PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

JFAULT:
TOLERÂNCIA A FALHAS TRANSPARENTE UTILIZANDO REFLEXÃO E COMPILAÇÃO DINÂMICA NO MODELO DE META-NÍVEIS

MARCIO GUSTAVO GUSMÃO SCHERER

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Dr. Fernando Luís Dotti

Porto Alegre
2015

Dados Internacionais de Catalogação na Publicação (CIP)

S326J Scherer, Marcio Gustavo Gusmão

JFault : tolerância a falhas transparente utilizando reflexão e compilação dinâmica no modelo de meta-níveis / Marcio Gustavo Gusmão Scherer. – Porto Alegre, 2015.
110 p.

Dissertação (Mestrado) – Faculdade de Informática, PUCRS.
Orientador: Prof. Dr. Fernando Luís Dotti.

1. Informática. 2. Tolerância a Falhas (Informática).
3. Sistemas Distribuídos. I. Dotti, Fernando Luís.
II. Título.

CDD 004.36

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "*JFault*: Tolerância a Falhas Transparente Utilizando Reflexão e Compilação Dinâmica no Modelo de Meta-Níveis" apresentada por Márcio Gustavo Gusmão Scherer como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, aprovada em 27/03/2015 pela Comissão Examinadora:

Prof. Dr. Fernando Luís Dotti– Orientador — PPGCC/PUCRS

Prof. Dr. Avelino Francisco Zorzo– — PPGCC/PUCRS

Prof. Dr. Elias Procópio Duarte Júnior– — UFPR

Homologada em 23/04/2015, conforme Ata No. 006 pela Comissão Coordenadora.

Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

PUCRS
Campus Central
Av. Ipiranga, 6681 – P32– sala 507 – CEP: 90619-900
Fone: (51) 3320-3611 – Fax (51) 3320–3621
E-mail: ppgcc@pucrs.br
www.pucrs.br/facin/pos

JFAULT: TOLERÂNCIA A FALHAS TRANSPARENTE UTILIZANDO REFLEXÃO E COMPILAÇÃO DINÂMICA NO MODELO DE META-NÍVEIS

RESUMO

Tolerância a falhas tornou-se um importante meio para se garantir alta disponibilidade de sistemas computacionais. No entanto, a construção de arquiteturas tolerantes a falhas não é uma tarefa trivial. Reflexão em arquiteturas de meta-nível tem sido usada há anos como um meio para implementação de requisitos não-funcionais. Dessa forma é possível ter uma separação clara e em níveis entre a implementação da lógica de negócios do sistema (requisitos funcionais) e as relacionadas ao uso da aplicação em termos de desempenho, usabilidade, segurança, disponibilidade, etc (não funcionais). Estes níveis se tornaram conhecidos na literatura, respectivamente, como nível-base e meta-nível e são frequentemente utilizados em sistemas hoje em dia visto que trazem vários benefícios como aumento de reuso de código e redução de acoplamento entre os elementos da arquitetura, além de trazer uma melhor divisão de responsabilidades entre os componentes do sistema. Por outro lado, se as arquiteturas de meta-nível se tornaram um artifício útil, existe a necessidade de se implementar os componentes de meta-nível responsáveis pela criação de serviços tolerantes a falhas, o que envolve esforço de desenvolvimento, adaptações no sistema e geralmente adiciona certa complexidade à arquitetura.

Este trabalho apresenta uma proposta de construir, de forma automática e em tempo de execução, os componentes de meta-nível para tolerância a falhas em serviços de aplicações. Mais precisamente, pretende propor um *framework* – chamado JFault - que usando reflexão e compilação dinâmica se propõe a preencher esse requisito de forma transparente e com pequenas alterações no sistema.

O *framework* é implementado em Java, linguagem que suporta tanto reflexão como compilação dinâmica, mas poderia ser construído em qualquer linguagem de programação que suporta tais APIs.

Palavras Chave: Reflexão, Compilação Dinâmica, Dependabilidade, Tolerância a Falhas, Sistemas Distribuídos, Cliente-Servidor, Arquiteturas de Meta-Níveis, Java, Framework

JFAULT: TRANSPARENT FAULT TOLERANCE USING REFLECTION AND DYNAMIC COMPILATION IN A META-LEVEL MODEL

ABSTRACT

Fault tolerance has become an important mean to achieve high availability in computational systems. However, building fault tolerant architectures is not a trivial task. Reflection in Meta-level architectures has been used for years as a mean for implementation of non-functional requirements. In this way it is possible to have a clear separation of its implementation from the implementation of the business logic itself (functional requirements) in layers or levels. These levels have become known, respectively, as base-level and meta-level and are regularly used in nowadays systems' architecture since they bring several benefits such as increased reuse and reduced complexity, furthermore, they provide better responsibilities separation among systems' components.

On the other hand, if the meta-level is a useful architecture artifice there is still the need to build the meta-level components that intend to handle fault tolerance in application's services, the components need to be implemented and integrated to the system's architecture, which involves some development effort and complexity.

This work presents a proposal to build, automatically and in runtime, the meta-level components for fault tolerance handling in application's services. More precisely, it intends to propose a framework – named JFault – which using reflection and dynamic compilation will leverage those requirements transparently and with minor changes in the system.

The framework is implemented in Java, language that supports both reflection and dynamic compilation, but could be built in any programming language that supports such APIs.

Keywords: Reflection, Dynamic Compilation, Dependability, Fault Tolerance, Distributed Systems, Client-Server, Meta-Level Architectures, Java, Framework

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
JSE	Java Standard Edition
JDBC	Java Database Connectivity
JVM	Java Virtual Machine
RMI	Remote Method Invocation
SLA	Service Level Agreement
TI	Tecnologia da Informação

SUMÁRIO

1 INTRODUÇÃO

Alta disponibilidade de sistemas computacionais tem se tornado cada vez mais essencial para muitas empresas e um importante fator a ser considerado para que estas se estabeleçam e se mantenham competitivas no mercado atual em muitos negócios. Mais do que aplicações que apresentem poucos períodos de indisponibilidade, as empresas têm cada vez mais almejado sistemas que se mantenham operacionais em tempo integral. Minutos de indisponibilidade dos sistemas podem causar perdas consideráveis para o negócio e prejuízos irreparáveis à imagem da empresa. Segundo Verma et al. [VER11], esses prejuízos podem ser ainda maiores quando os sistemas estão sendo utilizados em aplicações de operação crítica como, por exemplo, para controle de usinas nucleares, controle de espaço aéreo, sistemas automobilísticos, etc. Falhas nesses sistemas podem causar enormes perdas de investimento, esforço, danos ao meio ambiente ou até mesmo colocar vidas em risco.

No campo da Tecnologia da Informação (TI), os profissionais que arquitetam e desenvolvem sistemas tentam concentrar esforços na construção de arquiteturas que possam ser cada vez mais seguras, escaláveis, tolerantes a falhas e, assim altamente disponíveis e confiáveis. O segmento de estudo e desenvolvimento de técnicas sobre este aspecto da computação distribuída cunhou o termo dependabilidade[1] para expressar essa relação de confiança entre os componentes de um sistema. Segundo Avizienis et al., de outra maneira, podemos dizer que dependabilidade é uma propriedade dos sistemas computacionais que define a capacidade dos mesmos de prestar um serviço no qual se pode justificadamente confiar [AVI04].

Um importante meio utilizado para alcançar alta disponibilidade, e consequentemente aumentar a dependabilidade de sistemas computacionais é a utilização de técnicas de tolerância a falhas, aplicada em arquiteturas com componentes redundantes. Um sistema é considerado tolerante a falhas se o seu comportamento é consistente com a sua especificação, mesmo na presença de falhas ativas em seus subsistemas ou componentes; em outras palavras, as técnicas devem mascarar as falhas dos subsistemas de forma que estas não o impactem [JAL94].

