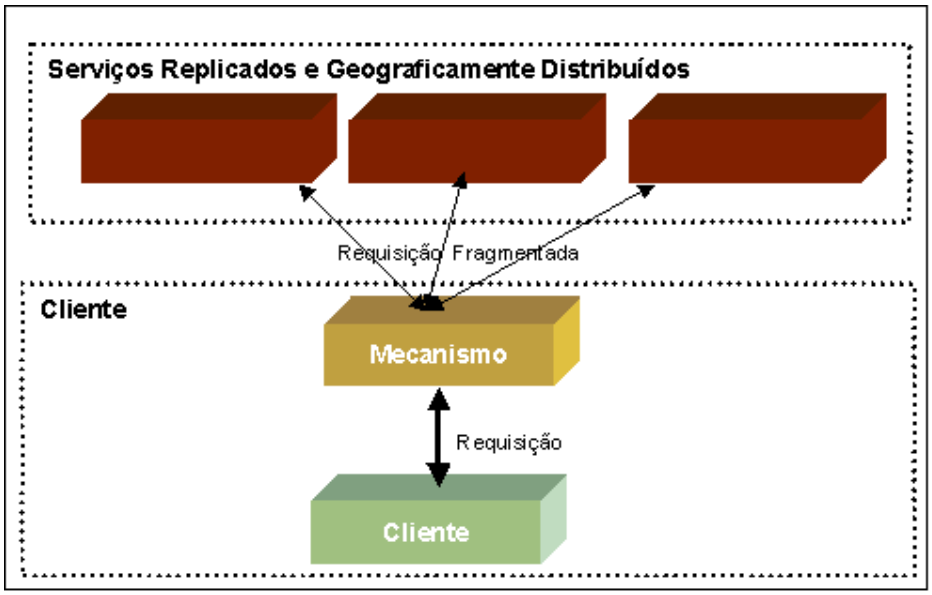

Figura 5.3 – Mecanismo C2LD

O C2LD invoca um método HTTP HEAD sobre cada réplica. A resposta da réplica é usada para:

1) obter o tamanho da página requisitada
2) estimar a taxa de dados da replica
3) calcular o tamanho do primeiro fragmento que pode ser obtido da replica

O mecanismo C2LD mantém uma variável global que indica se todos os fragmentos de uma página requisitada já foram retornados. Até que a página seja totalmente retornada o C2LD usa equações definidas em seu modelo analítico para computar o tamanho do fragmento que deve ser requisitado para cada réplica. Uma vez que o tamanho do fragmento foi calculado, o C2LD submete uma requisição HTTP GET para a replica.

Especificamente um fragmento de Z bytes é requisitado pela invocação do método HTTP GET com as seguintes opções "Variação: bytes Y-X" onde X e Y denotam os bytes correspondentes ao início e fim do fragmento requisitado de tamanho Z.

Para ajustar de forma adaptativa as possibilidades de flutuação de comunicação que ocorrem na internet (*delays*), o tamanho do fragmento é calculado toda vez que a página é requisitada. O tamanho é computado com base no valor URT experimentado na busca anterior de outros fragmentos.

Em alguns cenários as replicas podem não responder no tempo apropriado. Para isto o C2LD associa um tempo limite para cada requisição HTTP. Se o tempo limite for expirado antes de receber uma resposta o mecanismo assume que o servidor corrente está indisponível e o coloca numa lista de servidores com o estado de inativo. Os servidores neste estado são verificados periodicamente através do processo de monitoração e são ativados novamente quando o mecanismo percebe que estão respondendo adequadamente. As requisições que não forem atendidas por réplicas inativas são redirecionadas para réplicas ativas. Para aumentar o grau de paralelismo na busca de fragmentos dos documentos o mecanismo utiliza o modelo de *threads*.

O modelo foi validado através de um grande número de experimentos (4000 aproximadamente). Essencialmente esses experimentos consistiam em um programa cliente (*browser*) usando o mecanismo C2LD e realizando requisições de páginas web

de diferentes tamanhos através de quatro réplicas distribuídas geograficamente. A aplicação cliente e duas réplicas estavam localizadas na Itália respectivamente em Bologna, Cesena e Trieste. Uma réplica estava localizada na Inglaterra em Newcastle e outra nos Estados Unidos em San Diego.

Os parâmetros utilizados para a análise de performance foram:

i) o número de réplicas que implementam o serviço web

ii) as diferentes taxas de dados de cada réplica

iii) o período de monitoração das réplicas

iv) o tamanho do documento recebido.

Cada réplica mantém um conjunto de documentos de diferentes tamanhos com variação de 3 Kbytes a 1 Mbyte. O desempenho do mecanismo foi comparado com a requisição padrão de um documento (HTTP Get) sob as mesmas condições de tráfico na rede. O desempenho do mecanismo C2LD e o desempenho das requisições utilizando HTTP são equivalentes quando o documento tem até 50 kbytes. Entretanto para documentos com tamanho acima de 50 kbytes o mecanismo C2LD apresenta um desempenho superior. O número de réplicas é um fator determinante, quanto maior o número de replicas menor é o tempo de resposta retornado pelo mecanismo.

5.5. Replicação passiva em grid services

A combinação da tecnologia de *grid* e serviços web tem produzido uma atrativa plataforma para o desenvolvimento de aplicações distribuídas: grid services são representados por OGSI Open Grid Services Infraestructure [OGSA 2003].

Uma infra-estrutura de grid é uma coleção de recursos vulneráveis a vários tipos de falhas: paradas da aplicação, falhas do hardware, partições de redes e paradas não planejada para re-configuração de recursos. A maior parte das plataformas em grid possui mecanismos para tolerar ao menos alguns tipos destas falhas. Estes mecanismos tipicamente executam novamente as funções que falharam, utilizando mecanismos de *checkpoint* e *rollback* . O modelo apresentado em *"Fault-tolerant Grid Services using Primary-Backup"* [Zhang et. al.,. 2004] endereça a construção de serviços em grid com requisitos de alta disponibilidade através de serviços replicados em 2 ou mais servidores através da abordagem *primary-backup*.

As faltas toleradas pelo modelo proposto são classificadas como benignas como faltas por parada e perdas de mensagens. O servidor primário e os bakups devem rodar em *cluster* e devem ser gerenciados por um mecanismo responsável em detectar a falha no primário e gerenciar a recuperação. Produtos comerciais podem ser usados para este propósito.

Diferentemente dos serviços web, os serviços em grid são *stateful,* entretanto possuem um tempo de vida curto. A infra-estrutura disponibilizada pela OGSI permite que cada cliente escolha uma entre várias instâncias de serviços ou crie a sua própria instância. Interações entre um grid e uma aplicação cliente ocorre na forma de requisição-resposta. Em adição instâncias grid podem se inscrever usando uma interface especificada pela OGSI para receber notificações de mudança de estado.

A implementação da abordagem *primary-bakup* deve atender os seguintes requisitos:

1. Transferência do estado da aplicação
2. Detecção de falha.
3. Troca do primário para o servidor backup.

A notificação é um mecanismo natural para atender estes três requisitos porque a alteração de estado e a detecção de falhas são eventos assíncronos. A notificação provê um mecanismo simples para disseminação da informação para um número de partes interessadas. Neste modelo a notificação do tipo *push* foi utilizada para informar a alteração de estado e a notificação do tipo *pull* é utilizada para realizar a monitoração (detecção de falha).

A execução normal segue o seguinte fluxo: um cliente realiza a requisição para o servidor primário, quando o servidor primário termina de processar a requisição o estado é extraído e enviado para os *backups* via notificação. Quando o primário recebe dos *backups* todas as confirmações de recepção do novo estado, a resposta é enviada ao cliente. Os mecanismos de notificação são providos pela infra-estrutura de Grid. A falha do servidor primário é detectada se após um determinado período os *backups* não recebem a notificação de estado do primário, então um *backup* é eleito para substituí-lo. O novo primário envia uma notificação para o cliente que re-submete a requisição.

```
var target                              // ponteiro para o primário
var ops                                 // operações

INIT CLIENT ( )                         // invocado quando o cliente se liga ao grid
        (replica1, replica2, ...replican)  find replicas();
        target <- replica1;
        ops  <- 0;
                for each host 2 {replica2...replican} do
                        register_notification(&FAILURE HANDLER, host,FAILURE);
                        stub.init();
OP (params)
        var op                  // parametros da operação
        var done                        // semáforo que aguarda as execuções com sucesso
        var result                      // resultado das execuçoes
        op <- { params, &done, &result };
        ops <- ops [ &op;               // adiciona op para a lista em execução
        create_thread(&INVOKE OP, &op);
        wait_on_semaphore(&done);
        ops <- ops \ &op;               // remove op da lista
        return result;

INVOKE OP (op)
        var result                              // resultado das execuções
        result <- stub.op(op.params);           // realize as chamadas soap
        if result ≠ failure then
                op.result <- result;            // retorna o result para OP()
                signal(op.done); // wake it up

FAILURE HANDLER (new primary)
        target <- new primary;
        for each op E  ops do
                create_thread(&INVOKE OP, op);
```

Figura 5.4 – Pseudo-código utilizado no cliente

O pseudo-código mostrado na Figura 5.4, é interposto entre o cliente e a camada SOAP sem alterações na aplicação cliente. O método *init* é executado quando o cliente tenta realizar a ligação no *Grid* e registra o cliente para receber as notificações de falha do primário. O método *op* cria uma *thread* separada para executar o método *invoke_op* que submete a requisição do cliente. Para diminuir a duração do *failover* o mecanismo

não espera por exceções retornadas pelo método *op.* Assim que uma notificação de falha ocorre o mecanismo dispara outra *thread invoke_op.*

Nas réplicas o código é interposto entre a infra-estrutura de grid e a implementação do serviço para cada cliente *op* existe a implementação da operação no servidor. O serviço ainda deve implementar dois métodos *extract_state* e *inject_state* para que a trasferência de estado seja possível.

```
var rate_sending                                    // intervalo de envio de heartbeats
INIT PRIMARY ( )
        claim_notification_source(HEARTBEAT);
        claim_notification_source(STATE UPDATE);
        schedule(&HEARTBEAT GENERATOR, rate sending);   // programa as execuções

HEARTBEAT GENERATOR ( )
        notify_change(HEARTBEAT);                       // envia notificação

EXECUTE (request)
 var result                                             // resultado das execuções
 result <- check_previous_requests(request);
 if result = NULL then
        var state                                       // estado da aplicação
        result <- service.op(request.params);
        state <- service.extract state();
        notify_change_with_ ack(STATE UPDATE, state);   // aguarda acks
       return result;
```

Figura 5.5 – Pseudo-código utilizado no servidor primário

A Figura 5.5 apresenta o pseudo-código no servidor primário, onde cada operação é interceptada através do método *execute*. Primeiramente este método verifica se a requisição já foi processada, no caso do primário ter falhado antes de retornar a resposta para o cliente. Se a requisição ainda não foi processada então processa a requisição e em seguida envia a resposta para os *backups*. O método *notify_change_with_ack* notifica a alteração de estado e recebe a confirmação da recepção das mensagens dos *backups*. No processo de inicialização o primário se prepara para trabalhar com dois tipos de notificação *pull/push* e programa as mensagens de notificação de seu próprio estado para ser executado regularmente.

O pseudo-código mostrado na Figura 5.6 mostra a codificação nos servidores *backups*. Através do *state_handler* as alterações de estado são realizadas e o método *hb_handler* recebe as notificações de estado do servidor primário. Ambos armazemam o horário da ultima notificação *last_notification*. O *failure_detector* verifica qual foi a última notificação para assumir se o primário falhou ou não. Se a falha no primário é detectada um *backup* é eleito como novo primário e notifica o cliente imediatamente. Os outros servidores *backups* registram o novo servidor primário para receber as notificações de alteração de estado e as informações de estado do servidor primário. A configuração da réplica que será eleita como primário é realizada através do método *setup_senior* no momento de sua inicialização. Em situações onde o primário e o

bakcup sênior apresentam falhas ao mesmo tempo todos os *backups* verificam a falta do sênior e realizam a re-configuração através do método *init_backup*.

```
var rate checking                          // intervalo para verificar notificações
var last notification                      //ultima notificação
var primary_ is_ up                        // flag
var senior                                 // senior backup
INIT BACKUP ( )
        (replica1, replica2, ...replican)  find_replicas();
        Primary_ is_ up <- TRUE;
        if my_ url() = replica2 then
                senior <- TRUE;
        register_notification(&HB HANDLER, replica1, HEARTBEAT);
        register_notification(&STATE HANDLER, replica1, STATE UPDATE);
        claim_notification source(FAILURE);
        schedule(&FAILURE DETECTOR, rate checking);
        SETUP SENIOR(replica2);

SETUP SENIOR (senior url)
        if senior = TRUE then
                claim_notification_source(HEARTBEAT);
                claim_notification_source(STATE_UPDATE);
        else
                register_notification(&HB_HANDLER,senior_url,HEARTBEAT);
                register_notification(&STATE_HANDLER,senior_url,STATE_UPDATE);

FAILURE DETECTOR ( )
        if ( current_time() − last_notification) > rate_checking then
                if senior = TRUE then
                        switch_to_primary();
                        notify_change(FAILURE);
        else
                if primary_is_up = TRUE then
                        primary_is_up <- FALSE;
        else
                INIT BACKUP(); // backups
else
        if primary_ is_up = FALSE then
                primary_is_up <- TRUE;
                (replica1, replica2, ...replican) <- find_replicas();
                SETUP SENIOR(replica2);

STATE HANDLER (state)
service_inject_state(state);
last_notification_current_time();

HB HANDLER ( )
      last notification current time();
```

Figura 5.6 – Pseudo-código utilizado nos servidores backups

A avaliação de desempenho do modelo foi realizada utilizando duas diferentes implementações do mecanismo de notificação: *socket e call.* Call é um mecanismo provido pela própria infra-estrutura de grid, socket utiliza conexões TCP para realizar a notificação. Em todos os testes realizados a implementação socket apresentou tempo 20 vezes menor que implementação call. A avaliação de desempenho mostrou que o modelo *primary-backup* incrementa o tempo de resposta do cliente em aproximadamente 10% quando comparado com um serviço em grid sem replicação.

5.6. Tolerância a faltas suportada pelo Kernel

Diferentemente de outros modelos apresentados, que implementam a tolerância a faltas através de modificações e alterações na camada SOAP, o modelo apresentado em *"Transparent Fault-Tolerant Web Service with Kernel-Level Support"* [Navid, Yuval 2002] realiza alterações no *kernel* do sistema operacional e no servidor Web provendo um mecanismo de tolerância a faltas transparente para o cliente.

Neste modelo todas as requisições recebidas pelo servidor são registradas e enviadas para um servidor *backup.* As alterações realizadas no kernel do sistema operacional são denominadas de módulo kernel e provêem a implementação de um mecanismo de *multicast* permitindo que as requisições sejam enviadas para o servidor *backup* e o servidor primário. As alterações realizadas no servidor Web são denominadas de módulo servidor, este módulo atua como um manipulador e gerador de respostas para o cliente.

O endereço do servidor conhecido pelo cliente é mapeado para o servidor *backup*. O servidor *backup* recebe as requisições do cliente, após as operações de verificação da requisição no módulo kernel e registro da mensagem (*logging*), a requisição é enviada para o servidor primário. Após o processamento da requisição, o servidor primário envia uma cópia da resposta para o servidor *backup* e então envia a resposta ao cliente.

Através do registro das informações e da implementação deste mecanismo de *multicast*, quando é detectada uma falha no servidor primário o servidor *backup* possui toda a informação necessária para realizar o processamento e continuar a transmissão. A tolerância a faltas é assegurada durante a indisponibilidade de um dos servidores. A falha do servidor é detectada através de técnicas padrões para a verificação da disponibilidade do servidor (*heartbeats).* Este modelo assume que os dois servidores primário e backup devem estar localizados na mesma rede local, e que as conexões de rede entre cliente e servidor não sofrem faltas permanentes (Figura 5.7).

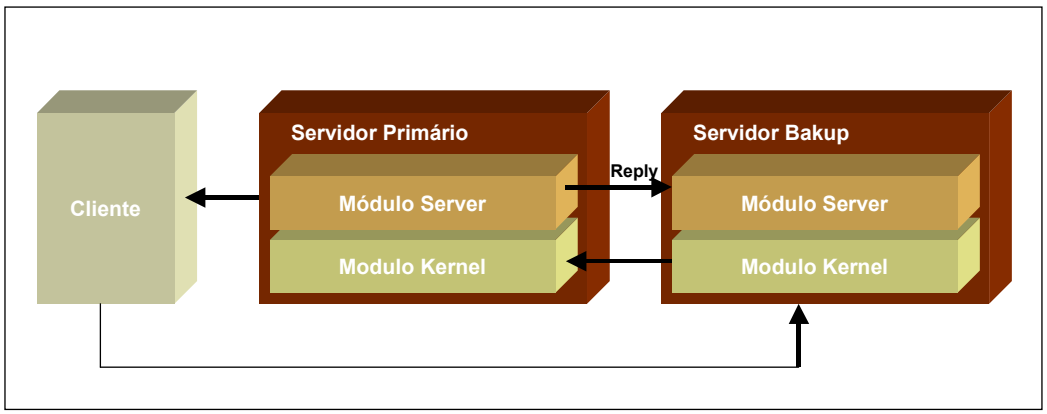

Figura 5.7 – Tolerância a Faltas suportada pelo kernel

O modelo foi implementado utilizando Linux e servidor Apache na avaliação de desempenho este modelo mostrou que a latência e o *overhead* são significantes devido a trocas de mensagens de reconhecimento (*acks*) entre o servidor primário e o *backup*.

Para que o *throughput* seja maximizado é crucial que exista uma conexão de rede dedicada entre o primário e o servidor *backup*.

5.7. Uma arquitetura tolerantes a faltas e a intrusões

O artigo "*An Architecture for an Adaptive Intrusion-Tolerant Server"* [Valdes et. al., 2002], descreve uma arquitetura tolerante a intrusões que pode ser aplicada em diferentes plataformas. A arquitetura proposta integra conceitos detecção de intrusões em sistemas distribuídos, tolerância a faltas e verificação formal. Em linhas gerais este modelo apresenta os seguintes componentes:

- Servidores de aplicação: estes componentes contêm os serviços utilizados pelos clientes. Este modelo apresenta serviços equivalentes replicados em um conjunto de servidores de aplicação. Estes serviços são implementados em diferentes plataformas. A redundância de serviços e a diversidade em sua implementação, assegura a tolerância a faltas e a intrusões.

- Proxy tolerante à faltas: é o componente central desta arquitetura. Possui a responsabilidade em mediar às requisições dos clientes e dinamicamente adaptar o seu sistema de operação de acordo com os reportes obtidos dos sistemas de monitoração. A tolerância a faltas neste componente é introduzida através de *proxies backups* que monitoram e detectam faltas no proxy primário. Quando ocorre uma falta no proxy primário um proxy backup é eleito como líder e substitui o proxy primário.

- Detector de intrusões: responsável por analisar o tráfico na rede e o estado dos servidores de aplicações e proxies reportando componentes suspeitos de intrusões. Os módulos que analisam a rede possuem hardware dedicado e os módulos que avaliam o estado dos servidores de aplicação e dos proxies residem no mesmo hardware destes componentes.

Quando uma requisição do cliente é recebida pelo proxy os seguintes passos são executados

1. O proxy primário aceita a requisição e filtra requisições mal formadas
2. Se a requisição é válida o proxy redireciona a requisição para um conjunto de servidores. O número de servidores que executará a requisição depende da política definida para o modelo.
3. Os servidores de aplicações executam a requisição e retornam a resposta ao proxy. Após um esquema de votação sobre os resultados a resposta é retornada ao cliente.
4. Os proxies auxiliares monitoram a transação e asseguram o correto comportamento do proxy primário.

São considerados suspeitos de intrusão um ou mais servidores que apresentem respostas diferentes da maioria das respostas retornadas. O modelo apresenta monitores que emitem alertas em acessos suspeitos. Em todo o sistema existem mecanismos de detecção de faltas e intrusões. Diferentes sensores cobrem diferentes áreas, as informações produzidas pelos monitores são agregadas e fornecem uma visão geral de todo o sistema.

Nesta arquitetura cada proxy envia um protoloco (*challenge-response)* para verificar o funcionamento dos servidores de aplicação e dos outros proxies. Este protocolo verifica se todos os componentes do sistema estão ativos, caso algum servidor não responda, um alarme é emitido. Este protocolo verifica também a integridade dos arquivos e diretórios localizados nos servidores e proxies.

Para cada arquivo cuja integridade é verificada um valor é computado e associado ao arquivo (*checksum*). Sempre que este protocolo verifica o arquivo ele compara com o valor do último checksum realizado. Caso algum arquivo seja corrompido o seu conteúdo é imediatamente substituído.

Para enviar a requisição aos servidores de aplicação, o proxy verifica quais servidores de aplicação processarão a requisição do cliente. Uma boa política deve prover um balanceamento das requisições escolhendo os servidores aleatoriamente então um intruso jamais saberá onde será processada a sua requisição.

A política de execução das requisições específica o número de servidores que devem responder a requisição, caso todos os servidores respondam a requisição da mesma forma então a resposta é enviada ao cliente. Se um dos servidores estiver indisponível ou não responder adequadamente este é identificado como um servidor suspeito de intrusão através dos monitores (Figura 5.8).

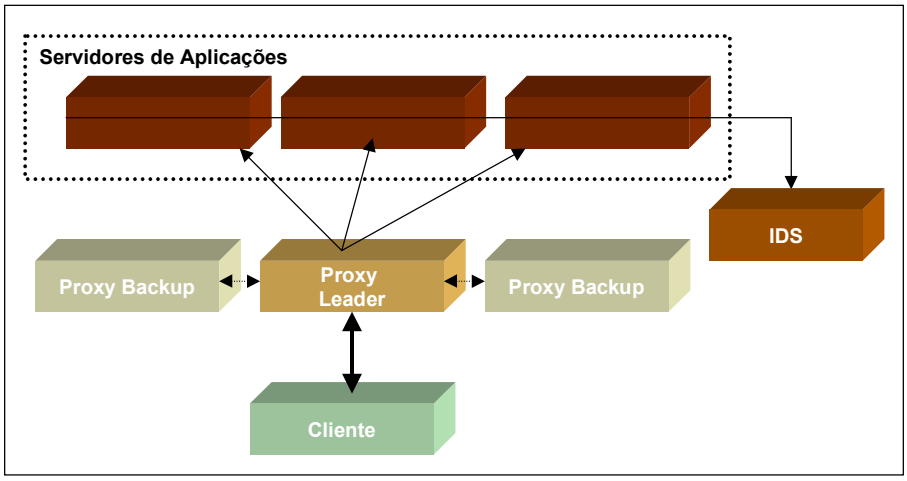

Figura 5.8 – Arquitetura tolerante a faltas e intrusões

Os monitores implementados incluem uma variedade de detectores de intrusão e mecanismos de emissão de alerta como:

- Detectores baseados na rede que verificam possíveis ataques em tempo real. São executados em uma máquina dedicada e verificam o trafego entre o *proxy* e os servidores de aplicações.
- Detectores baseados em servidores completam as funcionalidades dos detectores de rede monitorando os eventos da camada do sistema operacional.
- Detectores baseados na aplicação sensores que analisam as requisições http.
- Blue Sensor: monitora o status operacional de servidores e *proxys*.

Os principais mecanismos de segurança da arquitetura proposta são:

***IDS*:** Detecta ataques através do trafego na rede. Detecta acessos inesperados. Emite alerta em situações criticas.

***Content Agreement*:** protocolo que verifica o comportamento dos servidores.

Adaptative Agreement policies: política de segurança que evidencia eventos suspeitos, reduz a probabilidade de conteúdo incorreto nos servidores de aplicações.

***Chalenge-Response*:** protocolo que verifica a integridade e a disponibilidade dos application servers.

***Proxy-Hardening*:** limita a vulnerabilidade do proxy. Verifica URLs mal formadas. Restringe a comunicação dos servidores de aplicações com as aplicações clientes.

***Online Verification*:** monitora o comportamento do software.

Regular reboot dos proxies e servidores de aplicações: aumenta a confiabilidade evitando intrusões incrementais.

***Proxy peer monitoring*:** monitoração do comportamento do *proxy*.

A arquitetura proposta provê *proxies* e servidores de aplicação tolerantes a faltas através de redundância. O foco desta arquitetura esta em garantir a disponibilidade e a integridade das informações e não a confidencialidade. A arquitetura também provê escalabilidade servidores e proxies podem ser adicionados para melhorar o desempenho e a confiabilidade dependendo dos recursos disponibilizados. Este modelo não apresentou avaliação de desempenho.

5.8. Replicação ativa com diversidade de componentes

O potencial de integração das aplicações baseadas em serviços sugere que muitas destas aplicações podem ser vulneráveis a erros e falhas. Uma das características mais atrativas da tecnologia *grid* em arquiteturas orientadas a serviço é a habilidade de um cliente executar tarefas ou invocar serviços sobre um conjunto de recursos dinamicamente. Estes recursos devem ser autônomos e extremamente heterogêneos. Entretanto não é possível garantir confiabilidade ou o desempenho destes recursos.

Para minimizar estas limitações, o modelo definido em "*Replication-based Fault Tolerance in a Grid Environment"* [Townend, Xu, 2004] propõe um esquema de tolerância a faltas que permite a votação sobre resultados processados por serviços web equivalentes. Estes serviços são disponibilizados por diferentes provedores e são implementados em diferentes plataformas. O principal objetivo deste modelo é detectar anomalias nestes serviços e verificar o seu desempenho.

Este modelo é baseado em um componente central denominado *FT-Coordination* que pode estar localizado diretamente na máquina cliente ou em uma máquina remota. Para utilizar o *FT-Coordination*, primeiramente o cliente informa as características do serviço que deseja utilizar. A partir destas informações este mecanismo contata um ou mais registros UDDI para localizar os serviços compatíveis com a descrição informada pelo cliente. Após determinar os serviços e a sua localização o *FT-Coordination* envia a requisição do cliente concorrentemente para todos os serviços eleitos. Após o processamento da requisição este mecanismo realiza uma análise sobre as respostas retornadas.

A resposta correta é assumida através da expressão ($n/2 + 1$) onde n corresponde ao número de réplicas do serviço, ou seja, é considerada válida a resposta retornada pela maioria dos serviços. Caso a maioria não seja atingida o FT-Coordination procura por outros serviços equivalentes e submete a requisição. A resposta somente é retornada ao

cliente quando pelo menos três serviços fornecem a mesma resposta caso contrário uma mensagem de erro é retornada.

Para que o FT-Coordination não seja um ponto crítico de falha são utilizadas diferentes abordagens para assegurar a tolerância a faltas no mecanismo como recuperação em bloco distribuída e a presença de um FT-Coordination backup. Este modelo pode ser observado na Figura 5.9. Não faz parte do escopo desta arquitetura garantir o determinismo entre as réplicas e não foram apresentadas informações referentes a avaliação de desempenho.

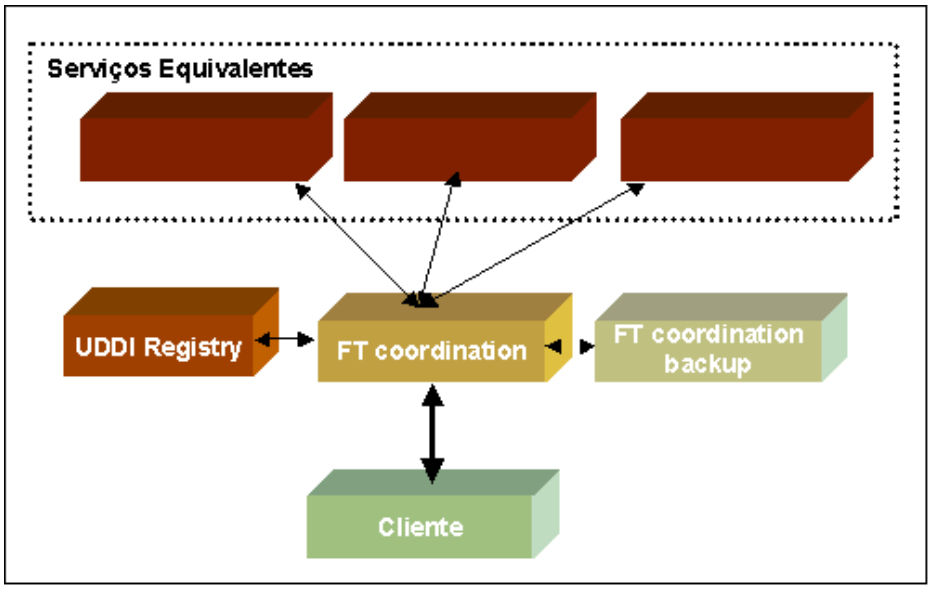

Figura 5.9 – Arquitetura FT-Coordination

5.9. Conclusões do capítulo

Embora a disponibilidade e confiabilidade sejam requisitos cruciais em aplicações críticas, os trabalhos relacionados à proposição de modelos tolerantes a faltas em arquiteturas orientadas a serviço são recentes. O modelo abordado em [Deron et. al., 2003] propõe extensões no padrão SOAP para alcançar a tolerância a faltas. Este modelo realiza alterações no documento WSDL inserindo informações referentes à réplica primária e às réplicas *backup*. A utilização de interceptadores na camada SOAP no cliente permite o redirecionamento da requisição para as réplicas em caso de falhas do primário. No servidor, os interceptadores adicionam componentes para registro de *log*, detecção de faltas e gerenciamento das réplicas. Este modelo apresenta suporte apenas às faltas por parada (*crash*) e não especifica o suporte a replicação ativa.

Em [Ghini et. al., 2001] é proposto um mecanismo para a construção de serviços web com requisitos de alta disponibilidade e tempo de resposta, através da técnica de replicação ativa. Todas as réplicas estão dispersas geograficamente e processam as requisições dos clientes. Entretanto este modelo pode apenas ser utilizado para requisições de leitura uma vez que o determinismo entre as réplicas não faz parte do escopo deste modelo.

O modelo proposto em [Aghdaie, Tamir, 2002] realiza alterações no *kernel* do sistema operacional e no servidor web provendo um mecanismo de tolerância a faltas

transparente para o cliente. Em comparação com os outros modelos, pode se considerar que este é o menos portável uma vez que a sua execução depende de um sistema operacional e um servidor web específico.

Proxies tolerantes a faltas são propostos em [Valdes et. al., 2002] permitindo a aplicação de técnicas de replicação passiva e ativa. Em adição apresenta componentes de monitoração e detecção propondo uma arquitetura tolerantes a faltas e a intrusões. Entretanto este modelo pode apresentar problemas no desempenho devido a utilização de protocolos que verificam a disponibilidade dos *proxies* e servidores de aplicações.

Os trabalhos abordados em [Dialani et. al., 2002], [Zhang et. al., 2004], [Townend, Xu, 2004] propõem modelos tolerantes a faltas para serviços web implementados e executados sob as especificações de serviços em *grid* [OGSA, 2003]. Em [Dialani et. al., 2002] a arquitetura proposta tem como principal objetivo realizar a detecção e a recuperação em situações de falhas, este modelo não trata a tolerância a faltas através da replicação de objetos, mas através de mecanismos de *checkpoint* e *rollback*. Este modelo apresenta um ponto de vulnerabilidade, caso o servidor que hospeda o serviço web fique indisponível a disponibilidade do sistema fica comprometida.

Em [Zhang et. al., 2004] é utilizada a técnica de replicação passiva através de mecanismos de notificação providos pela própria infra-estrutura de *grid*. [Townend, Xu, 2004] propõem a implementação de um mecanismo que executa um conjunto de serviços web equivalentes, entretanto implementados sob diferentes plataformas *(n-version)*. Após a execução um esquema de votação atua sobre as respostas retornando a resposta mais coincidente.

Em uma visão geral dos modelos apresentados todas as réplicas pertencem ao mesmo domínio. Estas abordagens são vulneráveis a falhas em roteadores, *gateways* e outros componentes que realizam interface com a rede. Os modelos que tratam a replicação ativa podem ser utilizados apenas para requisições de leitura, pois não garantem o determinismo entre as réplicas.

Capítulo 6: FTWeb – Uma Infraestrutura para Tolerância a Faltas em Serviços Web

6.1. Introdução

Para que todo o potencial dos serviços web possa ser explorado faz-se necessário a definição de uma infra-estrutura de desenvolvimento que atenda os requisitos de confiabilidade e alta disponibilidade. Esta infra-estrutura deve ser flexível o suficiente para que todas as características dos serviços web sejam mantidas.

Ainda são poucos os trabalhos que endereçam os requisitos de tolerância a faltas para os serviços web. O principal problema encontrado para a proposição de uma infra-estrutura tolerante a faltas nestes sistemas se concentra no fato dos servidores web não manterem uma conexão ativa durante todas as requisições do cliente e, como conseqüência, serem *stateless*. Por este motivo, aplicações críticas construídas sobre os protocolos Internet utilizam técnicas simplificadas. Por exemplo, usam mecanismos básicos que detectam a falha e direcionam futuras requisições para servidores redundantes. Estes mecanismos não são capazes de tolerar falta durante o processamento de uma requisição.