[1] O termo *dependability* também é conhecido como confiabilidade ou segurança de funcionamento em português, embora alguns autores já estejam usando o próprio termo dependabilidade como tradução.

Requisitos tais como disponibilidade, segurança e confiabilidade, também conhecidos como requisitos não funcionais, e os meios para alcançá-los (como técnicas de tolerância a falhas e escalabilidade), apresentam-se com frequência como um desafio para os arquitetos de sistemas. Esses desafios incluem não somente como implementá-los de forma efetiva mas também como possuir uma clara separação entre os componentes que implementam estes dos que implementam os requisitos de negócio (ou funcionais) das aplicações. Nesse respeito, a reflexão computacional no modelo arquitetural de meta-níveis vem sendo pesquisada e utilizada há anos, por exemplo, nos trabalhos de Zorzo et al. [ZOR96], Xu et al. [XUJ96] e Killijan et al. [KIL98], pois se propõe a dividir os componentes que implementam os requisitos funcionais dos não funcionais em camadas ou níveis.

Por outro lado, se as arquiteturas de meta-nível se tornaram um artifício útil, a implementação dos componentes de meta-nível para tolerância a falhas em serviços distribuídos ainda se faz necessária, o que envolve esforço de desenvolvimento, adaptações no sistema e geralmente adiciona certa complexidade à arquitetura. Thomas et al. [TOM03] utilizaram reflexão para minimizar os esforços de criação de objetos de meta-nível para tolerância a falhas em serviços e processos com o framework FT-Java, todavia, mesmo com a utilização do framework, o desenvolvedor ainda necessita de adaptações na aplicação e esforços de implementação. As técnicas de reflexão podem fornecer artifícios para se inspecionar e possivelmente alterar o comportamento de componentes com o sistema em pleno funcionamento; contudo, para que a geração desses componentes em meta-nível possa ser possível, o sistema precisa de alguma forma gerar e, sobretudo, compilá-los dinamicamente, em tempo de execução.

De acordo com Shankar et al. [SHA07], o termo compilação dinâmica se refere à geração de código executável em *runtime* (tempo de execução). Além de possuir suporte à compilação dinâmica, algumas linguagens de programação permitem também que a API de compilação seja acessada programaticamente, esse recurso é encontrado frequentemente em linguagens de alto nível como C#, C++, Java e VB [MI214] [MIC14]. No caso do Java, o acesso programático à API de compilação dinâmica foi disponibilizado na versão 6 [KUR06].

Tendo os conceitos até agora mencionados em vista, esse trabalho apresenta uma proposta de construir, de forma transparente e em tempo de execução, os objetos de meta-nível para tolerância a falhas em serviços distribuídos. Mais precisamente, pretende

propor um *framework* - JFault - que usando reflexão e compilação dinâmica de acesso programático se propõe a alavancar esses requisitos de forma transparente e pouco intrusiva, com pequenas adaptações no sistema.

O *framework* é implementado em Java, linguagem que suporta tanto reflexão quanto compilação dinâmica, mas pode ser construído em qualquer linguagem de programação que suporta tais requisitos. Para demonstrar a usabilidade do JFault, como caso de uso, este trabalho apresenta também a utilização do *framework* em aplicações cliente-servidor escritas em Java Standard Edition (JSE). Nesse contexto, o JFault é usado para criar os objetos de meta-nível, que expõem dinamicamente os serviços de negócio do sistema de forma remota, inserindo nestes mecanismos de tolerância a falhas. Os objetos de meta-nível são criados com base nas assinaturas dos métodos dos serviços o que torna o *framework* pouco intrusivo no sentido de que poucas mudanças são necessárias na aplicação para sua utilização. O objetivo do *framework* é demonstrar os conceitos propostos sendo implementados e utilizados na prática.

1.1 OBJETIVOS

São apresentados nas Seções 1.1.1 e 1.1.2, o objetivo geral e os específicos desta da pesquisa, respectivamente.

1.1.1 Geral

Demonstrar, através da criação de um *framework*, como a associação de técnicas de reflexão e compilação dinâmica pode ser utilizada para criar, sem esforço de desenvolvimento, objetos de meta-nível responsáveis por tolerar falhas em serviços distribuídos (nível-base).

1.1.2 Específicos

De forma a complementar o objetivo geral proposto, apresentam-se os seguintes objetivos específicos:

- revisar e apresentar brevemente os principais conceitos de dependabilidade e tolerância a falhas;
- expor os conceitos de reflexão computacional no modelo arquitetural de meta-níveis; apresentar como o modelo arquitetural de meta-níveis vem sendo utilizado para implementação de requisitos não funcionais citando trabalhos relacionados na área; apresentar algumas linguagens de programação que suportam reflexão e demonstrar exemplos;
- explanar os conceitos relativos à compilação dinâmica; apresentar algumas linguagens de programação que suportam compilação dinâmica de acesso programático e demonstrar exemplos;
- apresentar os principais conceitos relativos a meta-tags, arquiteturas cliente-servidor e as tecnologias Java envolvidas na criação do *framework* proposto;
- apresentar a proposta do framework JFault, explicando em detalhes como as técnicas de reflexão e compilação dinâmica podem ser utilizadas para construção de objetos de meta-nível em tempo de execução. Apresentar os principais componentes do *framework* a nível conceitual;
- apresentar os detalhes da implementação do *framework* em Java; demonstrar como o *framework* pode ser utilizado em aplicações JSE cliente-servidor para

alavancar de forma minimamente intrusiva os requisitos de tolerância a falhas em serviços remotos;

- realizar experimentos demonstrando em detalhes o processo de tolerância a falhas de Colapso e Tempo, injetando essas falhas em uma aplicação que será utilizada como caso de uso para demonstração do funcionamento do *framework*;
- realizar experimentos para verificar e avaliar o impacto do *framework* no processo de comunicação entre o cliente e o servidor, bem como o tempo necessário para inicialização do *framework* à medida que o número de serviços a serem suportados cresce.

1.2 ORGANIZAÇÃO DO TRABALHO

O Capítulo 2 apresenta uma breve revisão bibliográfica sobre os principais conceitos e tecnologias utilizadas nesse trabalho. Nas Seções 2.1 e 2.2, são apresentados os principais conceitos de dependabilidade e técnicas de tolerância a falhas a fim de introduzir o assunto ao leitor. Nas Seções 2.3 e 2.4 são apresentados conceitos e trabalhos relacionados sobre reflexão em arquiteturas de meta niveis e compilação dinâmica, conceitos e tecnologias chaves para o *framework* proposto. Ainda no Capítulo 2, são apresentados posteriormente conceitos sobre meta-tags, arquiteturas cliente servidor e os principais aspectos da tecnologia Java Standard Edition, que serão utilizados diretamente na implementação do *framework.* No Capítulo 3 é abordado a proposta do *framework* JFault, primeiramente a nível conceitual, onde os conceitos do *framework* são discutidos independentes de implementação. Posteriormente, uma implementação do *framework* utilizando a linguagem de programação Java é apresentada, juntamente com um exemplo de caso de uso e alguns experimentos. No Capítulo 4, abordaremos em maiores detalhes alguns trabalhos relacionados a esse estudo. Por fim, no Capítulo 5, são apresentadas as considerações finais.