Atualmente não existem especificações padrão que tratem a tolerância a faltas nos serviços web. O presente trabalho apresenta uma proposta de utilização da técnica de replicação ativa para alcançar a tolerância a faltas em arquiteturas orientadas a serviços. O modelo proposto é baseado em uma infra-estrutura denominada de *FTWeb* que permite que os serviços estejam replicados e distribuídos sobre a Internet. Esta infra-estrutura possui componentes responsáveis por invocar concorrentemente as réplicas do serviço, aguardar o processamento, analisar as respostas processadas e retornar a resposta ao cliente. Estes componentes são baseados nos modelos e conceitos do padrão FT-CORBA da OMG [OMG, 2002] para o desenvolvimento de aplicações distribuídas e tolerantes a faltas. O objetivo desta abordagem é prover tolerância nas seguintes classes de faltas: parada, omissão e valor. Este capítulo apresenta a arquitetura do *FTWeb* e as funcionalidades providas pelos componentes que formam esta arquitetura. Para facilitar a explanação dos componentes a mesma seqüência utilizada neste capítulo, é também utilizada no Capítulo 7 que apresenta os detalhes de implementação da infra-estrutura.

6.2. Descrição da Infra-estrutura FTWeb

A idéia fundamental do *FTWeb* é a utilização da técnica de replicação ativa para alcançar a tolerância a faltas em arquiteturas orientadas a serviço. As réplicas de um determinado serviço são organizadas em um grupo e todas as réplicas livres de falhas recebem, executam e respondem as requisições enviadas pelos clientes. Para implementar a ordem total, necessária à replicação ativa, foi utilizada a abordagem do seqüenciador [Defago, Shiper 2000] por ser a mais adequada às características dos serviços web e, em termos algorítmicos, menos complexa que outras encontradas na literatura.

Através desta abordagem é possível replicar os objetos em servidores dispersos geograficamente (em diferentes domínios) e delegar a sua administração à infra-estrutura *FTWeb* através de uma interface bem definida. Esta infra-estrutura absorve as funcionalidades providas pelo *FT-CORBA* e fornece componentes que invocam objetos CORBA na forma de serviços web. O *FTWeb* é divido em 3 diferentes módulos: *WSClient Driver* que atua no domínio da aplicação, *WSDispatcher* responsável pelo gerenciamento das réplicas e execução do serviço e *WSWrapper* utilizado para integração com objetos CORBA. Estes módulos são apresentados na Figura 6.1.

6.2.1 WSClient Driver

O módulo *WSClient Driver* é responsável em detectar uma falha no componente *WSDispatcher Engine* e transferir o processamento da requisição ao *WSDispatcher Engine Backup* localizado em um servidor independente. Este componente pode ser definido de duas formas:

- Como um interceptador na camada SOAP localizada no cliente. Esta abordagem provê a tolerância a faltas de forma transparente para a aplicação cliente.

- Através da alteração dos componentes responsáveis pela invocação do serviço, permitindo que após a detecção da falha o endereço do serviço seja alterado e a requisição seja redirecionada para o *WSDispatcher Engine Backup*.

As duas abordagens tem como objetivo evitar que o *WSDispatcher Engine* seja um ponto crítico de falha. Este componente deve tolerar as faltas apresentadas pelo servidor de aplicações onde está hospedado o *WSDispatcher Engine*.

Caso o *WSDispatcher Engine* falhe após o processamento da requisição, ou seja, no momento da entrega da resposta ao cliente, o componente *WSClient Driver* irá transferir a requisição ao *WSDispatcher Engine Backup* que invocará os serviços replicados. Através de mecanismos de *log* contidos nas réplicas é possível verificar se a requisição já foi processada e então as réplicas simplesmente retornam a resposta ao *WSDispatcher Engine Backup*. Outros mecanismos como "Máquina de Estados Finitos" [Fred Schneider 1990] podem ser utilizados para evitar o re-processamento de requisições em caso de falhas dos componentes intermediários. A Figura 6.1 apresenta o fluxo normal e o fluxo em caso de falhas no *WSDispatcher Engine*

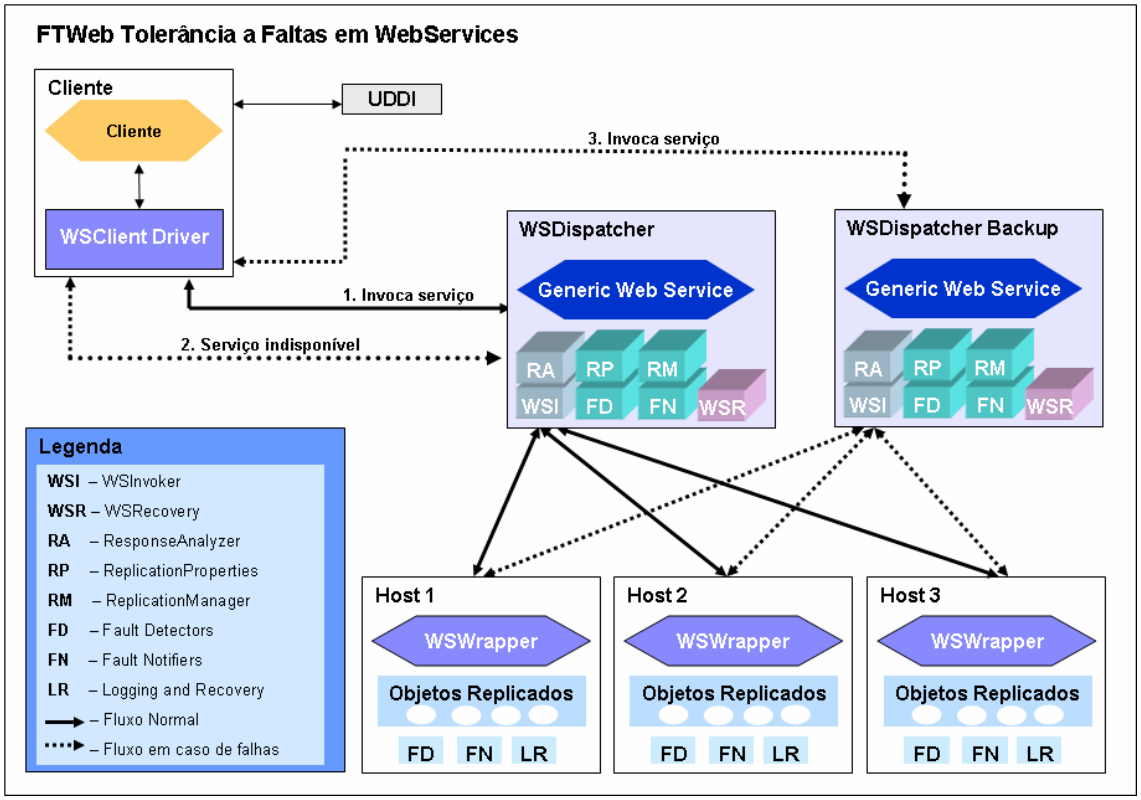

Figura 6.1 - Infra-estrutura FTWeb.

6.2.2 WSDispatcher Engine

O *WSDispatcher Engine* é o módulo central da infra-estrutura *FTWeb* e possui os mecanismos responsáveis por gerenciar réplicas, invocar concorrentemente as réplicas do serviço, analisar as respostas processadas, detectar e iniciar o processo de recuperação do estado em réplicas faltosas. O *WSDispatcher* é formado pelo conjunto de componentes descritos a seguir:

6.2.2.1 Generic Web Service

O componente *Generic Web Service* (Figura 6.1) é um serviço web genérico responsável em obter do cliente a referência do serviço web e os parâmetros necessários para a sua execução. Após a execução, este componente é encarregado de retornar a resposta obtida ao cliente. A sua utilização faz com que o cliente perceba como um único serviço, um conjunto de serviços replicados, independentes e dispersos geograficamente.

Para compor os grupos, necessários nas abordagens de replicação, foi utilizado o conceito de domínio de serviços [Tan et. al., 2004]. Um domínio de serviços permite a agregação e o compartilhamento de múltiplas descrições de serviços web (WSDL). As informações de ligação referem-se ao grupo, permitindo que vários serviços sejam virtualizados como um único serviço. Podem ser atribuídas regras ao domínio para administrar e controlar o comportamento dos serviços agregados. A Figura 6.2. exibe a diferença entre o modelo de domínio de serviços e o modelo convencional de serviços web.

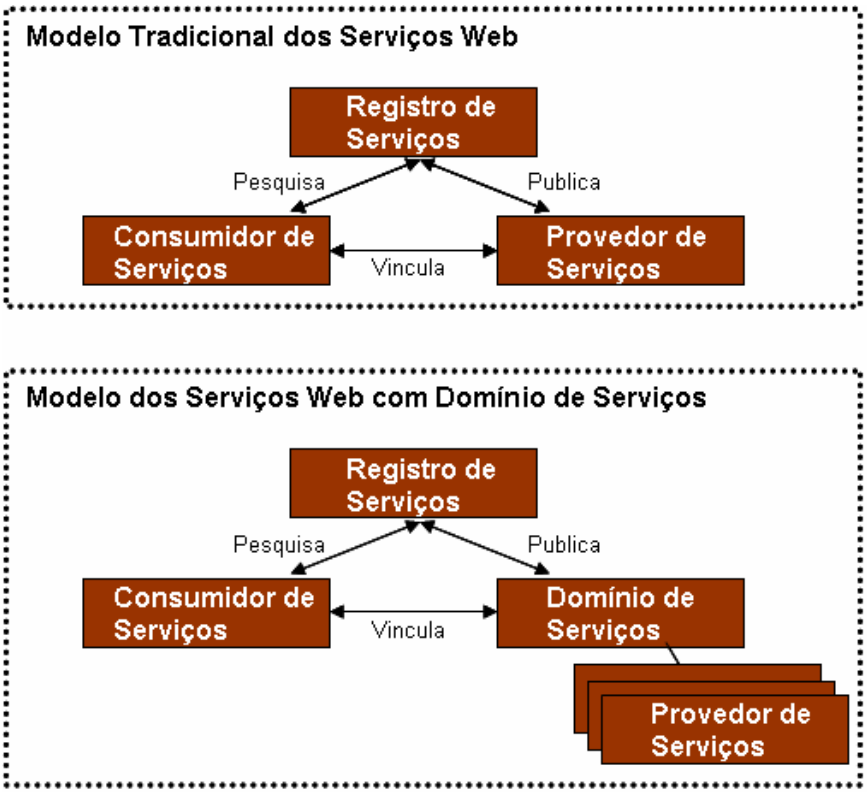

Figura 6.2. - Domínio de Serviços.

O *WSDispatcher Engine* possui um sistema de configuração (*Configuration System*) onde o administrador do serviço realiza a criação do grupo e indica através dos documentos WSDL as réplicas que farão parte do grupo. Neste sistema também são definidas as propriedades de replicação e gerenciamento de falhas.

6.2.2.2 WSInvoker

Este componente é apresentado na Figura 6.3. O seu funcionamento e a integração com os demais componentes podem ser descritos através de uma seqüência de 5 passos:

1. O cliente invoca o *Generic Web Service* informando a referência do grupo do serviço, o método a ser executado e os parâmetros necessários para a sua invocação;
2. O componente *Generic Web Service* invoca o *WSInvoker* e passa as informações obtidas do cliente;
3. *WSInvoker* interage com os componentes *Replication Manager* e *Replication Properties* para obter a localização das réplicas e as propriedades de tolerância a faltas definidas para o grupo;
4. O *WSInvoker* difunde a requisição, com garantia de ordenação FIFO, nas réplicas do serviço e gerencia a sua execução;
5. Após obter as respostas de todas as réplicas o *WSInvoker* invoca o componente *Response Analizer,* que realiza uma votação entre as respostas obtidas. O *WSInvoker* retorna ao *Generic Web Service* a resposta indicada pelo *Response Analizer*.

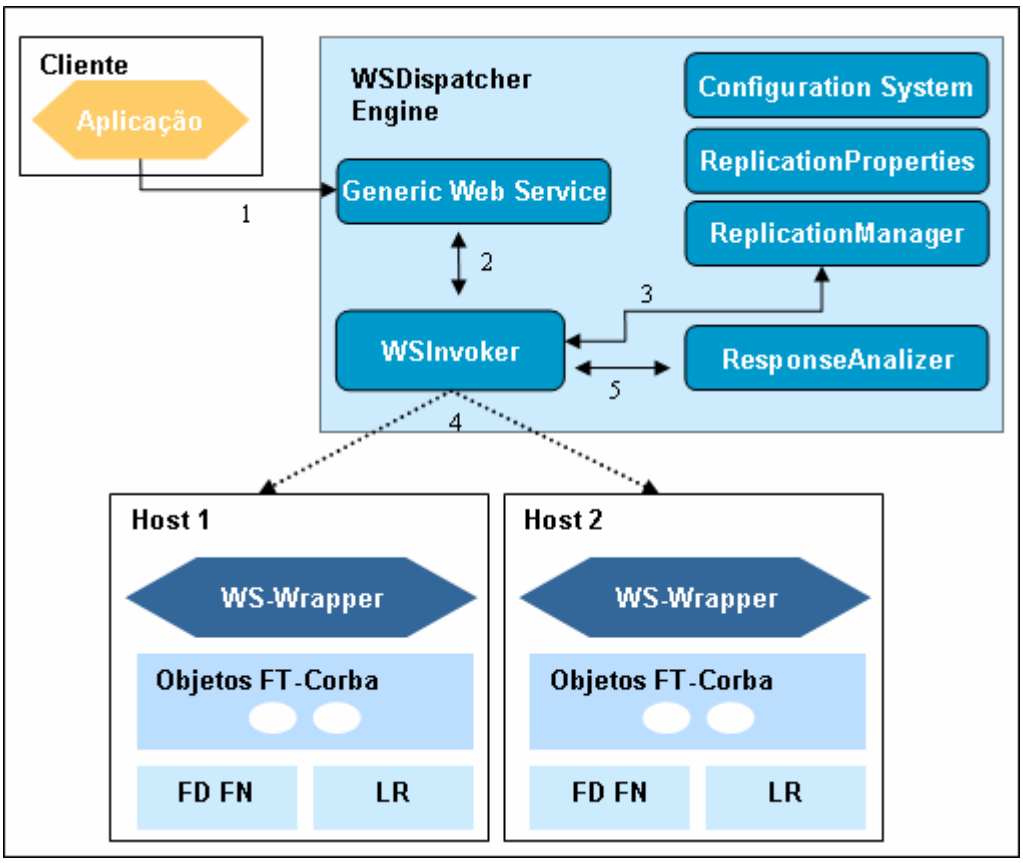

Figura 6.3. - Funcionamento do WSInvoker

O *WSInvoker* atua como um seqüenciador e assegura o determinismo entre as réplicas através de ordenação atômica [Defago, Shiper 2000]. Isso implica em um sincronismo na execução do serviço, o mesmo serviço não pode ser executado por mais de um cliente ao mesmo tempo. Através do determinismo alcançado pelo módulo *WSDispatcher Engine* é possível a sua utilização em serviços web *statefull,* ou seja, serviços que mantêm informações de seu estado durante as requisições do cliente. A Figura 6.4 apresenta o funcionamento do seqüenciador utilizado neste modelo.

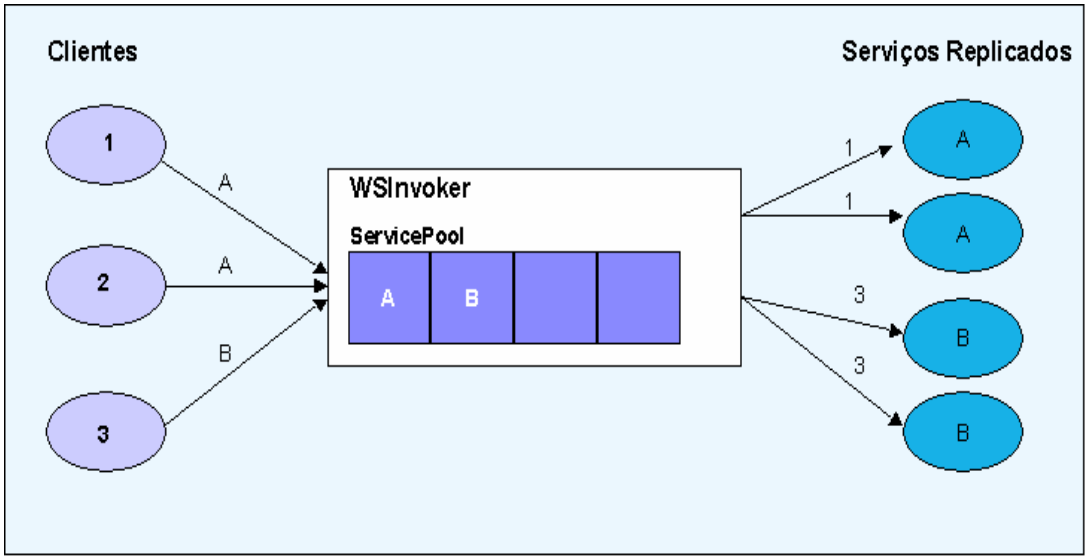

Figura 6.4- Funcionamento do Sequenciador

O *WSInvoker* possui uma estrutura denominada de *ServicePool,* nesta estrutura ficam registrados apenas os serviços em execução a fim de garantir que o serviço não seja acessado por dois clientes ao mesmo tempo. A Figura 6.4 apresenta 3 clientes requisitando a utilização de serviços. Os clientes representados nesta figura como 1 e 2 submetem requisições para o serviço A. O cliente 3 submete a requisição ao serviço B. Quando as requisições são repassadas ao *WSInvoker*, a estrutura *ServicePool* garante que apenas um cliente acessará o serviço enquanto o outro cliente aguardará a sua liberação. No exemplo representado pela Figura 6.4 o cliente 2 aguarda o término do processamento da requisição do cliente 1 e o cliente 3 acessa o serviço paralelamente. Caso outro cliente tente acessar o serviço B deverá obrigatoriamente aguardar o término do processamento da requisição realizada pelo cliente 3.