2 REFERENCIAL TEÓRICO

Este capítulo apresenta uma base teórica referente aos principais conceitos que envolvem essa pesquisa. Na Seção 2.1 é apresentada uma revisão bibliográfica sobre os conceitos de dependabilidade de sistemas. As Seções 2.2 e 2.3 apresentam uma revisão bibliográfica nos conceitos e técnicas sobre tolerância a falhas. As Seções 2.4 e 2.5 apresentam os conceitos referentes à reflexão em arquiteturas meta-nível e compilação dinâmica, respectivamente, conceitos e tecnologias que serão posteriormente importantes para o entendimento da proposta do *framework*. As Seções posteriores apresentam respectivamente alguns conceitos sobre meta-tags, arquiteturas cliente-servidor e um breve contexto sobre a tecnologia Java, cobrindo os principais aspectos relacionados à versão Java Standard Edition (JSE).

2.1 DEPENDABILIDADE

Sistemas computacionais podem ser caracterizados por quatro propriedades fundamentais: funcionalidade, desempenho, custo e dependabilidade [AVI04]. Como visto anteriormente, a dependabilidade pode ser descrita como uma propriedade dos sistemas computacionais que define a capacidade dos mesmos de prestar um serviço no qual se pode justificadamente confiar [AVI04].

A dependabilidade pode também ser vista como um conceito macro e, de acordo com Avizienis et al. [AVI04], é divida em três partes: ameaças (ameaças contra a dependabilidade), atributos e meios (meios para alcançar a dependabilidade). A Figura 1 lista os principais elementos da taxonomia da dependabilidade.

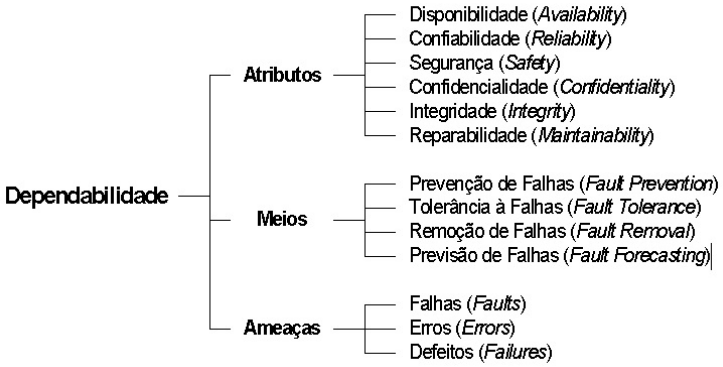

Figura 1 – Taxonomia da dependabilidade

Fonte: [AVI04]

O serviço fornecido por um sistema é o seu comportamento, tal qual é percebido por seus usuários. Um usuário de um serviço é outro sistema, eventualmente até o próprio ser humano, que interage com o primeiro através da interface do serviço. A função de um sistema é que ele se dispõe a fazer, o que é descrito pela sua especificação [DAN05].

2.1.1 Ameaças

Um serviço é chamado correto quando implementa a funcionalidade do sistema, ou seja, responde corretamente de acordo com sua especificação. Um sistema apresenta um defeito quando não é capaz de prestar um serviço correto, ou seja, seu serviço se desvia da especificação do sistema. O defeito é o evento que causa a transição de estado do serviço de um sistema de correto para incorreto. A transição do serviço incorreto para o correto é chamada de restauração do serviço (*service restoration*), enquanto o intervalo de tempo em que o serviço está sendo fornecido de forma incorreta é chamado de interrupção do serviço (*outage*). Um defeito somente ocorre quando um erro existente no sistema alcança a interface do serviço e altera um serviço prestado [AVI04].

O erro é um estado indesejado do sistema que pode ou não vir a causar um defeito. Dessa forma, um sistema pode ter um ou mais erros e continuar a prestar um serviço correto, sem defeito. O tempo entre o surgimento do erro e a manifestação do defeito é chamado de latência do erro. A transição de estados em um sistema devido à ocorrência de defeitos é ilustrada na Figura 2 [DAN05].

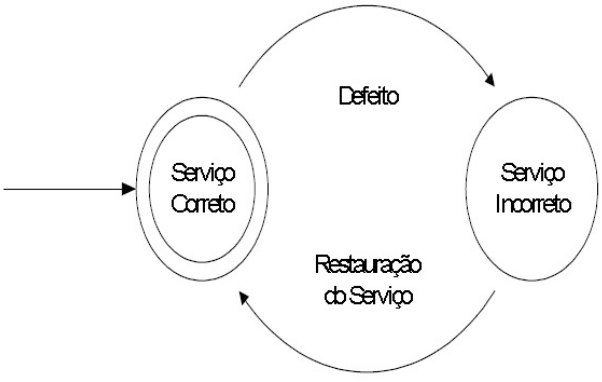

Figura 2 - Restauração do Serviço

Fonte: [DAN05]

Avizienis et al. [AVI04] afirmam que, quanto ao domínio, os defeitos de valor (ou conteúdo) ocorrem quando o conteúdo da informação que está sendo enviado na interface de serviço está incorreto, ou seja, possui algum desvio se comparado à especificação. Já as falhas de tempo, são a respeito do tempo em que a informação demora a chegar na interface de serviço ou sua duração: quando estas se desviam da especificação do sistema possuímos um defeito de tempo. Quando possuímos ambos defeitos em um sistema (conteúdo e tempo) então o defeito é caracterizado em duas classes:

1) **Defeito de parada (*halt failure*):** quando o serviço é interrompido, ou, em outras palavras, quando a atividade do sistema, se existir, não mais é perceptível ao usuário;
2) **Defeito errático (*erratic failures*):** quando o serviço está sendo entregue (não foi interrompido), porém de forma errática, com algum desvio se comparado à especificação. Esse desvio pode ser tanto uma demora excessiva na resposta (*delay*) quanto respostas corretas e incorretas sendo retornadas arbitrariamente.

No que diz respeito à percepção pelos usuários, um defeito pode ser consistente ou inconsistente. Um defeito consistente é aquele em que o serviço incorreto é percebido da mesma forma por todos os usuários. Já um defeito inconsistente também conhecido como “Bizantino”, é percebido de diferentes formas. Possivelmente, para alguns usuários o serviço pode até mesmo estar sendo entregue de uma forma correta e, para outros, de forma incorreta [AVI04].

Os defeitos Bizantinos, de acordo com Coulouris et al. [COL07], são aqueles onde qualquer tipo de falha pode ocorrer. Por exemplo, um processo pode atribuir aleatoriamente valores incorretos a seus dados ou retornar um valor igualmente incorreto em resposta a uma invocação. Pelo fato de os defeitos serem arbitrários, também não podem ser detectados verificando se o processo responde às invocações, pois ele poderia omitir arbitrariamente a resposta incorreta.

Embora semelhante ao modelo descrito por Avizienis et al. [AVI04], Hadzilacos apud Tanenbaum et al. [TAN06] descrevem uma classificação ligeiramente diferente para os tipos possíveis de defeitos, como pode ser visto na Tabela 1.

Tabela 1 – Tipos de defeitos

Defeito	Descrição
Colapso (*Crash Failure*)	O serviço para, mas estava funcionando corretamente até parar. Um importante aspecto desse tipo de defeito é que uma vez que ele ocorra nada mais é retornado ou executado.
Omissão (*Omission Failure*)	O serviço falha ao receber ou enviar requisições ou mensagens. Um exemplo típico são defeitos causados por falhas de sistema (software) como laços infinitos ou incorreto gerenciamento de memória.
Tempo (*Timing Failure*)	O serviço responde, mas fora do tempo limite especificado.
Resposta (*Response Failure*)	A resposta retornada pelo serviço está incorreta.
Arbitrária ou Bizantina (*Arbitrary Failure*)	O serviço retorna respostas incorretas arbitrariamente em diferentes momentos.

Fonte: [TAN06]

Quanto às consequências no ambiente (também chamadas de “severidade”, nos estudos de Avizienis et al. [AVI04]), um defeito pode ser caracterizado desde defeitos menores, de baixo impacto para as funcionalidades, até defeitos catastróficos, que podem interromper totalmente um ou mais serviços ou até mesmo causar total indisponibilidade da aplicação.

2.1.2 Meios

Avizienis et al. [AVI04] colocam que a dependabilidade pode ser alcançada no desenvolvimento de sistemas computacionais utilizando-se a combinação de quatro técnicas:

1) **Prevenção de falhas**: como evitar a ocorrência ou introdução de falhas em um sistema;
2) **Tolerância a falhas**: como continuar o fornecimento de um serviço correto mesmo na presença de falhas;

3) **Remoção de falhas**: como remover/reduzir o número de falhas (ou reduzir sua severidade) em um sistema;
4) **Previsão de falhas**: como estimar a quantidade atual, a incidência futura de falhas e a potencial consequência das mesmas no sistema.

2.1.3 Atributos

2.1.3.1 Disponibilidade

Toeroe [TOE12] define o termo disponibilidade (*availability*) como sendo a proporção do tempo em que o sistema está em plenas condições de realizar suas funções. O autor apresenta a medida da disponibilidade como um percentual de tempo em que o sistema é capaz de prover seus serviços corretamente durante um intervalo de tempo. Por exemplo, uma disponibilidade de 100% significa que o sistema não teve nenhuma interrupção em seus serviços durante o tempo em questão.

O autor ainda afirma que, geralmente, um sistema é dito altamente disponível (*high available*) quando possui ao menos 99.999% de disponibilidade.
Essa medida de alta disponibilidade também ficou conhecida na literatura como cinco 9s (five nines) e, na prática, representa dizer que o sistema pode apresentar no máximo 5 minutos de indisponibilidade por ano [LEM07].

2.1.3.2 Confiabilidade

Toeroe [TOE12] define a confiabilidade (*reliability*) como a habilidade de um sistema em realizar uma função específica corretamente sobre determinadas condições durante um período de tempo definido. Para Dantas [DAN05], a confiabilidade, enquanto atributo, é quantificada como a probabilidade “R(t)” e deve ser entendida como uma métrica que avalia o quanto um sistema pode apresentar um serviço correto continuadamente durante um intervalo de tempo “t”, dado que este sistema apresentava serviço correto em t=0.

2.1.3.3 Segurança de Funcionamento

Avizienis et al. [AVI00] definem o atributo segurança de funcionamento (*safety*) como sendo a ausência de defeitos (*failure*) catastróficos no sistema que possam trazer grandes prejuizos para o usuário ou ambiente. Quando o estado correto do serviço e os

estados incorretos (devido a defeitos não catastróficos) são agrupados em um estado seguro (no sentido de serem livres de danos catastróficos) segurança é a medida da continuidade do serviço seguro. Em outras palavras, segurança é o mesmo que o atributo confiabilidade (*reliability*) com respeito a defeitos catastróficos.

2.1.3.4 Confidencialidade

Segundo Pfleenger [PFL03] a confidencialidade (confidentiality) visa garantir que somente quem deve ter acesso a algo será capaz de acessá-lo. Por acesso, estão incluídos não só acesso de leitura, mas também de impressão e mesmo do simples conhecimento que determinada coisa ou informação existentes.

Dantas [DAN05] afirma que a confidencialidade de um sistema tem probabilidade “C(t)” de não ocorrer divulgação não autorizada de informação, em um intervalo de tempo “t”.

2.1.3.5 Integridade

Integridade é definida por Pfleenger [PFL03] como um atributo que visa garantir o nível de confiança sobre um determinado ativo (*asset*). Se um ativo é íntegro, isso significa que se ele foi modificado, então foi modificado somente por pessoas ou processos autorizados. Neste contexto, modificações incluem criar, alterar ou remover um ativo.

2.1.3.6 Reparabilidade

O grau de reparabilidade (*maintainability*) avalia o quanto um sistema poder ser restaurado, retornando ao estado de serviço correto no tempo “t”, dado que o mesmo apresentou defeito e “t=0”. Em resumo, o grau de reparabilidade é a capacidade que um sistema tem de passar por reparos e modificações [DAN05].

2.2 TOLERÂNCIA A FALHAS

Um sistema é considerado tolerante a falhas se o seu comportamento é consistente com a sua especificação, mesmo na presença de falhas ativas em seus subsistemas ou componentes. Um fato importante a se observar é que a tolerância a falhas implica necessariamente em uma recuperação automática do sistema no caso de falhas, que

deve ser feita de forma a não causar impacto ou, no máximo, uma leve degradação do sistema. A este respeito, Jalote [JAL94] defende que o reparo do sistema deve ser feito sem interrupções e sem intervenção manual para que o sistema possa ser considerado tolerante a falhas. Caso o reparo tenha de ser realizado manualmente, então, segundo o autor, o sistema não pode ser considerado tolerante a falhas.

2.3 TÉCNICAS DE TOLERÂNCIA A FALHAS

Avizienis et al. [AVI04] afirmam que as técnicas de tolerância a falhas devem ser usadas para preservar o correto funcionamento dos serviços mesmo na presença de falhas.

Tanto Avizienis et al. [AVI04] quanto Jalote [JAL94] afirmam que as principais técnicas envolvidas no processo de tolerância a falhas são a detecção de erros e a recuperação do sistema. Avizienis et al. apresentam uma classificação completa para as técnicas de tolerância a falhas na Figura 3.

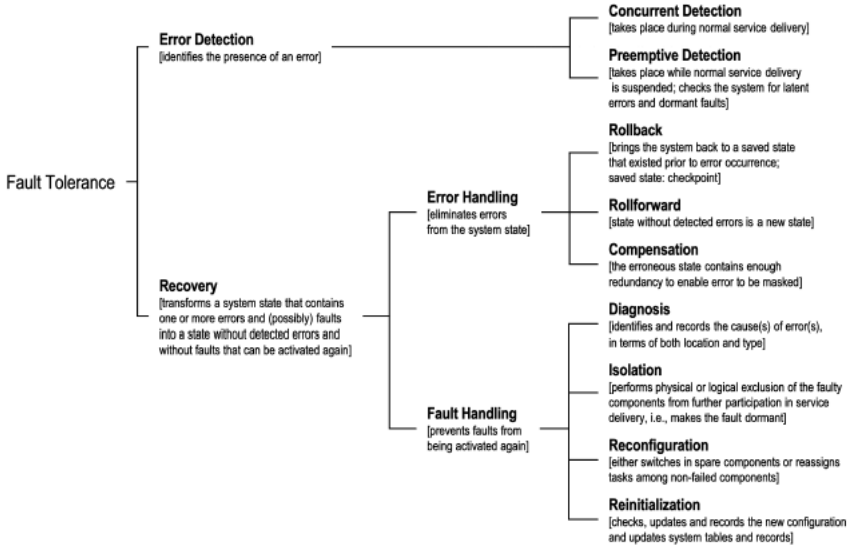

Figura 3– Técnicas de Tolerância a Falhas

Fonte: [AVI04]

De acordo com Jalote [JAL94], a detecção de erros (*Error Detection*) é o ponto de partida para tolerância a falhas. Visto que o erro é definido pelo estado do sistema (ou subsistema), verificações podem ser feitas para se identificar a presença de erros. A eficiência de um mecanismo tolerante a falhas vai depender muito do quão efetivo é o mecanismo de detecção de erros.

Ainda segundo Jalote [JAL94], detecção concorrente (*Concurrent Detection*) é uma técnica aplicada a fim de verificar erros no sistema ainda em execução, enquanto o serviço está executando sua função. Já a detecção preemptiva (*Preemptive Detection*) visa localizar erros enquanto o sistema está suspenso, procurando tanto erros latentes quanto falhas dormentes.