O repasse da requisição do cliente pelo *WSInvoker* é feito através de difusão confiável com ordem FIFO implementado através de múltiplas invocações ponto-a-ponto. Esta invocação é baseada na especificação *WS-Reliable Message* [WS-RM, 2004] na qual define uma extensão ao protocolo SOAP para que este suporte comunicações ponto-a-ponto confiável.

Se uma réplica apresenta falha no momento da sua execução ou não responde no tempo limite estabelecido nas configurações do serviço, o componente *WSInvoker* ativa os mecanismos de notificação para que o *ReplicationManager* retire a réplica faltosa do grupo do serviço. Neste caso, a réplica faltosa, fica fora do grupo até que esta tenha o seu estado restabelecido através dos mecanismos de recuperação. O *WSInvoker* não atua no controle transacional, se todas as réplicas apresentarem falhas o cliente deve possuir a lógica necessária para realizar a compensação de transações.

6.2.2.3 Response Analizer

O componente *Response Analizer* é opcional (Figura 6.3) e atua como um votador. Após a execução de todas as réplicas, o componente *WSInvoker* delega ao componente *Response Analizer* a análise de todas as respostas obtidas. A resposta com mais ocorrências é assumida. Este componente é utilizado para tolerar falta de valor.

6.2.2.4 Replication Manager

O componente *Replication Manager* estende as funcionalidades de gerenciamento de réplicas do *FT-CORBA* para os serviços web. Este componente controla dinamicamente a adição de novas réplicas e a remoção de réplicas faltosas segundo as regras definidas no *Replication Properties*. Este componente é notificado de alterações nos grupos de serviços através das seguintes abordagens:

- Através do componente *Fault Detector* que realiza o processo de monitoração e detecta réplicas faltosas.

- Através do componente *WSInvoker* quando o serviço apresenta falha na sua execução.

- Através do sistema de configuração quando um novo grupo de serviço é estabelecido ou quando uma nova réplica é adicionada em um grupo existente.

- Através do componente *WSRecovery* quando o estado de uma réplica que apresentou falha é reestabelecido.

6.2.2.5 Replication Properties

Este componente realiza o mapeamento das propriedades de tolerância a faltas definidas no *FT-CORBA* para a infra-estrutura *FTWeb*. Como mencionado anteriormente, o *WSDispatcher Engine* possui um sistema de configuração (apresentado na Figura 6.3, como *Configuration System*) que permite ao administrador do serviço definir as propriedades de replicação e gerenciamento de falhas. Através do sistema de configuração o componente *Replication Properties* obtém estas propriedades no formato XML. Estas propriedades são definidas como:

- *Replication Style:* define o estilo de replicação como replicação passiva fria, replicação passiva quente ou replicação ativa.
- *Monitoring Style*: define o estilo de monitoração como PULL ou PUSH. No estilo PULL o detector de falhas envia periodicamente mensagens para o objeto monitorado verificando se ele está ativo. No estilo PUSH, o objeto réplica envia mensagens periodicamente ao detector de falhas indicando que está ativo;
- *Monitoring Interval And Timeout*: define o intervalo de monitoramento (*ping*) e o tempo máximo de resposta (*timeout*) do serviço monitorado para determinar se está faltoso;
- *Response Timeout*: define o tempo de resposta (*timeout*) do serviço quando invocado pelo *WSInvoker*;
- *Recovery*: indicador do processo de recuperação do serviço. O estado do serviço pode ser recuperado automaticamente através de mecanismos fornecidos pela infra-estrutura *FTWeb* ou manualmente pelo administrador.

6.2.2.6 Fault Detector e Fault Notifier

Estes componentes estendem as funcionalidades de detecção e notificação de falhas do FT-CORBA para os serviços web. O estilo *PULL* de monitoração é utilizado por este componente: nesta abordagem o detector de falhas envia periodicamente mensagens para o serviço monitorado verificando se ele está ativo. Para que um serviço web seja monitorado é necessário que ele implemente a interface *PullMonitorable* que contém o método *isAlive()*.

Através da invocação deste método o componente *Fault Detector* monitora as réplicas. A monitoração é realizada conforme as propriedades obtidas através do *Replication Properties* e definidas no sistema de configuração.

Quando ocorre uma falha o componente *Fault Notifier* recebe uma notificação de falha do *Fault* Detector. O *Fault Notifier* notifica o *Replication Manager* que retira a réplica faltosa do grupo do serviço web. A Figura 6.4 apresenta o gerenciamento de falhas da infra-estrutura *FTWeb*.

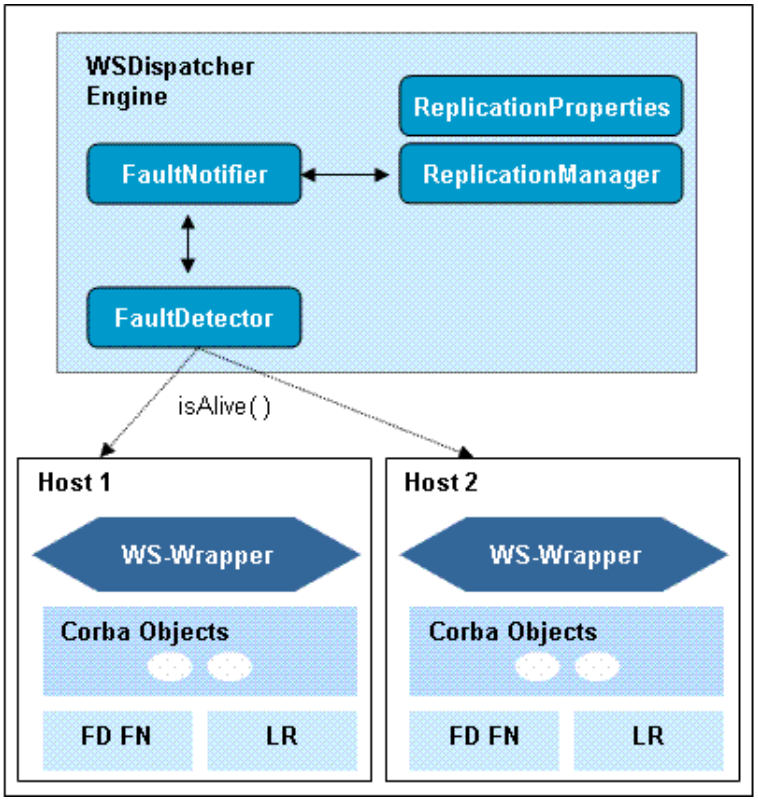

Figura 6.4– Gerenciamento de Falhas

6.2.2.7 WSRecovery

Este componente é responsável pela recuperação do estado de réplicas faltosas. O *WSDispatcher Engine* possui uma console de monitoração que exibe todas as réplicas que apresentaram falhas durante a requisição de uma transação ou durante o processo de monitoração. Esta console permite ao administrador do serviço, iniciar o processo de recuperação de uma ou mais réplicas. O próprio administrador pode informar o estado

do serviço em caso de falhas em todas as réplicas ou iniciar o processo de recuperação apenas das réplicas faltosas, este processo é denominado de recuperação manual.

Na recuperação automática o *WSRecovery* verifica periodicamente as réplicas faltosas, obtém o estado das réplicas não faltosas e através do mecanismo de votação definido pelo componente *Response Analizer*, restabelece o estado da réplica que apresentou a falha. O funcionamento deste componente é similar ao *WSInvoker* (Figura 6.3) no entanto a sua funcionalidade é invocada a partir da console de monitoração ou através da notificação do componente *Fault Detector* ao detectar uma réplica faltosa.

Tanto o processo de recuperação manual quanto o automático concorrem com os clientes na execução do serviço web, ou seja, o serviço web somente é liberado para ser utilizado pelos clientes após o término do processo de recuperação. Quando a réplica que apresentou a falha tem o seu estado re-estabelecido, esta é inserida novamente pelo componente *Replication Manager* no grupo do serviço web correspondente.

6.2.3 WSWrapper

Para explorar a integração entre as tecnologias CORBA e serviços web foi construído o módulo *WSWrapper* que realiza a interface entre o módulo *WSDispatcher Engine* e os objetos que processarão no provedor as requisições dos clientes. Este componente foi baseado nos modelos definidos em [Gokhale et. al 2003][Jandl et. al 2003].

Através deste componente, requisições SOAP são convertidas em invocações a objetos CORBA. O *WSWrapper* utiliza a interface de invocação dinâmica para invocar os objetos, podendo ser utilizado para executar qualquer objeto CORBA no provedor do serviço. Através desta abordagem é possível replicar os objetos em servidores dispersos geograficamente e delegar a sua administração ao *WSDispatcher Engine*.

Devido ao determinismo alcançado pelo componente *WSInvoker* e a integração com objetos CORBA alcançado pelo componente *WSWrapper,* é possível a utilização desta infra-estrutura na invocação de objetos replicados no servidores, implementados sob a especificação *FT-Corba,* aumentando a confiabilidade e disponibilidade dos sistemas.

6.3. Cenários

O *FTWeb* não afeta a operabilidade de serviços já existentes, podendo coexistir no mesmo ambiente, serviços executados sob o *FTWeb* e serviços web tradicionais. O desenvolvimento de serviços web tolerantes a faltas usando a infra-estrutura *FTWeb* é bastante simplificada, requer apenas a inserção dos métodos de monitoração e recuperação do estado da réplica fornecidos pela infra-estrutura. O *FTWeb* pode ainda ser utilizado em diferentes tipos de serviço web: *statefull*, *stateless*, síncronos e assíncronos.

Nos serviços web assíncronos é importante a utilização da especificação Web Service Reliable Messaging na invocação do serviço, no cliente e no *WSDispatcher Engine.* Esta especificação define um protocolo e um conjunto de mecanismos que permite os desenvolvedores de serviços web assegurar que mensagens são entregues de forma confiável entre dois pontos e suporta um conjunto de garantias na entrega de mensagens tornando a aplicação mais robusta.

A infra-estrutura *FTWeb* não restringe a localização dos clientes ou das réplicas que podem estar localizados no mesmo domínio ou dispersos geograficamente. O componente *WSDispatcher Engine* e o *WSDispatcher Engine Backup* podem estar localizados na mesma rede em cluster e outros componentes como balanceadores de carga podem ser utilizados para aumentar a disponibilidade do sistema. Entretanto esta abordagem centralizada da infra-estrutura em um único site torna o modelo vulnerável a falhas em roteadores e outros componentes que realizam interface com a rede.

Quando o *WSDispatcher Engine* e o *WSDispatcher Engine Backup* estiverem localizados em redes distintas são necessárias configurações específicas para este tipo de ambiente:

- A utilização de APIs que implementam a especificação Web Service Reliable Messaging na invocação dos serviços é indispensável a fim de garantir a tolerância a faltas na camada de rede.

- É necessária a utilização de dois componentes *Fault Detector* um para cada instalação do *WSDispatcher*, a fim de permitir que os grupos de serviço estejam consistentes.

- O *WSDipatcher Engine* definido como réplica apenas será acessado em caso de indisponibilidade do *WSDispatcher Engine* primário a fim de garantir o determinismo entre as réplicas.

- O *WSClient Driver* apenas realiza o redirecionamento das requisições quando é detectada uma falha no servidor de aplicações que hospeda o *WSDispatcher Engine*

- Toda alteração de configuração dos serviços realizadas através do sistema de configuração no primário deve ser replicada no *WSDispatcher Engine Backup* a fim de manter a configuração dos serviços consistentes nos dois componentes. Para realizar esta atividade a infra-estrutura *FTWeb* disponibiliza serviços web no servidor backup que são invocados pelo primário após a configuração de um serviço.

6.4. Abordagens de Replicação

Devido a flexibilidade da infra-estrutura *FTWeb* é possível a sua utilização nas abordagens de replicação passiva quente, passiva fria e semi-ativa. As técnicas de replicação foram abordadas no Capítulo 2 item 2.5.3. Os detalhes da utilização destas abordagens com a infra-estrutura *FTWeb* são descritos a seguir.

6.4.1 Replicação Passiva

Na replicação passiva somente um membro (primário) recebe, executa e responde as invocações dos clientes. As outras réplicas (*backups*) do conjunto tem a finalidade de substituir o primário caso este apresente falhas. A Figura 6.5 apresenta o utilização da infra-estrutura *FTWeb* com abordagem replicação passiva.

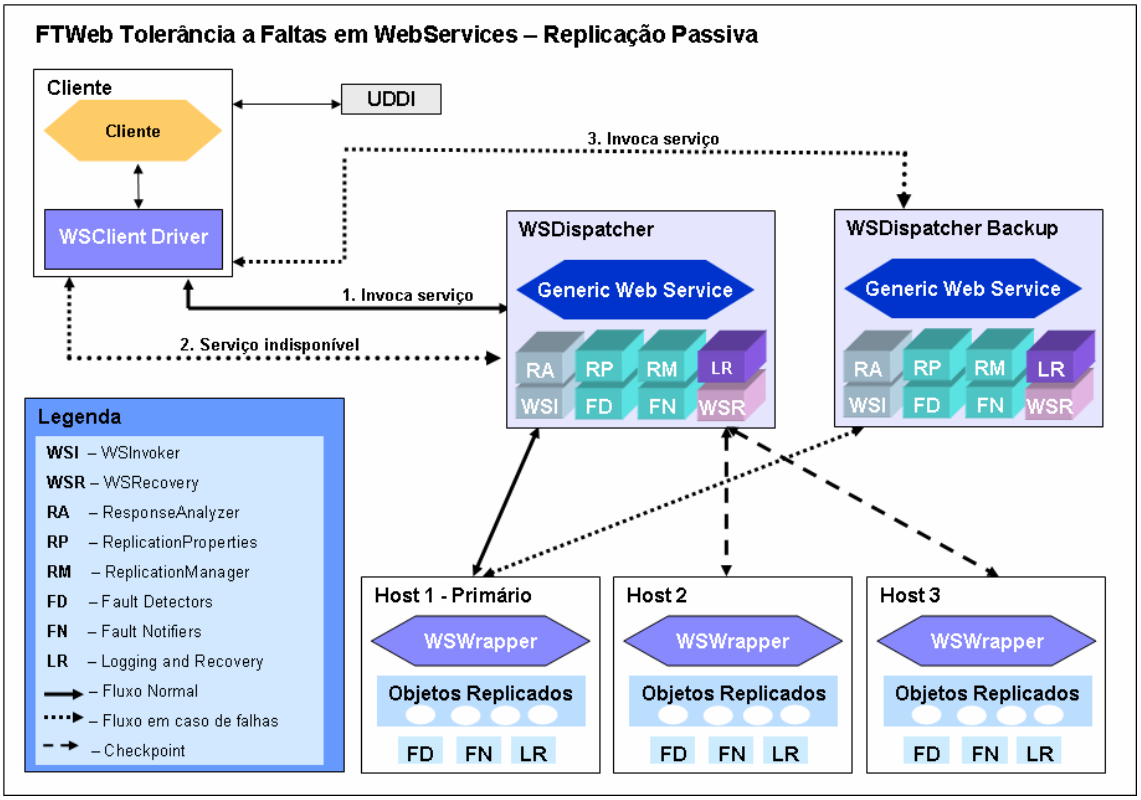

Figura 6.5 – Replicação Passiva com a infra-estrutura FTWeb

Através da infra-estrutura *FTWeb* o cliente envia a requisição sem conhecer o membro primário e a infra-estrutura, através do componente *WSinvoker,* redireciona a sua requisição. A adição de um mecanismo de *log* (LR – *Logging and Recovery*), no módulo *WSDispatcher,* permite que o estado do primário seja persistido e enviado às réplicas *backups* para garantir a consistência do estado dos membros *(checkpoint)*.