Uma vez que o erro é encontrado, ele precisa ser eliminado ou mascarado de alguma forma para que o mesmo não impacte a aplicação como um todo. Este é o foco das técnicas de recuperação de erros (*Recovery*). O tratamento de erros (*Error Handling*) é composto das seguintes técnicas [AVI04]:

1) **Retrocesso (*Rollback*):** o estado do sistema, enquanto o mesmo está sendo executado de forma correta, é salvo em imagens chamadas “ponto de checagem” (*checkpoints*). Quando erros são detectados nos componentes ou subsistemas, essa imagem é recuperada, trazendo o sistema a um estado anterior (teoricamente correto);
2) **Avanço (*Rollforward*):** não existe estado anterior, a técnica visa levar o sistema a um novo estado tentando remover os erros existentes, tomando as medidas corretivas necessárias;
3) **Compensação (*Compensation*):** o serviço que apresenta erros é isolado e compensado por outro serviço. Dessa forma, o subsistema ou componente auxiliar assume o serviço e mascara o erro. É importante observar que para um serviço ser compensado deve existir necessariamente alguma forma de redundância, ou seja, algum componente idêntico ou similar ao que está indisponível ou agindo de forma errônea que possa assumir o serviço.

Ainda de acordo com os estudos de Avizienis et al. [AVI04], o tratamento de falhas (*Fault handling*) visa garantir que as falhas ocorridas não se tornarão novamente ativas e é composto das seguintes técnicas:

1) **Diagnóstico (*Diagnosis*):** identifica e registra a causa dos erros, sua localização e o tipo;

2) **Isolamento (*Isolation*):** altera o sistema física e logicamente, de forma que o componente falho não participe mais durante a execução do serviço. Em outras palavras, esta técnica torna a falha “dormente”;
3) **Reconfiguração (*Reconfiguration*):** muda os componentes falhos ou reenvia as tarefas para os componentes não falhos;
4) **Reinicialização (*Reinitialization*):** verifica, atualiza e registra a nova configuração e os registros do sistema.

2.4 REFLEXÃO EM ARQUITETURAS DE META-NÍVEIS

Smith apud Robben et. al [BER98] coloca que arquiteturas de meta-níveis tem se tornado um importante tópico de pesquisa em programação orientada a objetos. Essa área é muito próxima da programação reflexiva visto que ambas compartilham do mesmo objetivo: habilitar os desenvolvedores de aplicações a codificarem sistemas que possam de alguma forma manipular o seu próprio estado de execução.

Segundo Maes [MAP87], reflexão pode ser descrita como a habilidade de um programa inspecionar a si mesmo em tempo de execução e possivelmente modificar o seu próprio comportamento usando uma representação de si mesmo. Esse modelo de representação é conhecido na literatura como meta-modelo (*meta-model*), que é conectado com o sistema real de tal forma que se ocorrerem mudanças no meta-modelo então essas são refletidas no sistema real, potencialmente modificando o seu comportamento. Da mesma forma, mudanças no sistema real são também refletidas no meta-modelo. A conexão entre os modelos é realizada através de uma interface, chamada de meta-interface.

Um sistema reflexivo é composto de uma meta-interface e dois níveis: um nível-base (*base-level*), onde a computação normal do sistema ocorre, e um meta-nível (*meta-level*), onde os aspectos abstratos do sistema são computados. Através do uso da meta-interface o meta-nível pode coletar informação ou executar operações do nível-base (processo chamado *reification*) e computar os aspectos não funcionais do sistema, eventualmente interferindo nele e alterando seu comportamento (processo chamado reflexão). Esse princípio, quando utilizado, fornece uma separação clara da computação normal do sistema (requisitos funcionais) dos seus aspectos não funcionais [VAL06].

Barth et al. [BAR05] complementam dizendo que arquiteturas de meta-níveis têm sido adotadas para expressar características não funcionais, como confiança (r*eliability)* e

segurança (s*afety)*, de forma independente do domínio da aplicação. No nível-base encontram-se os objetos que implementam as funcionalidades do sistema, enquanto no meta-nível*,* estão as estruturas de dados e ações que são realizadas sobre os objetos do nível-base. O limite de domínio é o aspecto mais interessante das arquiteturas reflexivas segundo o autor, não somente porque permite a construção de sistemas adaptáveis, mas também porque estimula o reuso de componentes. O principal ponto é permitir que o programador da aplicação se concentre na solução específica do problema do domínio da aplicação.

A Figura 4 representa graficamente um exemplo de processo de reflexão em um sistema computacional. O sistema é dividido em dois ou mais níveis computacionais. O usuário envia uma mensagem ao sistema; a mensagem é tratada pelo nível funcional, que é responsável por realizar o trabalho corretamente, enquanto o nível não funcional gerencia o trabalho do nível funcional.

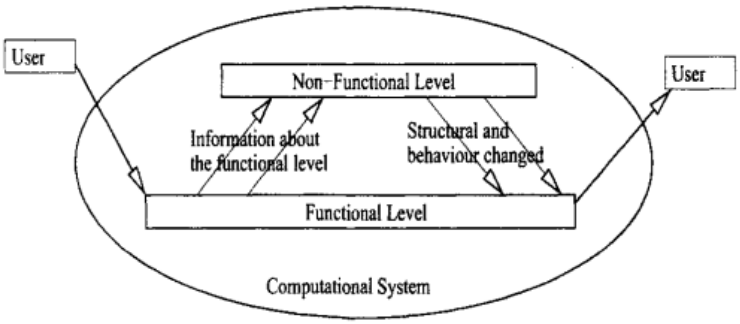

Figura 4– Aplicação utilizando uma arquitetura reflexiva de meta-níveis

Fonte: [BAR05]

Com essa característica, torna-se mais simples modificar e adaptar a estrutura e o comportamento do sistema. A separação em dois níveis agrega benefícios aumentando o reuso e diminuindo a complexidade, fazendo o sistema ficar mais flexível no geral.

As características principais da reflexão computacional segundo Wu et al. apud Barth são [BAR05]:

1) separar as habilidades básicas das não básicas através de níveis arquiteturais;
2) satisfazer os requisitos funcionais do sistema nos objetos da aplicação, ou seja, no nível base;
3) satisfazer os requisitos não funcionais da aplicação através de objetos-meta específicos, ou seja, no meta-nível;

4) os objetos no nível base (*base-level*) podem ser estruturalmente e comportamentalmente modificados em tempo de execução ou compilação.

Maes [MAP87] ainda descreve a existência de dois modelos de reflexão computacional: a reflexão estrutural (*structural*) e a comportamental (*behavioural*). O modelo de reflexão estrutural é focado nos aspectos estruturais do programa (tipos de dados, assinaturas de métodos e estrutura de classes, por exemplo), enquanto a reflexão comportamental expõe o comportamento e o estado do sistema.

A Figura 5 ilustra os conceitos graficamente representando um sistema comportamental reflexivo. Entidades podem ser reflexivas, como A, C e X, e não reflexivas, como Z. Nem todas as entidades precisam necessariamente refletir em um único objeto-meta (*meta-object*). C0 e C1, por exemplo, possuem um objeto-meta único C'. Entidades A, C0 e C1 pertencem a um grupo com funcionalidades relacionadas, que é refletido no modelo-meta também.

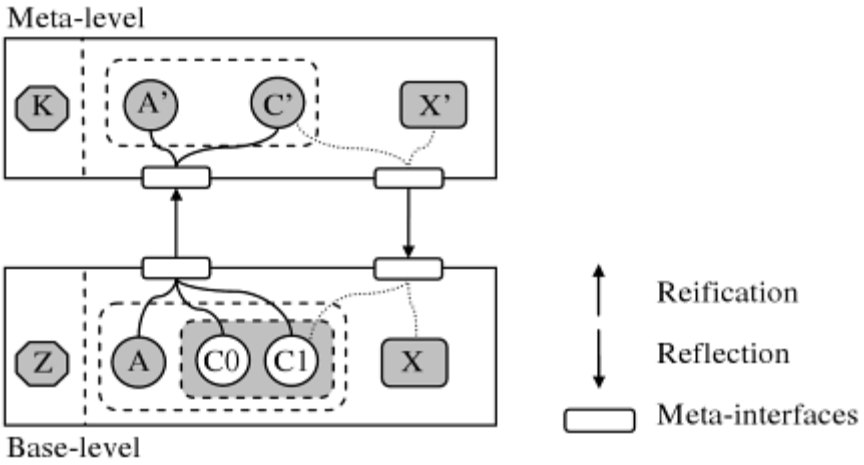

Figura 5– Meta-level e base-level

Fonte: [VAL06]

Romanovsky [ROM01], em seus trabalhos relacionados a tratamento de erros (*error handling*) utilizando reflexão, afirma que podem existir protocolos específicos para comunicação em arquiteturas *de* meta-níveis chamados de protocolos de objetos-meta (*meta-object protocol, ou MOP)*. Tais protocolos se propõem a fornecer uma interface de alto nível que ajuda a revelar informações do comportamento e da estrutura interna do sistema, que normalmente são escondidos pelo compilador em tempo de execução.

Segundo Scott [SCO08], nenhuma linguagem de programação implementa reflexão na totalidade. Todavia, afirma que linguagens como Prolog, Java e C# possuem amplos recursos relacionados a reflexão. Ainda segundo o autor, linguagens de script, como Perl, PHP, Tcl, Python, Ruby e Java Script também fornecem recursos de suporte à reflexão. Os recursos variam de linguagem para linguagem, assim como a sintaxe, contudo, todas permitem que o programa explore sua própria estrutura e seus tipos de dados. Do ponto de vista do desenvolvedor, a principal diferença entre reflexão em Java e C# de um lado, e as linguagens de script do outro, é que as linguagens de script são tipadas dinamicamente. Em Ruby, por exemplo, é possível descobrir-se a classe de um objeto, os métodos existentes e o número de parâmetros esperados por cada método, todavia, os parâmetros não são tipados até o método ser chamado. Scott ainda pondera que reflexão é mais “natural” em linguagens de script do que em linguagens compiladas e com tipagem estática como Java e C#.

Na Figura 6 podemos analisar um exemplo de código reflexivo utilizando a linguagem Java. Nesse exemplo, a utilização do método reflexivo “*getDeclaredMethods()”*, na linha 5, retorna informações sobre a estrutura dos métodos declarados na classe passada como parâmetro para o método “*listMethods(String s)”*. Como pode ser visto também na linha 6, o método “*forName()”*, da classe “*Class”*, também utiliza reflexão para criar o objeto “c” dinamicamente, em tempo de execução, dado o nome da classe como parâmetro.

```
1.  import static java.lang.System.out;
2.  public static void listMethods(String s)
3.          throws java.lang.ClassNotFoundException {
4.      Class c = Class.forName(s); // throws if class not found
5.      for (Method m : c.getDeclaredMethods()) {
6.          out.print(Modifier.toString(m.getModifiers()) + " ");
7.          out.print(m.getReturnType().getName() + " ");
8.          out.print(m.getName() + "(");
9.          boolean first = true;
10.         for (Class p : m.getParameterTypes()) {
11.             if (!first) out.print(", ");
12.             first = false;
13.             out.print(p.getName());
14.         }
15.         out.println(") ");
16.     }
17. }
```

Figura 6 – Exemplo de programação reflexiva em Java: métodos declarados

Fonte: [SCO08]

Na Figura 7 é possivel ver o resultado da execução do método quando a classe "*java.lang.reflect.AccessibleObject*" é passada como parâmetro. É interessante notar que todas as informações a respeito do método puderam ser coletadas, como o nível de acesso (*public, private*, etc), o tipo do objeto retornado pelo método, o nome do método e seus parâmetros.

Exemplo de retorno do método listMethods() utilizando a classe "java.lang.reflect.AccessibleObject" como parâmetro:

```
public java.lang.annotation.Annotation getAnnotation(java.lang.Class)
public boolean isAnnotationPresent(java.lang.Class)
public [Ljava.lang.annotation.Annotation; getAnnotations()
public [Ljava.lang.annotation.Annotation; getDeclaredAnnotations()
public static void setAccessible([Ljava.lang.reflect.AccessibleObject;, boolean)
public void setAccessible(boolean)
private static void setAccessible0(java.lang.reflect.AccessibleObject, boolean)
public boolean isAccessible()
```

Figura 7 – Exemplo de retorno de um método reflexivo em Java

Fonte: [SCO08]

Em Java, além de inspecionar elementos, é possível utilizar reflexão para executar métodos de classes que não são conhecidas em tempo de compilação. Um exemplo de execução de métodos de forma dinâmica pode ser visto na Figura 8. Na linha 1, os métodos da classe referente ao objeto "*u*" são capturados e armazenados no objeto "*method1*". Posteriormente, nas linhas 2 e 3, o primeiro método da classe é instanciado e então executado. No exemplo em questão nenhum parâmetro está sendo passado na execução, contudo, se o método tivesse parâmetros, esses poderiam ser passados no próprio método "*invoke*" da linha 3.

```
1. Method uMethods = u.getClass().getMethods();
2. Method method1 = uMethods[1];
3. Object rtn = method1.invoke(u);
```

Figura 8 – Exemplo de programação reflexiva em Java: executando métodos

Fonte: [SCO08]

A API de reflexão do C# é similar à do Java: o tipo básico que define a estrutura do componente "*System.Type*" em C# é análogo ao tipo "*java.lang.Class*" em Java; assim como a *namespace* "*System.Reflection*", pacote que contém a maioria das ferramentas de reflexão, é analogo ao pacote "*java.lang.Reflect*" do Java [SCO08]. A Figura 9 mostra um código em C# que, dado o nome de um módulo (*assembly*), lê todos os tipos de dados existentes. Fazendo uma comparação com o Java, seria como a API de reflexão estivesse sendo utilizada para ler todas as classes que fazem parte de um determinada biblioteca (*.jar*) [KOG08].

```
1. Using System.Reflection;
2. Assembly assembly = Assembly.loadFrom(assemblyName);
3. Type[] type = assembly.getTypes();
```

Figura 9 – Exemplo de programação reflexiva em C#: listando classes

Fonte: [KOG08]

Uma vez que possuímos acesso a lista de tipos disponíveis dentro de um módulo, é possível adquirir várias informações sobre estes, como *namespace*, níveis de acessos, métodos, etc.

2.5 COMPILAÇÃO DINÂMICA

De acordo com Shankar et al. [SHA07] o termo compilação dinâmica se refere à geração de código executável em *runtime* (tempo de execução). Ainda segundo os autores, uma grande vantagem da compilação dinâmica é que o compilador pode usufruir e explorar as informações geradas em tempo de execução, para, por exemplo, realizar otimizações que não estariam disponíveis quando se utiliza compilação estática.

A compilação dinâmica estende a tradicional noção de compilação e geração de código adicionando um novo estágio no modelo clássico de compilar, vincular (*linkining*) e carregar o código (*loading code*) – o estágio de compilação efetuada pelo compilador dinâmico. O compilador dinâmico pode aproveitar a informação gerada em tempo de execução para, por exemplo, customizar um programa de acordo com a informação contida em seus registradores ou seu controle de fluxo atual. Outras oportunidades únicas tratando-se de compilação dinâmica são o potencial de acelerar a execução de código legado e realizar a migração de softwares de uma arquitetura para outra. Mesmo para máquinas que possuem a mesma família de arquitetura, um compilador dinâmico pode ser utilizado para atualizar o software a fim de explorar as capacidades adicionais das novas gerações [SHA07].