O componente *WSRecovery* possui duas importantes funções na replicação passiva: realiza a atualização dos estados dos *backups* e recupera o estado primário e dos *backups* quando estes apresentarem falhas. Para transferir o estado do primário para os *backups*, o *WSRecovery* interage com o mecanismo de *log* (LR) para obter as requisições e os estados persistidos.

Os mecanismos que realizam a transferência do estado do primário para os *backups* podem seguir duas diferentes abordagens: replicação passiva fria e replicação passiva quente. Na replicação passiva quente o mecanismo de *checkpoint* é realizado em intervalos pré-definidos (apresentado como *T* na Figura 6.6) e garante que apenas as requisições realizadas após o último *checkpoint* serão enviadas aos *backups*. Após o *checkpoint* as requisições são descartadas. Em caso de falha do primário, o novo primário eleito assume a execução das operações a partir do *checkpoint* mais recente. A Figura 6.6 representa o mecanismo de *checkpoint.*

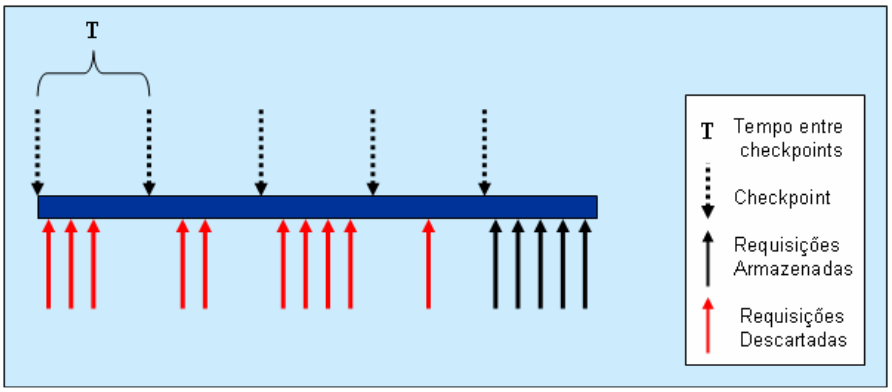

Figura 6.6 – Mecanismo de *checkpoint*

Na replicação passiva fria o mecanismo de *log* contido no *WSDispatcher Engine* também armazena as requisições, entretanto o *WSRecovery* apenas envia as requisições para os *backups* em caso de falhas do primário. Pode-se concluir que o mecanismo de recuperação da abordagem passiva fria leva mais tempo que o mecanismo implementado na abordagem passiva quente.

O sistema de configuração deve permite ao administrador do serviço definir uma réplica como primária e estabelecer os intervalos de *checkpoint*. Em situação de falha do primário é necessário um protocolo que defina entre os *backups* o novo primário. Neste caso o *WSDispatcher Engine* detecta a falha do primário, elege um novo primário e redireciona a requisição. Diferentemente dos outros modelos que propõem a técnica de replicação passiva como [Deron et. al., 2003] e [Zhang et. al., 2004], utilizando a infra-estrutura *FTWeb* não é necessário o *rebind* do cliente.

Na abordagem de replicação passiva necessariamente os componentes *WSDispatcher Engine e WSDipatcher Engine Backup* devem estar presentes na mesma rede porque o registro do estado dos serviços deve ser compartilhado pelos dois componentes. As funções de seqüenciador do componente *WSInvoker* não são utilizados nesta abordagem, uma vez que não é necessário garantir o determinismo entre as réplicas.

6.5. Replicação Semi-Ativa

Na replicação semi-ativa todas as réplicas livres de falhas recebem e executam a requisição, entretanto apenas a réplica líder é responsável pelo retorno das mensagens ao cliente. Esta abordagem, utilizando a infra-estrutura *FTWeb,* pode ser implementada da seguinte forma: a requisição é submetida pelo cliente e o *WSDispatcher Engine* redireciona a execução para todas as réplicas, no entanto aguarda apenas a resposta da réplica líder (Figura 6.7). Neste modelo não é necessário mecanismo de log no *WSDispatcher Engine,* assim como na abordagem replicação ativa. Desta forma o *WSDispatcher Engine* e o *WSDispatcher Engine Backup* podem estar localizados em redes distintas.

Em caso de falhas na réplica líder, o mesmo protocolo utilizado na replicação passiva para escolha do primário, é utilizado neste modelo para definir o novo líder. Os mecanismos de recuperação, detecção de falhas e gerenciamento de réplicas são utilizados da mesma forma que na replicação ativa. Assim como na replicação passiva o sistema de configuração deve possuir na sua interface um mecanismo para a definição da réplica líder. As funções de seqüenciador do componente *WSInvoker* são necessárias nesta abordagem a fim de garantir o determinismo entre as réplicas. O componente Response Analizer não é utilizado neste modelo.

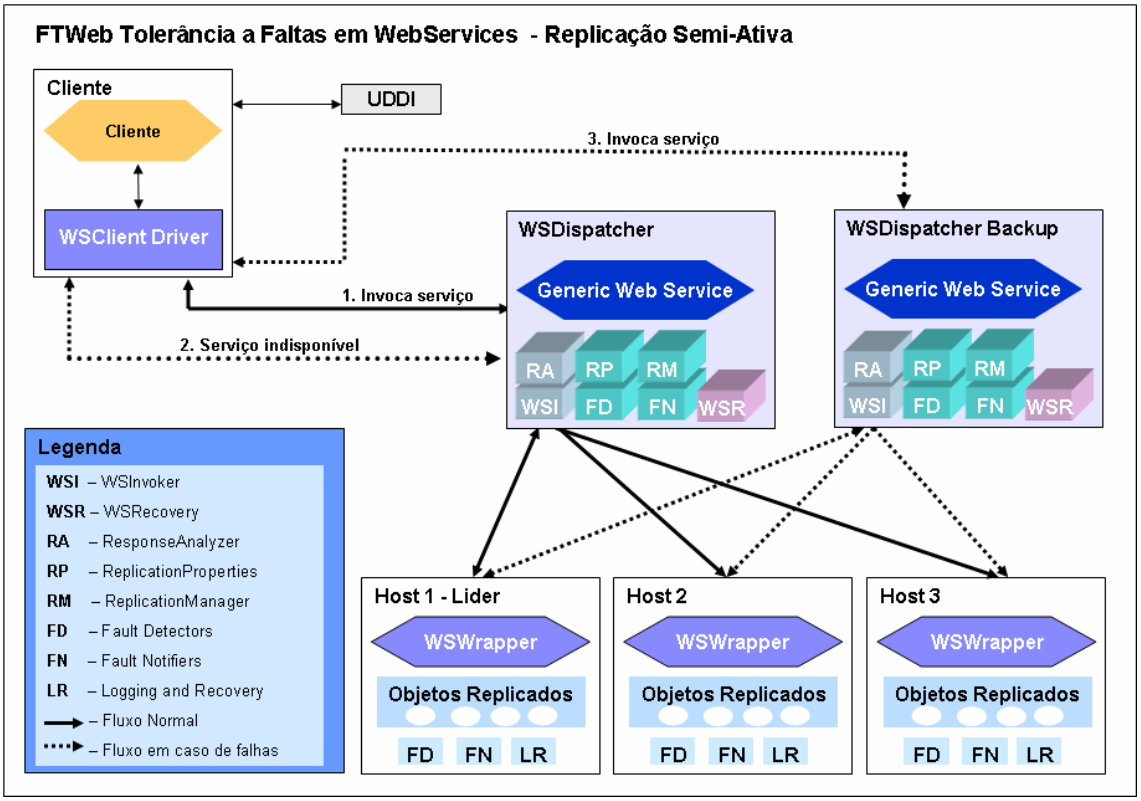

Figura 6.7 – Replicação semi-ativa com a infra-estrutura FTWeb

6.6. Conclusão

Este capítulo descreveu uma proposta de infra-estrutura para alcançar tolerância a faltas em arquiteturas orientadas a serviço. Esta infra-estrutura é denominada de *FTWeb* e tem como base fundamental a utilização da técnica de replicação ativa. No entanto apresenta uma arquitetura flexível o suficiente para que sejam empregadas outras abordagens de replicação.

Através desta abordagem é possível replicar os objetos em servidores dispersos geograficamente (em diferentes domínios) e delegar a sua administração à infra-estrutura *FTWeb* através de uma interface bem definida.O *FTWeb* não afeta a operabilidade de serviços já existentes, podendo coexistir no mesmo ambiente, serviços executados sob o *FTWeb* e serviços web tradicionais. O *FTWeb* pode ser utilizado em diferentes tipos de serviço web: *statefull*, *stateless*, síncronos e assíncronos.

No próximo capítulo são descritos os detalhes de implementação do protótipo e as experiências realizadas para validar o seu funcionamento e avaliar o seu desempenho.

Capítulo 7: Implementação e Avaliação de Desempenho

7.1. Introdução

O suporte de tolerância a faltas para aplicações desenvolvidas em arquiteturas orientadas a serviços através da infra-estrutura *FTWeb* é especificado no Capítulo 6. Esta infra-estrutura define um conjunto de componentes para a implementação de técnicas de replicação em ambientes distribuídos e heterogêneos. No decorrer deste capítulo serão apresentados os detalhes de implementação e os resultados obtidos com a avaliação de desempenho desta infra-estrutura.

7.2. Implementação da Infra-estrutura

A implementação da infra-estrutura *FTWeb* foi realizada utilizando a linguagem Java JDK 1.4.2 [J2SE, 2003], GroupPac 1.4 [Cheuk et. al. 2001], JacORB 1.4.1 [JacOrb 2002] e AXIS 1.2 [AXIS, 2004] IBM Web Services Toolkit 5.1.2 [WSAD, 2003]. O servidor de aplicações utilizado foi IBM WebSphere Application Server 5.1 e todas as APIs utilizadas neste ambiente estão em conformidade com os padrões de interoperabilidade definidos pelo *WS-I Basic Profile 1.0* [WS-I, 2004].

7.2.1 WSClient Driver

O *WSClient Driver* tem como objetivo evitar que o *WSDispatcher Engine* seja um ponto crítico de falha. Este componente deve tolerar as faltas apresentadas pelo servidor de aplicações onde está hospedado o *WSDispatcher Engine*. Este componente pode ser implementado através de duas diferentes abordagens através da implementação de um interceptador na camada SOAP ou através da alteração dos componentes responsáveis pela invocação do serviço.

Os interceptadores na arquitetura SOAP são denominados de *Handlers* e são componentes que atuam entre as invocações dos serviços e podem executar desde tarefas simples como o registro de logs até tarefas complexas como a implementação de mecanismos de autenticação e autorização. Os *Handlers* podem atuar no ambiente do consumidor ou no ambiente do provedor do serviço, como apresentado na Figura 7.1, e podem ter o seu escopo reduzido a um único serviço ou atuar sobre todos os serviços. Os *Handlers* podem ser agrupados e quando agrupados são denoninados de *Chain.* Um

Chain é um *Handler* especial que define a seqüência de execução dos *Handlers* que o compõe.

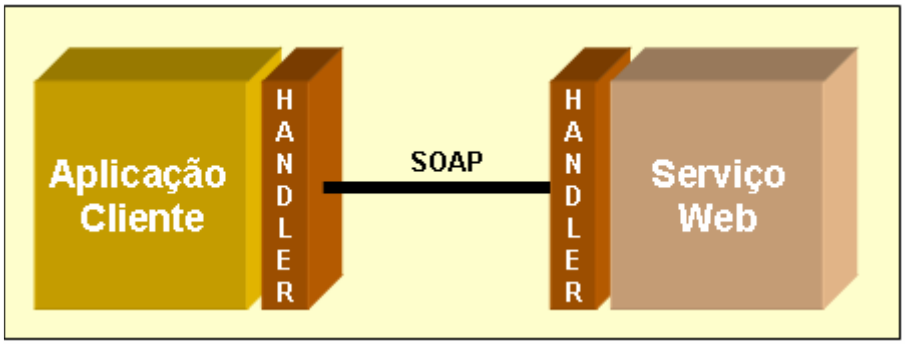

Figura 7.1 – Utilização dos *Handlers*

Como um interceptador na camada SOAP, o *WSClientDriver* necessariamente deve estender as funcionalidades da classe org.apache.axis.handlers.BasicHandler conforme apresentado na Figura 7.2 e implementar o redirecionamento das mensagens para o componente *WSDispatcher Engine Bakup* através do método *onFault.*

```
1.    public class RedirectHandler extends BasicHandler
2.    {
3.          public  void  invoke(MessageContext  msgContext)  throws
            AxisFault
4.          {
5.          }
6.          public void onFault(MessageContext context)
7.          {
8.          }
9.    }
```

Figura 7.2 – Estensão da classe *Basic Hanlder*

A implementação do *WSClientDriver* através de alterações nos componentes que invocam o serviço deve prever a detecção da falha e o redirecionamento da requisição para o servidor redundante. Esta abordagem não é transparente para a aplicação cliente e necessita que o algoritmo de redirecionamento seja implementado junto a invocação do serviço. A Figura 7.3 apresenta o fragmento do código desta abordagem.

```
1.    Client c=new Client();
2.    try
3.    {
4.          c.invoke(urlEndPoint,serviceName,method);
5.    }
6.    catch (Exception e)
7.    {
8.          if(ExceptionManager.contains(e))
9.            c.invoke(urlEndPointBackup,serviceName,method);
10.   }


11. public Object invoke(String urlEndPoint, String serviceName,
String
    method) throws Exception
12. {
```

```
13.   Service  service = new Service();
14.   Call     call    = (Call) service.createCall();
15.   call.setTargetEndpointAddress( new java.net.URL(urlEndPoint));
16.   call.setOperationName(new QName(serviceName, method) );
17.   Object res = call.invoke( new Object[] {} );
18.   return res;
19. }
```

Figura 7.3 – Fragementos do código *WSClient Driver*

A Figura 7.3 apresenta a invocação de um serviço que contém um método sem parâmetros e este método retorna uma resposta. O método *invoke* do objeto *Client* (linha 4) realiza a invocação deste serviço através da URL, do nome do serviço e do método. Estas informações são passadas como parâmetros. A implementação do método *invoke* consiste na criação do objeto *Call* (linha 145) da API SOAP e na configuração necessária para a execução do serviço através dos métodos *setTargetEndpointAddress* (linha 15) e *setOperationName* (linha 16) que definem a localização, o serviço e a operação a ser executada. Caso haja uma exceção ao invocar o serviço, esta exceção é obtida através do bloco *catch* (linha 6). A classe *ExceptionManager* (linha 8) faz parte do *WSClient Driver* e é responsável por verificar se a exceção retornada deve ser ignorada e, consequentemente, se a execução do serviço deve ser redirecionada para o *WSDispatcher Engine Backup*.O *WSClient Driver* deve apenas redirecionar as requisições quando a exceção estiver relacionada ao servidor de aplicações. Exceções do serviço não são redirecionadas pelo *WSClient Driver*.

7.2.2 WSDispatcher Engine

Os componentes que formam o módulo central *WSDispatcher Engine,* podem ser classificados segundo a forma em que são implementados como:

- Componentes implementados na forma de uma aplicação J2EE e instalados no servidor de aplicações tais como o *Generic Web Service*, o *WSInvoker* e o *Response Analyzer*.