Junto com várias oportunidades, a compilação dinâmica também introduz um conjunto grande de desafios. Um dos maiores desafios é a amortização da sobrecarga (*overhead*) gerada pela compilação. Se a compilação dinâmica é sequencialmente intercalada com a execução do programa, o tempo despendido pela compilação diretamente contribui para o tempo total de execução do programa. Essa sobrecarga pode na verdade reduzir muito (ou todo) o tempo ganho com as otimizações geradas pela compilação dinâmica. Se a compilação dinâmica ocorrer em paralelo com a execução do programa em um sistema com múltiplos processadores, a sobrecarga gerada é menos importante, pois a compilação não irá diretamente impactar o desempenho do programa.

Outro ponto a ser considerado quando se utiliza compilação dinâmica é a quantidade de memória necessária para se executar um programa. Visto que a

compilação e a carga do código ocorrem dinamicamente, é possível que a quantidade de memória aumente consideravelmente à medida que o programa é executado. Essa questão deve ser cuidadosamente controlada principalmente em sistemas embutidos (*embedded*) e móveis (*mobile systems*) onde os recursos em geral são mais limitados [SHA07].

Existem várias abordagens que possibilitam compilação e geração dinâmica de código executável. Essas abordagens se diferenciam em vários aspectos, incluindo o grau de transparência, a extensão e o escopo da compilação dinâmica. Uma das abordagens mais transparentes e frequentemente disponíveis em linguagens de alto nível é a JIT (Just-in-time compilation). A compilação JIT refere-se à compilação de código intermediário de máquina virtual (*virtual machine*) em tempo de execução, muito útil, pois gera código portável independente de plataforma. Ou seja, o processo não inicia com o código executável todo compilado.

A compilação JIT foi introduzida para Smalltalk na década de 80, mas se popularizou com linguagens de alto-nível mais recentes como Java e C# [SHA07][LOT10]. No caso do C#, quando um código gerenciado (código gerado pelo compilador para o suporte *CLR*[2]) é compilado, um código intermediário chamado Microsoft Intermediate Language (ou MSIL) é gerado, o qual é independente de CPU. O código contido no MSIL não pode ser executado diretamente; antes, é preciso convertê-lo para instruções que possam ser interpretadas pela CPU. A conversão é realizada por um compilador JIT (.NET Framework just-in-time) que transforma o código intermediário para código específico da plataforma [LOT10]. No Java, o código intermediário é chamado *bytecode*. A compilação no Java pode ser feita em tempo de execução (JIT) ou antes do programa rodar, processo chamado “compilação estática”.

Compilação estática é um procedimento no qual o programa é compilado e gera código executavél antes da inicialização do programa. A compilação estática geralmente produz código executável de alto desempenho, mas não permite que classes sejam carregadas dinamicamente e que o código possa ser executado em múltiplas plataformas. A compilação dinâmica é feita em tempo de execução, permite que classes sejam carregadas em tempo de execução e gera código independente de plataforma, contudo, possui menor desempenho [LAB10].

---

[2] CLR (Commom Language Runtime) é o mecanismo responsável pela execução das aplicações no .NET framework [LOT10].

Algumas linguagens de programação, além de possuírem suporte para compilação dinâmica, permitem que o programador acesse programaticamente o compilador. Em outras palavras, o próprio programa pode criar um recurso novo (uma classe, um tipo, etc) em tempo de execução, invocando a API de compilação com o sistema em pleno funcionamento. Esse recurso é chamado na literatura de compilação dinâmica de acesso programático, ou compilação em memória, e pode ser frequentemente encontrado em linguagens de alto nível como C#, C++, Java e VB [MI214] [MIC14]. A Figura 10 apresenta um exemplo de utilização da API de compilação dinâmica do Java, compilando uma nova classe criada em tempo de execução [ZUK06].

```
1.  import java.lang.reflect.*;
2.  import java.io.*;
3.  import javax.tools.*;
4.  import javax.tools.JavaCompilerTool.CompilationTask;
5.  import java.util.*;
6.  public class CompileSource {
7.      public static void main(String args[]) throws IOException {
8.          JavaCompiler compiler = ToolProvider.getSystemJavaCompilerTool();
9.          DiagnosticCollector<JavaFileObject> diag = new DiagnosticCollector<JavaFileObject>();
10.         StringWriter writer = new StringWriter(); PrintWriter out = new PrintWriter(writer);
11.         out.print("public class HelloWorld{");out.print("public static void main(String args[]) {");
12.         out.print("System.out.println(\" Hello, world\");"); out.print("System.out.println("}}");
13.         out.close();
14.         JavaFileObject file =
15.             new JavaSourceFromString("HelloWorld",writer.toString());//String2FileObject, imp omitted
16.         // compile class invoking dinamic compiler
17.         Iterable<? Extends JavaFileObject> compilationUnits = Arrays.asList(file);
18.         CompilationTask task = compiler.getTask(null,null,diagnostics, null, null, compilationUnits);
19.         task.run();
20.         Boolean success = task.getResult();
21.     }
22. }
```

Figura 10 – Utilização da API de compilação dinâmica no Java

Fonte: [ZUK06]

Nesse exemplo, podemos ver na linha 18 a invocação programática da API de compilação dinâmica do Java: no caso, a API está compilando, em tempo de execução

do programa, a classe “*HelloWorld*” gerada dinamicamente pelo código adicionado no programa entre a as linhas 11 e 13. A classe “*Diagnostic*” pode ser utilizada para obter os detalhes da compilação; essa classe é particularmente útil quando a compilação falha e se faz necessário analisar os detalhes do problema.

Em outras linguagens, como C#, a utilização da API de compilação dinâmica de acesso programático é similar, como pode ser visto na Figura 11.

```
1.  public static bool CompileCode(CodeDomProvider provider, String sourceFile, String exeFile)
    {
2.      CompilerParameters cp = new CompilerParameters();
3.      cp.GenerateExecutable = true; cp.OutputAssembly = exeFile;
4.      cp.IncludeDebugInformation = true; cp.ReferencedAssemblies.Add( "System.dll" );
5.      cp.GenerateInMemory = false; cp.WarningLevel = 3;
6.      cp.TreatWarningsAsErrors = false; cp.CompilerOptions = "/optimize";
7.      cp.TempFiles = new TempFileCollection(".", true);
8.      if (provider.Supports(GeneratorSupport.EntryPointMethod)) {
9.          cp.MainClass = "Samples.Class1";
10.     }
11.     if (Directory.Exists("Resources")) {
12.         if (provider.Supports(GeneratorSupport.Resources)) {
13.             cp.EmbeddedResources.Add("Resources\\Default.resources");
14.             cp.LinkedResources.Add("Resources\\nb-no.resources");
15.         }
16.     }
17.     // Invoke compilation.
18.     CompilerResults cr = provider.CompileAssemblyFromFile(cp, sourceFile);
19.     if(cr.Errors.Count > 0) { // if there were errors
20.         return false;
21.     } else { // compilation has been done sucessfully
22.         return true;
23.     }
24. }
```

Figura 11 – Utilização da API de compilação dinâmica no C#

Fonte: [MIC14]

Nesse exemplo, temos um método que recebe o arquivo fonte por parâmetro e invoca o compilador (linha 18) para gerar a classe em tempo de execução. Podemos ver

também, na linha 19, que é possível verificar programaticamente, da mesma forma que no Java, se a compilação ocorreu com sucesso ou se houve alguma falha.