- Objetos CORBA implementados sob a especificação *FT-Corba*. *ReplicationManager, ReplicationProperties,* e *Fault Detector.*

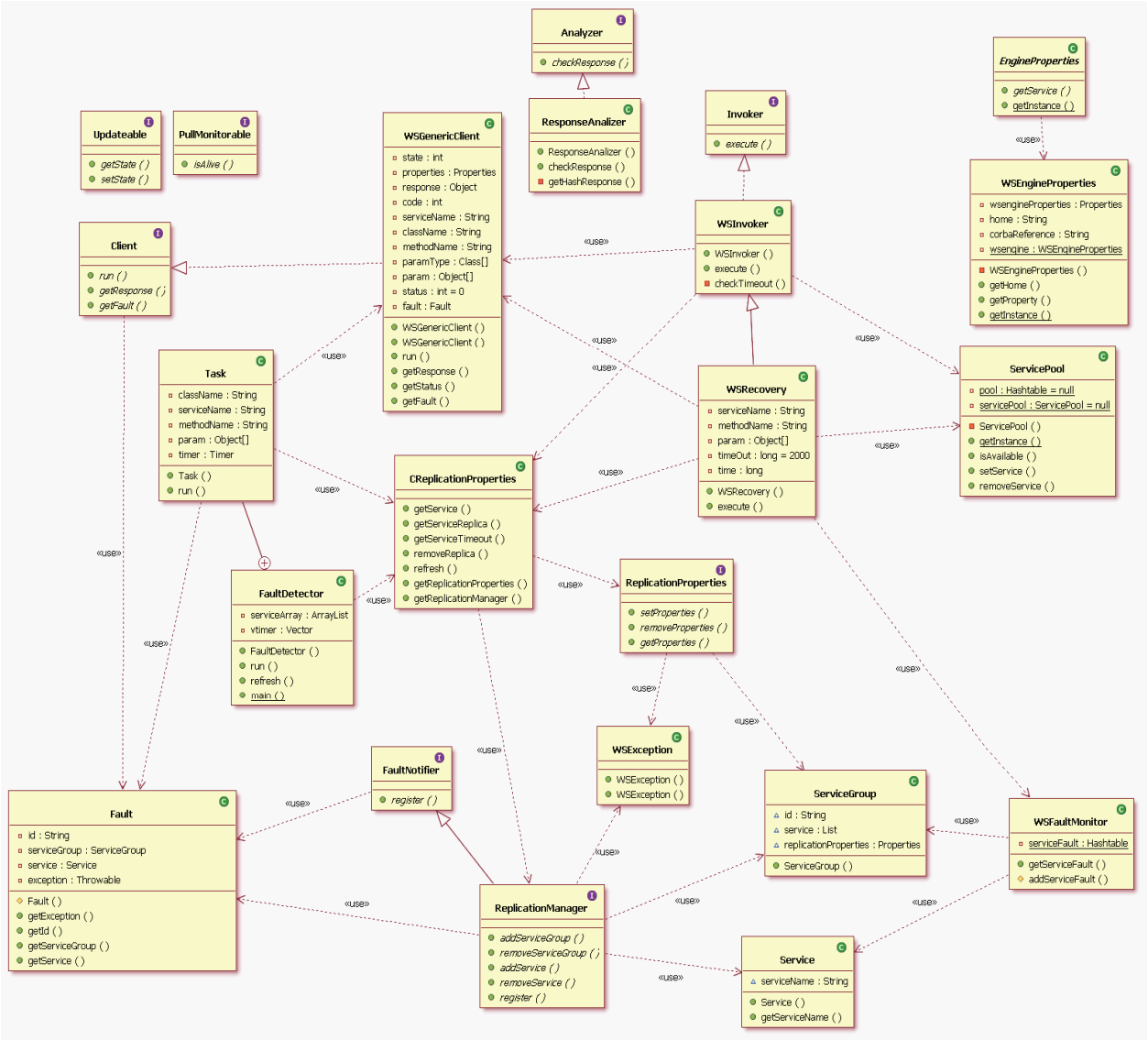

Figura 7.4. Diagrama de classes *WSDispatcher Engine*

O diagrama de classes contendo os principais componentes do *WSDispatcher* pode ser visto na Figura 7.4. Em uma visão geral a relação entre esses objetos, é descrita abaixo:

- A classe *WSEngineProperties* define as implementações que serão utilizadas dos componentes *WSGenericClient*, *WSInvoker* e *ResponseAnalizer* estas classes possuem interfaces definidas permitindo que as suas funcionalidades sejam reimplementadas e adaptadas conforme os requisitos dos serviços nos provedores.

- A classe *ServicePool* é utilizado pelo *WSInvoker* para implementar as funcionalidades do seqüenciador de requisições.

- A classe *CReplicationProperties* é utilizada pelo *WSInvoker* e *WSRecovery* e tem a responsabilidade de obter as informações gerenciadas pelas classes *ReplicationManager* e *ReplicationProperties*. Desta forma os objetos implementados sob a tecnologia CORBA são abstraídos dos componentes que formam a aplicação J2EE.

- A classe *WSRecovery* estende as funcionalidades da classe *WSInvoker* e reimplementa o método *execute* a fim de realizar a recuperação do estado de réplicas faltosas. Esta classe utiliza a classe *FaultMonitor* para obter a relação das réplicas que apresentaram falhas.

- A classe *WSException* é utilizada pelas classes *ReplicationProperties* e *ReplicationManager* para o lançamento de exceções em suas operações

- A classe *EngineException* é utilizada pelas classes *WSInvoker, ResponseAnalyser, WSGenericClient* para o lançamento de exceções em suas operações.

- A classe *WSGenericClient* é responsável por concretizar a invoção dos serviços nos provedores e é utilizado pelo *WSInvoker, FaultDetector* e *WSRecovery.*

- As classes *ServicePool* e *Service* representam respectivamente o grupo de serviços e os serviços nos provedores.

- A classe *Task* é utilizada para a programação de eventos que ocorrem com periodicidade. Esta classe é utilizada pelo *FaultDetector* nos mecanismos de detecção de faltas.

- As interfaces *PullMonitorable* e *Updateable* devem ser implementadas pelos serviços para que seja possível realizar a monitoração e a recuperação do seu estado.

As funcionalidades providas pelos componentes que formam *WSDispatcher* e os detalhes de sua implementação são descritos nos próximos itens.

7.2.2.1 Generic Web Service

O componente *Generic Web Service* realiza a interface com o cliente e recebe como parâmetros: a identificação do grupo do serviço web, o método a ser executado e os parâmetros para a sua execução. A Figura 7.5 apresenta o documento WSDL deste componente:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://wsproxy.service.edu"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:intf="http://wsproxy.service.edu"
xmlns:impl="http://wsproxy.service.edu">
  <wsdl:types>
    <schema elementFormDefault="qualified" targetNamespace="http://wsproxy.service.edu"
xmlns="http://www.w3.org/2001/XMLSchema" xmlns:impl="http://wsproxy.service.edu"
xmlns:intf="http://wsproxy.service.edu" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <element name="execute">
    <complexType>
     <sequence>
      <element name="serviceName" nillable="true" type="xsd:string"/>
      <element name="method" nillable="true" type="xsd:string"/>
      <element maxOccurs="unbounded" name="param" type="xsd:byte"/>
     </sequence>
    </complexType>
   </element>
   <element name="executeResponse">
```

```
   <complexType>
    <sequence>
     <element name="executeReturn" nillable="true" type="xsd:anyType"/>
    </sequence>
   </complexType>
  </element>
 </schema>
 </wsdl:types>
 <wsdl:message name="executeResponse">
   <wsdl:part name="parameters" element="intf:executeResponse"/>
 </wsdl:message>
 <wsdl:message name="executeRequest">
   <wsdl:part name="parameters" element="intf:execute"/>
 </wsdl:message>
 <wsdl:portType name="WSProxyEngine">
   <wsdl:operation name="execute">
     <wsdl:input name="executeRequest" message="intf:executeRequest"/>
     <wsdl:output name="executeResponse" message="intf:executeResponse"/>
   </wsdl:operation>
 </wsdl:portType>
 <wsdl:binding name="WSProxyEngineSoapBinding" type="intf:WSProxyEngine">
   <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
   <wsdl:operation name="execute">
     <wsdlsoap:operation soapAction=""/>
     <wsdl:input name="executeRequest">
       <wsdlsoap:body use="literal"/>
     </wsdl:input>
     <wsdl:output name="executeResponse">
       <wsdlsoap:body use="literal"/>
     </wsdl:output>
   </wsdl:operation>
 </wsdl:binding>
 <wsdl:service name="WSProxyEngineService">
   <wsdl:port name="WSProxyEngine" binding="intf:WSProxyEngineSoapBinding">
     <wsdlsoap:address
location="http://localhost:9080/WebServiceServerProject/services/WSProxyEngine"/>
   </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

Figura 7.5. WSDL do componente *Generic WebService*

7.2.2.2 WSInvoker

Após o componente *Generic WebService* receber os parâmetros necessários para a invocação do serviço estes parâmetros são repassados ao componente *WSInvoker*. A Figura 7.6 apresenta fragmentos do código do componente *WSInvoker,* as linhas estão numeradas e são referenciadas durante a explanação das funcionalidades do código.

A classe *CReplicationManager* (linha 1) obtém dos componentes *ReplicationManager* e *ReplicationProperties* a localização e as propriedades de replicação através do método *getServiceReplica* (linha 2). O método *isAvailable* da classe *ServicePool* (linha 3), verifica a disponibilidade do serviço e aguarda, caso o serviço esteja sendo executado. Após a liberação o *WSInvoker* aloca o serviço através do método *setService* (linha 7). Através da classe *ServicePool,* o componente *WSInvoker* difunde a requisição com ordenação FIFO, garantindo o determinismo entre as réplicas. A classe *WSGenericClient* (linha 12) é responsável pela invocação do

serviço nas réplicas. Após o termino da execução a referência do serviço é liberada pelo método *removeService* (linha 16) da classe *ServicePool,* desta forma o serviço torna-se disponível para ser executado por outros clientes.

A infra-estrutura *FTWeb* permite que um grupo de serviços web replicados utilizem um componente votador para tolerar faltas de valor, neste caso após a execução do serviço nas réplicas as respostas são submetidas ao *ResponseAnalizer* (linha 22), a resposta indicada por este componente será retornada ao *Generic Web Service,* para ser retornada ao cliente.

```
1. CReplicationManager replicationManager= new CReplicationManager();
2. Properties serviceProperties=replicationManager.getServiceReplica(serviceName);
3.  if(serviceProperties.isEmpty())
          throw new EngineException(" Servico nao encontrado " + serviceName);
4.  ThreadGroup tg = new ThreadGroup(serviceName);
5.  ServicePool servicePool = ServicePool.getInstance();
6.  servicePool.isAvailable(serviceName);       // verifica se o serviço esta disponível
7.  servicePool.setService(serviceName,tg);   //  aloca o serviço para o cliente
8.  Enumeration en= serviceProperties.elements();
9.  List service = new ArrayList();
10. for(int i=0;en.hasMoreElements();i++)
    {
11.       String serviceReplica= (String)en.nextElement();
12.       WSGenericClient eng= new WSGenericClient(serviceReplica,this.methodName,this.param);
13.       service.add(eng);
14.       Thread t = new Thread(tg,eng,serviceName+i);
15.       t.start();
    }
...
...
16.  servicePool.removeService(serviceName);  // torna o serviço disponível após a execução
17.  List responseList=new ArrayList();
18.  for(int i=0;i<service.size();i++)
     {
19.       WSGenericClient ws =(WSGenericClient)service.get(i);
20.        responseList.add(ws.getResponse());
     }
....
21.  ResponseAnalizer votador =new ResponseAnalizer();
22.  Object  response =votador.checkResponse(responseList);
23.   return response;
```

Figura 7.6. Fragmentos do código do *WSInvoker*

Como pode ser observado na Figura 7.6 o *WSInvoker* invoca o serviços web através da classe *WSGenericClient.* Os componentes *ResponseAnalizer* e *WSGenericClient* são implementados sob o conceito de *driver*, ou seja, permitem que estes componentes sejam adaptados conforme o requisito do ambiente de execução, desde que sejam implementados obedecendo as diretrizes de uma determinada interface. Esta abordagem permite que o *WSGenericClient* utilize diferentes formas e APIs para a invocação dos serviços como a Web Service Invocation Framework [WSIF, 2003] e Apache AXIS [AXIS, 2004].

A invocação do serviço pode ser realizada de duas diferentes formas dinâmica também conhecida como *stubless* através do documento WSDL e através da geração de classes (*stubs*) para acesso o serviço. Caso a abordagem escolhida seja a geração de

stubs é necessária uma mudança na console de configuração (7.10) e no lugar de documentos WSDL devem ser informadas as classes que implemetam a invocação do serviço. Nesta abordagem o *WSGeneric Client* utilizará reflexão [Reflection 1996] para acessar o serviço. A Figura 7.7 abaixo apresenta os fragmentos do código utilizado para invocar serviços. A invocação dinâmica é apresentada nas linhas 1 a 14 e utiliza as classes da API AXIS para a invocação do serviço. A invocação estática é apresentada nas linhas 15 a 20 e utilizam as classes de reflexão da linguagem Java para invocar o *stub,* gerado a partir do documento WSDL, que contém as chamadas do serviço.

```
1.    //Invocação dinâmica
2.    Service service = selectService(serviceNS, serviceName);
3.    Operation operation = null;
4.    org.apache.axis.client.Service dpf = new
5.    org.apache.axis.client.Service(wsdlParser, service.getQName());
6.    Port port = selectPort(service.getPorts(), portName);
7.    Binding binding = port.getBinding();
8.    Call call = dpf.createCall(QName.valueOf(portName),
                                 QName.valueOf(operationName));
10.   ...
11.   //input: vetor contendo parametros de entrada
12.   Object ret = call.invoke(input.toArray());
13.   //obtém parâmetro de retorno
14.   Map output = call.getOutputParams();

15.   // Invocação estática utilizando reflexão
16.   Class classDefinition = Class.forName(className);
17.   Object object = classDefinition.newInstance();
18.   Class[] parameterType = FormatParam.getParamType(param);
19.   Method method =
      classDefinition.getMethod(methodName,parameterType);
20.   response = method.invoke(object, param);
```

Figura 7.7. Fragmentos do código do *WSGeneric Client*

7.2.2.3 Response Analizer

O componente *Response Analizer* é opcional e atua como um votador, retornando a resposta mais coincidente. Conforme mencionado anteriormente este componente é implementado utilizando o conceito de *driver* e pode ter a sua implementação substituída ou adaptada conforme os requisitos da aplicação. A interface do componente *Response Analizer* pode ser observada na Figura 7.8**.** A implementação do método *checkResponse* (linha 3) em caso de problemas na verificação das respostas, deve lançar a exceção *EngineException,*esta classe de exceção é fornecida pela infra-estrutura *FTWeb*.

```
1.    public Interface ResponseAnalizer
2.    {
3.          public Object checkResponse(List responseList)throws
            EngineException;
4     }
```

Figura 7.8. Interface do componente Response Analizer

7.2.2.4 Replication Manager

O componente *ReplicationManager* é um objeto implementado sob as especificações de tolerância a faltas do CORBA (Grouppac), as interfaces deste objeto pode ser visualizada na Figura 7.9**.**

```
1.  public interface ReplicationManager
2. {
3.    void addServiceGroup(ServiceGroup serviceGroup);
4.    void removeServiceGroup(ServiceGroup serviceGroup);
5.    void addService(ServiceGroup serviceGroup, Service service);
6.    void removeService(ServiceGroup serviceGroup, Service service);
7.    void registerNotify(FaultNotifier fault);
8.   ServiceGroup getServiceGroup(String serviceName) ;
9.}
```

Figura 7.9. Interface do componente Replication Manager

O método *addServiceGroup* (linha 3) é invocado na criação de novo grupos de serviço e recebe como parâmetro o objeto *ServiceGroup*. O método *addService* (linha 5) adiciona novas réplicas de serviço em grupos já estabelecidos. O método *removeServiceGroup* (linha 4) remove o grupo de serviços. O método *removeService* retira do grupo de serviço réplicas faltosas e réplicas retiradas da configuração do grupo pelo administrador do serviço. Os métodos *addServiceGroup* e *removeServiceGroup* são invocados exclusivamente pelo o sistema de configuração após a notificação de inclusão ou remoção de um grupo de serviço. Os métodos *addService* e *removeService* são invocados pelo sistema de configuração, *Fault Detector, WSInvoker* após a notificação de alterações no grupo de serviço. O método *registerNotify* registra as notificações de falhas nas réplicas do serviço. O método *getServiceGroup* retorna o objeto *ServiceGroup* que representa o grupo do serviço, através deste componente é possível descobrir a localização das réplicas (documento WSDL).

7.2.2.5 Replication Properties

O componente Replication Properties contém as propriedades de replicação dos grupos A interface do componente Replication Properties pode ser observada na Figura 7.10:

```
1. public interface ReplicationProperties
2. {
3.    void setProperties(ServiceGroup serviceGroup, Properties props);
4.    void removeProperties(ServiceGroup serviceGroup);
5.    Properties getProperties(ServiceGroup serviceGroup);
6. }
```

Figura 7.10. Interface do componente Replication Manager

O método *setProperties* (linha 3) recebe como parâmetro o identificador do grupo de serviço e permite a configuração das propriedades de replicação. O método *removeProperties* (linha 4) remove as propriedades e o método *getProperties* (linha 5) retorna as propriedades estabelecidas para um determinado grupo de serviços. Todas as propriedades atribuídas pelo componente Replication Properties são definidas em um sistema de configuração, representado pela Figura 7.11.

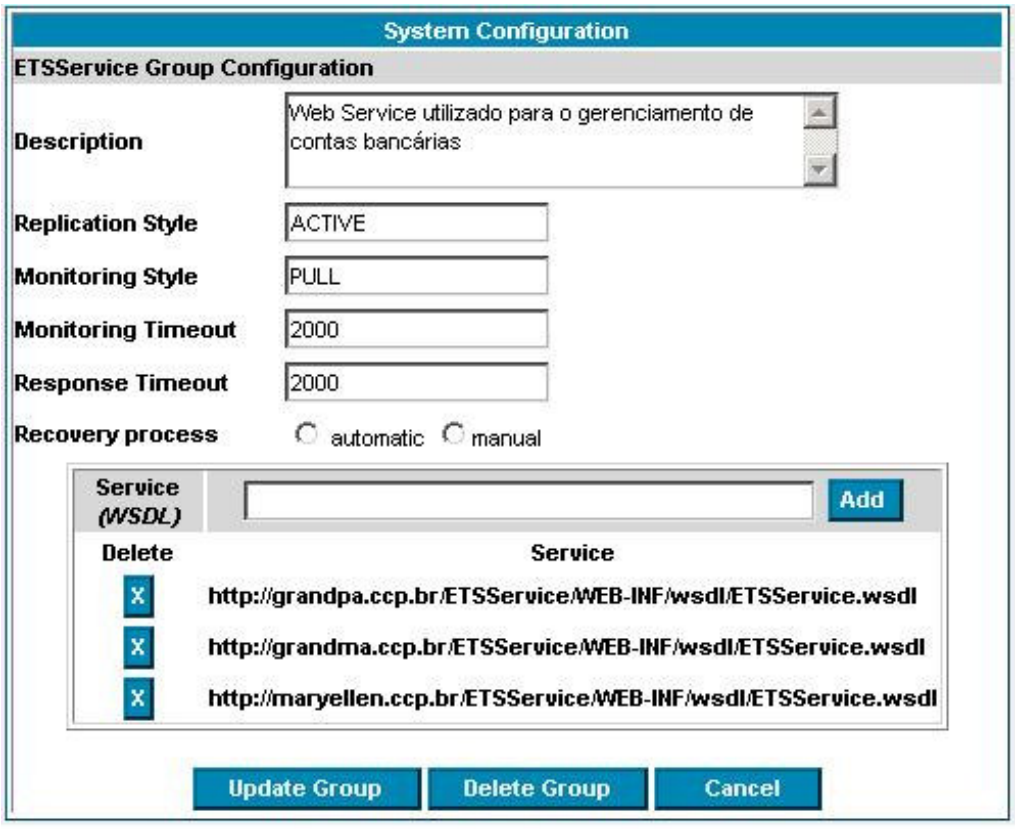

Figura 7.11. Sistema de Configuração

Este sistema permite ao administrador do grupo de serviço definir além das propriedades de replicação, a localização das réplicas através dos documentos WSDL O sistema de configuração foi desenvolvido em *Java Server Pages 1.2* e *Java Servlet 2.3*

permitindo que os administradores possam configurar os serviços remotamente fazendo o uso apenas de um navegador *web*.

7.2.2.6 Fault Detector e Fault Notifier

Estes componentes realizam a detecção e notificação de falhas. Para realizar o processo de monitoração os serviços web devem implementar a interface *PullMonitorable*. Esta interface pode ser visualizada na Figura 7.12:

```
1. public interface PullMonitorable
2. {
3.              booalen isAlive();
4. }
```

Figura 7.12. Interface de Monitoração

Através da invocação do método *isAlive()* (linha 3) o componente *Fault Detector* monitora as réplicas. A monitoração é realizada conforme as propriedades obtidas através do *Replication Properties* e definidas no sistema de configuração. *O componente Fault Detector* estende as funcionalidades da classe java.util.TimerTask permitindo *Task* de monitoração seja programada no intervalo definido nas propriedades de replicação. A forma de invocação do serviço é realizada pela classe *WSGenericClient* definida no item 7.2.2.

7.2.2.7 WSRecovery

Este componente é responsável pela recuperação do estado de réplicas faltosas. O processo de recuperação pode ser definido como manual ou automático no sistema de configuração. No processo manual o *WSRecovery* é invocado a partir da console de monitoração. No processo automático periodicamente o *WSRecovery* verifica as réplicas faltosas e tenta re-estabelecer o seu estado.

A console de monitoração é apresentada na Figura 7.13. Através do link associado a figura  na console de monitoração o administrador obtém detalhes da falha ocorrida em determinada réplica do serviço. Através do link associado a figura  o administrador pode iniciar o processo de recuperação da falta no grupo de serviço configurado com a propriedade recuperação manual. A console de monitoração foi desenvolvido em *Java Server Pages 1.2* e *Java Servlet 2.3* permitindo que os administradores possam monitorar os serviços remotamente fazendo o uso apenas de um navegador *web*.

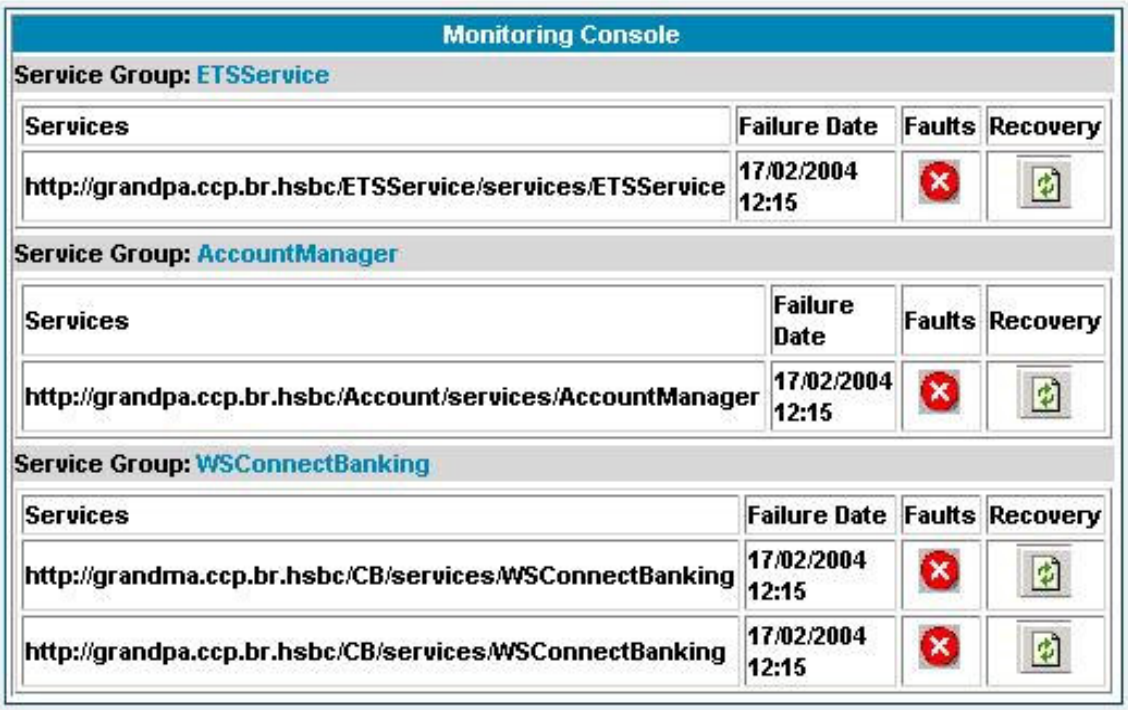

Figura 7.13. Console de Monitoração.

Para realizar o processo de recuperação os serviços web devem implementar a interfaces *Updateable*. Esta interface pode ser visualizada na Figura 7.14. O método *setState* permite que o processo de recuperação defina o novo estado da réplica faltosa. O método *getState* é utilizado para descobrir o estado das réplicas que não apresentaram falhas.

```
1. public interface Updateable
2. {
3.          void setState(State state);
5.          State getState();
6. }
```

Figura 7.14. Interface de Recuperação de estado.

7.2.3 WSWrapper

O *WSWrapper* foi desenvolvido para permitir a exposição de objetos implementados sob a arquitetura CORBA, seu funcionamento é similar ao componente *Generic Web Service* e obtém a referência do objeto, o método a ser executado e os parâmetros necessários para a sua execução. A Figura 7.15 apresenta o documento WSDL deste componente.

```
<wsdl:definitions targetNamespace="http://wrapper.corba.services.edu"
xmlns:impl="http://wrapper.corba.services.edu"
xmlns:intf="http://wrapper.corba.services.edu"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <wsdl:types>
```

```
  <schema elementFormDefault="qualified"
targetNamespace="http://wrapper.corba.services.edu"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:impl="http://wrapper.corba.services.edu"
xmlns:intf="http://wrapper.corba.services.edu"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <element name="execute">
    <complexType>
     <sequence>
      <element name="serviceName" nillable="true" type="xsd:string"/>
      <element name="methodName" nillable="true" type="xsd:string"/>
      <element maxOccurs="unbounded" name="param" type="xsd:byte"/>
     </sequence>
    </complexType>
   </element>
   <element name="executeResponse">
    <complexType>
     <sequence>
      <element name="executeReturn" nillable="true"
type="xsd:anyType"/>
     </sequence>
    </complexType>
   </element>
  </schema>
 </wsdl:types>
<wsdl:message name="executeResponse">
     <wsdl:part element="intf:executeResponse" name="parameters"/>
   </wsdl:message>
   <wsdl:message name="executeRequest">
      <wsdl:part element="intf:execute" name="parameters"/>
   </wsdl:message>
   <wsdl:portType name="WSWrapper">
      <wsdl:operation name="execute">
         <wsdl:input message="intf:executeRequest"
name="executeRequest"/>
         <wsdl:output message="intf:executeResponse"
name="executeResponse"/>
      </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding name="WSWrapperSoapBinding" type="intf:WSWrapper">
      <wsdlsoap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
      <wsdl:operation name="execute">
         <wsdlsoap:operation soapAction=""/>
         <wsdl:input name="executeRequest">
            <wsdlsoap:body use="literal"/>
         </wsdl:input>
         <wsdl:output name="executeResponse">
            <wsdlsoap:body use="literal"/>
         </wsdl:output>
      </wsdl:operation>
   </wsdl:binding>
   <wsdl:service name="WSWrapperService">
      <wsdl:port binding="intf:WSWrapperSoapBinding" name="WSWrapper">
         <wsdlsoap:address
location="http://localhost:9080/WSWrapper/services/WSWrapper"/>
      </wsdl:port>
   </wsdl:service>
```

```
</wsdl:definitions>
```

Figura 7.15. WSDL do componente *WSWrapper*

O componente *WSWrapper* é um serviço web que realiza conversão de requisições SOAP em invocações a objetos CORBA. A invocação dos objetos é realizada através da interface de invocação dinâmica (DII) provida pelo CORBA. A Figura 7.16 apresenta fragmentos do código utilizado para conversão dos parâmetros e a utilização da interface de invocação dinâmica.

```
1.    org.omg.CORBA.Object o     = ns.resolve_str(serviceName);
2.    org.omg.CORBA.Request ob = obj._request(methodName);
3.    Object[] parameter = Object[])SerializableUtils.deserialize(param);
4.    //Mapeamento dos parâmetros
5.    for(int i=0;i<parameter.length;i++)
6.    {
7.        if(parameter[i] instanceof Integer)
8.        {
9.                ob.add_in_arg().insert_long(((Integer)parameter[0]).intValue());
10.       }
11.       else if(parameter[i] instanceof String)
12.       {
13.               ob.add_in_arg().insert_string(((String)parameter[0]));
14.       } ...
15.   }
16.   ob.invoke();
```

Figura 7.16. Fragmentos de código da *WSWrapper*

7.3. Avaliação de Desempenho

Visando verificar o desempenho da infra-estrutura proposta foram executados testes em uma rede local de 10Mbps compostas por computadores Intel Pentium IV 2.8 GHz com 1Gb de memória RAM e sistema operacional Microsoft 2000 Professional. O *WSDipatcher Engine* foi instalado em dois servidores, sendo um servidor *backup*. As réplicas foram distribuídas em até sete computadores todos contendo o *IBM WebSphere Application Server 5.1*.

Para avaliar o *overhead* do *Fault Detector*, este componente foi instalado em um computador independente monitorando grupos de serviços web com 3, 5 e 7 réplicas. Como apresentado na Figura 7.17, para grupos compostos com 7 réplicas o percentual de utilização da CPU é aproximadamente 4% quando o intervalo de monitoração é definido em 2 segundos. A avaliação de desempenho mostrou para o ambiente de execução utilizado o intervalo de monitoração mais indicado para o *FTWeb* é de 30 segundos.

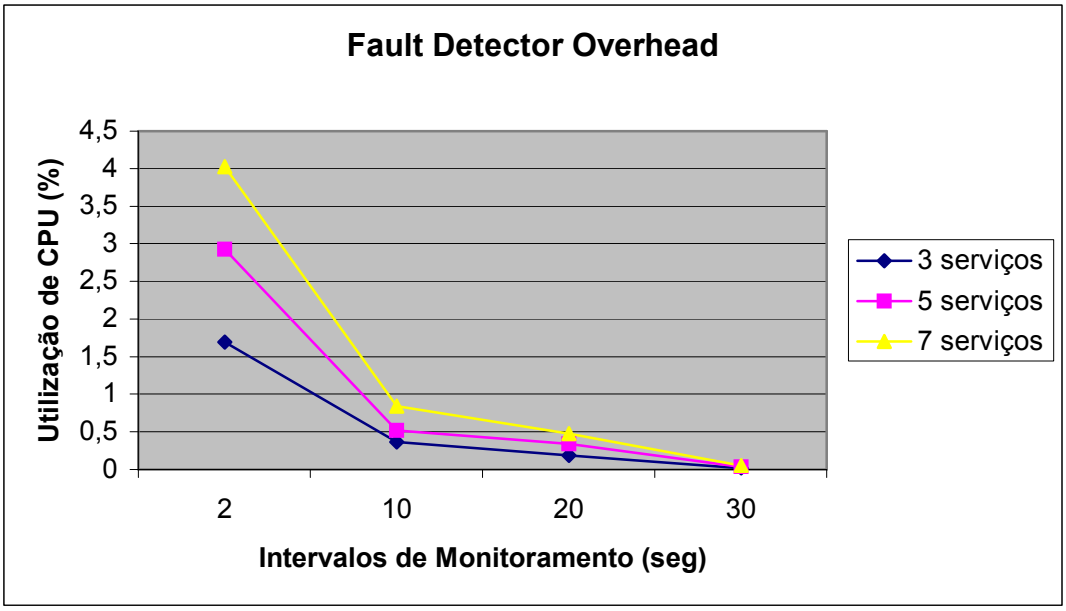

Figura 7.17 - Desempenho do Fault Detector.

Para avaliar o tempo de resposta adicionado pela infra-estrutura *FTWeb* considerando o tamanho das mensagens, foram realizados testes com grupos de serviços com até 4 réplicas e com variação no tamanho da mensagem de 1 a 256 Kbytes. Para avaliar o custo da tolerância a faltas provida pela infra-estrutura, foram realizados testes com o mesmo serviço sem a utilização do *FTWeb*.

É possível observar na Figura 7.18 que para mensagens com até 16 Kbytes a variação do número de réplicas que compõem o grupo, não afeta significativamente o tempo de resposta do serviço. Os testes apresentaram aproximadamente 25% de acréscimo em relação a um serviço executado sem o *FTWeb*. Entretanto, o acréscimo no tempo de resposta pode chegar a 60%, considerando mensagens com 256 Kbytes e 4 réplicas compondo o serviço utilizando o esquema de votação. Para os testes executados com esquema de votação, o tempo era determinado pela execução da réplica localizada no computador mais lento. Os testes executados sem votação, enviando ao cliente a primeira mensagem processada, apresentaram tempos iguais, aos testes realizados utilizando o *FTWeb* com apenas 1 serviço no grupo.

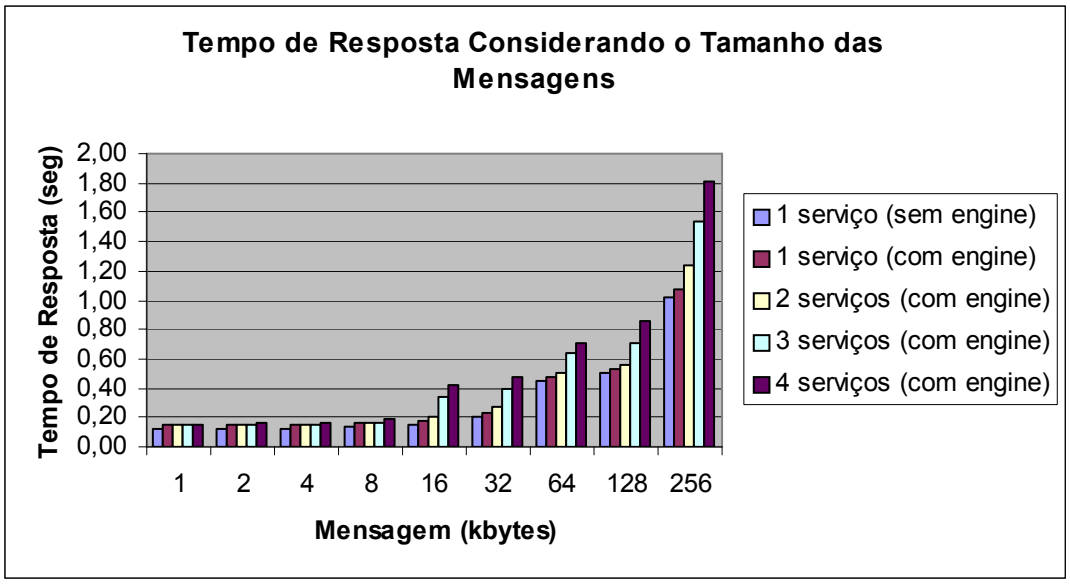

Figura 7.18. Tempo de Resposta Considerando o Tamanho da Mensagem.

Para avaliar o tempo de resposta adicionado pelo *FTWeb*, considerando o número de usuários simultâneos acessando o serviço, foram realizados testes com grupos de serviços com até 4 réplicas e com variação de 2 a 20 usuários simultâneos. O tamanho da mensagem utilizada foi 4 Kbytes.

É possível observar na Figura 7.19 que a variação do número de réplicas que compõem o grupo não afeta significativamente tempo de resposta do serviço. O tempo de resposta é afetado quando incrementado o número de usuários simultâneos devido aos mecanismos que provêm o determinismo das réplicas. Quando submetido a 20 usuários simultâneos o acréscimo no tempo de resposta foi de aproximadamente 41%.

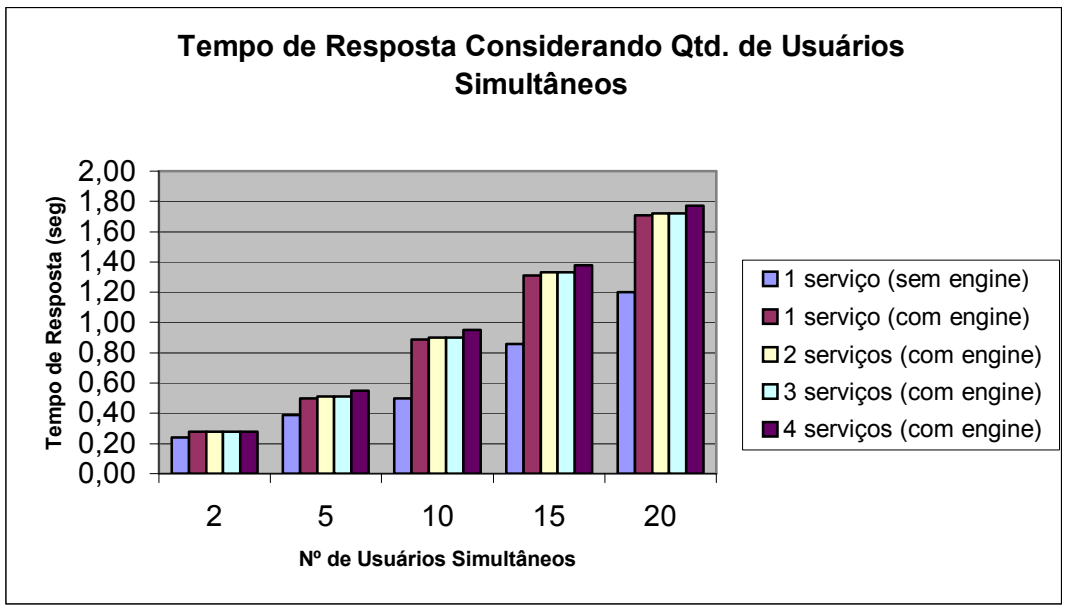

Figura 7.19. Tempo de Resposta Considerando a Qtd. De Usuários Simultâneos.

Para avaliar a disponibilidade fornecida pelo FTWeb, considerando serviços dispersos geograficamente, foram realizados testes com 2 grupos de serviços web diferentes. O primeiro grupo contém 3 serviços web públicos, localizados em diferentes países que realizam operações de conversão de temperatura. O segundo grupo contém 3 serviços web públicos localizados em diferentes países que realizam operações de conversão de moeda. É possível observar na Tabela 7.3 que quando os serviços são submetidos a 1000 execuções individualmente, cada serviço apresenta um percentual de falha. Entretanto quando estes serviços são agrupados e submetidos a 1000 execuções utilizando a infra-estrutura FTWeb o percentual de falha é 0.0%.

Serviço: Conversão de Temperatura

FTWeb	Xmethod (USA)	WebServiceX (UK)	DeveloperDays (USA)
0,0%	0,9%	1,7%	0,5%

Serviço: Conversão de Moeda

FTWeb	Xmethod (USA)	WebServiceX (UK)	DeveloperDays (USA)
0,0%	1,2%	0,5%	0,0%

Serviços Equivalentes -1000 execuções por serviço

Tabela 7.3 – Teste de Disponibilidade Considerando Serviços Dispersos Geograficamente

7.4. Conclusão

Este capítulo apresentou os detalhes de implementação e a avaliação de desempenho dos componentes que compõem a infra-estrutura *FTWeb*. O protótipo construído demonstra que a abordagem é viável e apresenta um desempenho satisfatório considerando o ganho em disponibilidade e confiabilidade propiciado pela infra-estrutura.

O objetivo principal do projeto é a utilização da técnica de replicação ativa para alcançar a tolerância a faltas em arquiteturas orientadas a serviços. Esta abordagem provê suporte para as seguintes classes de faltas: parada, omissão e valor. A infra-estrutura proposta é baseada em um mecanismo denominado de *WSDispatcher Engine* que possui componentes responsáveis por: compor grupos de serviços, detectar e recuperar faltas, invocar concorrentemente as réplicas do serviço, garantir o determinismo entre as réplicas e estabelecer um esquema de votação sobre as respostas retornadas pelos serviços.

A aplicação do modelo em serviços web já implementados e operacionais é bastante simplificada requer apenas a inserção de métodos de monitoração e recuperação do estado da réplica. O modelo ainda oferece um sistema de configuração e monitoração das réplicas que compõem o grupo de serviço e permite que administradores realizem a monitoração e a configuração remotamente fazendo o uso apenas de um navegador *web*.

Capítulo 8: Conclusão e Trabalhos Futuros

8.1. Conclusão

A arquitetura de serviços web surgiu como uma resposta à busca da interoperabilidade entre aplicações. Nos últimos anos existe um interesse crescente em executar na Internet aplicações com requisitos de alta disponibilidade e confiabilidade, contudo as tecnologias associadas a essa arquitetura ainda não oferecem suporte adequado a esses requisitos.

Um conjunto de técnicas tem sido proposta na literatura que endereça esta questão provendo alta disponibilidade em serviços web. Em geral estas técnicas são baseadas em desenvolvimento de um grupo de serviços distribuído em computadores dentro da mesma rede corporativa. A disponibilidade é alcançada através da redundância na execução do serviço. O número de requisições que o serviço pode resolver é otimizado através da distribuição de carga entre os servidores replicados. Estas técnicas são vulneráveis a falhas em roteadores, *gateways* e outros componentes que realizam interfaces com a rede.

A infra-estrutura proposta, neste trabalho, se situa neste contexto e provê uma nova camada de software que atua como um *proxy* entre as requisições do cliente e os serviços nos provedores. O objetivo principal é garantir tolerância a faltas transparente para o cliente através da técnica de replicação ativa. Através desta abordagem é possível replicar os objetos em servidores dispersos geograficamente e delegar a sua administração a infra-estrutura *FTWeb.*

Esta infra-estrutura permite composição de grupos de serviços fazendo com que o cliente perceba com um único serviço um conjunto de serviços replicados e independentes. Os componentes desta infra-estrutura são responsáveis pela detecção e recuperação de falhas, pela invocação paralela das réplicas, por garantir o determinismo entre as réplicas e estabelecer um esquema de votação sobre as respostas retornadas pelos serviços.

O *FTWeb* não afeta a operabilidade de serviços já existentes, podendo coexistir no mesmo ambiente, serviços executados sob o *FTWeb* e serviços web tradicionais. O desenvolvimento de serviços web tolerantes a faltas usando a infra-estrutura *FTWeb* é

bastante simplificada, requer apenas a inserção dos métodos de monitoração e recuperação do estado das réplicas fornecidos pela infra-estrutura. O *FTWeb* pode ainda ser utilizado em diferentes tipos de serviço web: *statefull*, *stateless*, síncronos e assíncronos.

A flexibilidade apresentada pela arquitetura do *FTWeb* permite que sejam empregadas outras abordagens de replicação como a replicação passiva e a replicação semi-ativa. O protótipo construído demonstra que a abordagem é viável e apresenta um desempenho satisfatório considerando o ganho em disponibilidade e confiabilidade propiciado pela infra-estrutura. No entanto, os algoritmos que implementam o determinismo entre as réplicas podem ser otimizados a fim de obter um melhor desempenho no tempo de resposta quando o serviço é invocado por múltiplos usuários simultaneamente.

Os componentes da infra-estrutura foram desenvolvidos prevendo baixo acoplamento e alta coesão, permitindo que estes componentes sejam adaptados conforme as necessidades da aplicação. A maior parte dos componentes foi construída sob o conceito de *drivers,* ou seja, a implementação pode ser substituída ou adaptada desde que sejam respeitadas as interfaces estabelecidas. A principal contribuição deste trabalho foi a proposição de uma infra-estrutura para tolerar faltas de valor, omissão e parada em arquiteturas orientadas a serviços. Resultados parciais deste trabalho foram publicados em [Santos et. al. 2005].

8.2. Trabalhos Futuros

Como objetivo para trabalhos futuros está análise e a implementação de um protótipo que realize a integração do modelo *FTWeb* com as especificações de serviços em *grid* [WS-RF, 2004]. A integração deste modelo com as especificações de segurança, utilizando um modelo de autenticação e autorização também fazem parte dos trabalhos futuros.

Referências Bibliográficas

Aghdaie, N., Tamir, Y. (2002). Implementation and Evaluation of Transparent Fault-Tolerant Web Service with Kernel-Level Support. Proceedings of the IEEE International Conference on Computer Communications and Networks Miami, Florida, pp. 63-68

Anderson, T., LEE, P. A. (1981) Fault tolerance -principles and practice. Englewood Cliffs, Prentice-Hall.

AXIS (2004) Apache Web Services Project, http://ws.apache.org/axis/

Birman, K. P. (1996). Building Secure and Reliable Network Applications. Manning,Greenwich.

Budhiraja, N., Marzulo, K., Schneider, F. B. e Toueg, S. (1993). Distributed Systems, chapter 4. The Primary-Backup Approach. Addison Wesley. 2nd edition.

Chappell D., Jewell T. (2002) Java Web Services. O'Reilly

Cheuk, L., Padilha, R., Souza, L., Fraga, J. (2001). Implementação das Especificações FT-CORBA, Relatório Técnico, LCMI-DAS-UFSC, (http:/www.lcmi.ufsc.br/grouppac).

Coulouris, G., Dollimore, J., (2001) Distributed Systems - Concepts and Design. Addison-Wesley 3º Edition.

Defago, X., Schiper, A., Urban, P. (2000). Totally Ordered Broadcast and Multicast Algorithms. Technical Report DSC/2000/036, Dept. of Communication Systems, EPFL.

Deitel, H.M., .Gadzik, J.P, (2003) Java Web Services for Experienced Programmer. Prentice Hall

Deron L., Fang, C., Chen, C., Lin, F. (2003). FT-SOAP: A Fault-Tolerante Web Service. Tenth Asia-Pacific Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference, Chiang Mai Thailand pp.310

Dialani, V., Miles, S., Moreau, L. De Roure, D., Luck, M. (2002). Transparent Fault Tolerance for Web Services Based Architectures. Euro-Par 2002. Parallel Processing: 8th International Euro-Par Conference Paderborn, Germany Proceedings. Volume 2400

Fabre, J., Deswart, Y. (1994) Design Secure Reliable Applications using

Fragmentation Redundancy Scattering: na Object-Oriented Approach. LAAS

Favarim F., (2003) Componentes em um Esquema de Tolerância a Faltas Adaptativa. Dissertação de Mestrado – Engenharia Elétrica UFSC.

Fraga, J., Cheuk, L., Westphall, C., Montez, C., (2001) Suporte para aplicações críticas nas especificações CORBA: Tolerância a Faltas, Segurança e Tempo Real 19º Simpósio Brasileiro de Redes de Computadores

Ganger, G., Khosla, P., Michael, B. W., Goodson, G. R., Semih, Pandurangn Vijay, (2001) Survivable Storage System. CiteSeer (http://citeseer.ist.psu.edu)

Ghini V., Panzieri F., Roccetti M., (2001) Client-centered Load Distribution: A Mechanism for Constructing Responsive Web Services. Proceedings of the 34th Hawaii International Conference on System Sciences – 2001

Gokhale, A., Kumar, B., Sahuguet A. (2002). "CORBAWeb Services", Proceedings of the 11th International World Wide Web Conference, Honolulu, Hawaii.

Hendricks, M., Galbraith, B., Romin, I., (2002) Professional Java Web Services. Alta Books.

JacOrb (2002) Java Object Request Broker http://www.jacorb.org/docs/ProgrammingGuide_1_4_1.pdf

Jalote, P., (1994) Fault tolerance in distributed systems. Prentice Hall, Englewood Cliffs, New Jersey.

Jandl, M., Radinger, W., Goeschka, K. M. (2003). Integration of CORBA with Directory Services and Web Services, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brasil

J2SE Java Runtime Environment (2003) - Java Sun http://java.sun.com/j2se/1.4.2/download.html

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system.Communications of the ACM, 21(7):558.565.

Mello, R., E., (1993) Redes de confiança em sistemas de objetos CORBA. Dissertação de Mestrado UFSC

OGSA Open Grid Services Architecture (2003) www.globus.org/ogsa/

OMG FT-CORBA Specification (2002). Common Object Request Broker Architecture: CoreSpecification Chapter 23. www.omg.org.

OMG WSDL SOAP to CORBA Interworking (2003) OMG Document.

http://www.omg.org

Potts, S., Kopack, M., (2002) Web Services. Campus.

Pradhan, D., (1996) Fault-Tolerant Computer Design. Englewood Cliffs, Prentice-Hall.

Santos, G., Lung, L., Montez, C., (2005). FTWeb: A Infrastructure for Fault Tolerant "The Enterprise Computing Conference" Ninth International IEEE EDOC Conference

Schneider, F. B. (1990). Implementing Fault-Tolerant Service Using the State Machine Approach: A Tutorial, ACM Computing Survey, 22(4):299-319.

Schneider, F. B., (1990), "Implementing Fault-Tolerant Service Using the State Machine Approach: A Tutorial", ACM Computing Survey, 22(4):299-319.

Shuping, R., (2003) A Model for Web Services Discovery With QoS CSIRO Mathematical and Information Sciences

SOAP Simple Object Access Protocol (2003) – World Wide Web Consortium http://www.w3c.org/TR/soap/

Tan, S., Vellanki, V., Xing, J., Topol B., Dudley G. Service Domains (2004). IBM System Journal VOL 43, N4.

Townend, P., Xu, J. (2004). Replication-based Fault Tolerance in a Grid Environment, Proceedings of the UK e-Science All Hands Meeting

UDDI Universal Description, Discovery and Integration (2002) – OASIS http://www.oasis-open.org/committees/uddi-spec/doc/contribs.htm#uddiv1

Valdes A., Almgren M., Cheung, S., Deswarte Y., Dutertre B., Levy J., Saiidi H., Stavridou V. and Uribe T., (2002) "An Architecture for an Adaptive Intrusion-Tolerant Server" LAAS

Weber, T., (1999), Tolerância a faltas. Instituto de Informática – UFRGS

Westphall, C.M., (1998) Esquemas de Autorização para Programação Distribuída combinando os Modelos de Segurança Java/CORBA/Web, Exame de Qualificação de Doutorado, LCMI-DAS-UFSC

WS-ADD Web Services Addressing (2002) W3C World Wide Web Consortium http://www.w3.org/2002/ws/addr/

WS-ARCH Web Services Architecture (2004) W3C World Wide Web Consortium http://www.w3.org/TR/ws-arch/

WSAD WebSphere Studio Application Developer (2003) http://www-

106.ibm.com/developerworks/websphere/downloads

WSDL Web Services Description Language (2001) – World Wide Web Consortium http://www.w3c.org/TR/wsdl/

WS-I Web Service Interoperability Organization Basic Profile 1.0 (2004). http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html

WSIF Web Service Invocation Framework (2003) Apache Web Services Project http://ws.apache.org/wsif

WS-RF OGSI - Open Grid Services Specification (2004). 1.0 http://www-128.ibm.com/developerworks/library/ws-resource/ws-wsrf.pdf

WS-RM Web Services Reliable Messaging Specification (2004). 1.0 ftp://www6.software.ibm.com/software/developer/library/ws-reliablemessaging200502.pdf

Wu, T., Malkin, M., Bonch, D., (2000) Build Intrusion Tolerante Applications

XML Extensible Markup Language (2000) – World Web Consortium http://www.w3.org/TR/2000/REC-xml-20001006

Zhang, X., Zagorodnov, D., Hiltunen, M. (2004). Fault-tolerant Grid Services Using Primary-Backup: Feasibility and Performance. Cluster 2004, San Diego, California.