2.6 META-TAGS

Uma meta-tag é um artifício comumente presente em linguagens de programação orientadas a objetos que permite ao desenvolvedor armazenar meta informações que podem ser utilizadas pelo próprio sistema em tempo de execução. Meta-tags podem ser utilizadas para marcar um determinado componente do sistema e podem ser geralmente mantidas em tempo de desenvolvimento (para marcar um componente como depreciado, por exemplo) ou em tempo de execução. Quando uma meta-tag é mantida em tempo de execução, ela pode ser capturada pelo compilador, que pode utilizá-la para tomar decisões com o sistema em pleno funcionamento. O compilador pode, por exemplo, utilizar uma meta-tag para identificar um componente que deve ser monitorado pelo sistema ou para efetuar algum controle específico de transação.

Em Java, por exemplo, meta-tags são chamadas de anotações (a*nnotations*) e são adicionadas no código fonte através da utilização do caractere “@”. Esses conceitos serão cobertos em detalhes na Seção 2.7.3 visto que anotações são amplamente utilizadas na implementação do *framework* JFault em Java. Outras linguagens, como o C#, também possuem meta-tags. Em C#, meta-tags são chamadas de atributos (a*ttributes*) e também podem ser mantidas em tempo de execução para serem acessadas pelo compilador. A Figura 12 mostra um exemplo de utilização de atributos em C#. O atributo “*DocumentationAttribute*”, criado entre as linhas 3 e 12, visa prover uma meta-tag que pode ser utilizada para documentar dados de criação de uma classe (incluindo autor, data de criação, número de revisão e documentação). O atributo é então utilizado para documentar a classe “*Attr*”, linhas 14 e 15. Nas linhas 17 a 20 podemos ver os atributos sendo lidos através de reflexão e impressos na console do sistema [ZUK06].

```
1.  using System;
2.  using System.Reflection;
3.  [AttributeUsage(AttributeTargets.Class)]
4.  public class DocumentationAttribute : System.Attribute {
5.      public string author;
6.      public string date;
7.      public double revision;
8.      public string docString;
9.      public DocumentationAttribute(string a, string d, double r, string s) {
10.         author = a; date = d; revision = r; docString = s;
11.     }
12. }
13. [Documentation("Michael Scott","July, 2008", 0.1, "Illustrates the use of attributes")]
14. public class Attr {
15.     public static void Main(string[] args) {
16.         System.Reflection.MemberInfo tp = typeof(Attr);
17.         object[] attrs = tp.GetCustomAttributes(typeof(DocumentationAttribute), false);
18.         DocumentationAttribute a = (DocumentationAttribute) attrs[0];
19.         Console.WriteLine("author: " + a.author + "date: " + a.date + “doc:”+a.docString)
20.     }
21. }
```

Figura 12 – Exemplo de utilização de meta-tags (Attributes) em C#

Fonte: [ZUK06]

2.7 JAVA STANDARD EDITION

O Java Standard Edition é a especificação padrão da linguagem Java e da máquina virtual Java. A API inclui desde suporte básico à programação, como funções matemáticas, coleções para armazenamento e manipulação de objetos e comandos para controle de fluxo e laços, até recursos mais avançados como suporte para criação de interfaces (AWT e Swing), conectividade a banco de dados (JDBC [JDB14]) e comunicação distribuída (RMI, CORBA/IDL e Sockets). A maioria dos pacotes necessários para a construção de uma aplicação Java está disponível no JSE [SAM11].

O Java Standard Edition também pode ser divido no JDK (Java Standard Kit), que possui a API e todo o suporte necessário para o desenvolvimento de uma aplicação Java (como o compilador Java - Javac), e o JRE (Java Runtime Environment), utilizado quando

somente o suporte de execução do *Java* é necessário. A Figura 13 dá uma visão mais ampla e completa do Java Standard Edition:

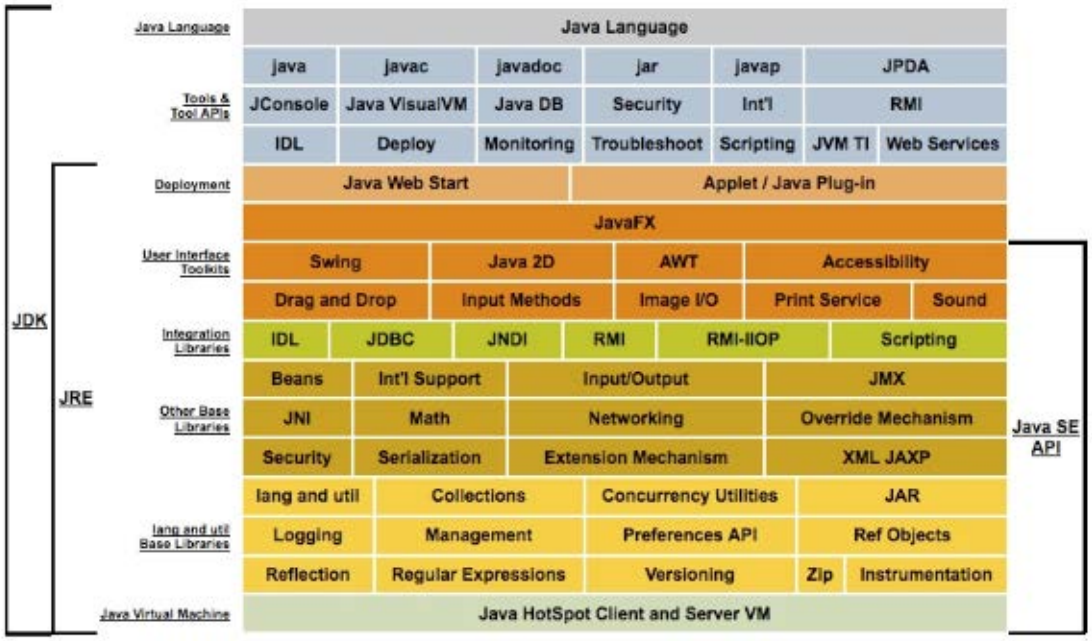

Figura 13 – Plataforma *Java Standard Edition*

Fonte: [JSE14]

Nas seções subsequentes, abordaremos algumas dessas tecnologias, mas especificamente as que serão mencionadas no capítulo que apresenta a proposta do *framework* JFault.

2.7.1 Java AWT/Swing/SWT

O Abstract Window Toolkit *(*AWT) é um conjunto de interfaces de programação de aplicativos (APIs) utilizadas no Java para criar interfaces gráficas, também conhecidas como GUIs *(Graphical User Interfaces).* Presente desde as versões iniciais do Java, a API fornece apoio para criação de componentes gráficos com botões, barras de rolagem e janelas, assim como provê todo suporte necessário para tratamento e captura de eventos de tela (movimento de *mouse*, fechamento de janelas, etc). Uma particularidade do AWT é que ele fornece um nível de abstração sobre a interface de usuário nativa do sistema operacional, ou seja, se uma aplicação Java possuir uma caixa de seleção AWT, por exemplo, essa caixa ficaria no padrão Windows, quando a aplicação for executada no

Windows, e no padrão Apple Macintosh, quando a mesma for rodada no Mac. No entanto, alguns desenvolvedores não utilizam o AWT pois preferem que suas aplicações fiquem idênticas em todas as plataformas.

O Swing, criado a partir da versão 1.2 do JSE, ultrapassou o AWT e passou a ser mais amplamente utilizado para criação de aplicações *desktop* em Java por oferecer um conjunto mais rico de componentes de interfaces e também por fornecer apoio à criação de componentes próprios de tela com a utilização conjunta do Java 2D). O Swing também fornece a vantagem de dar opção ao desenvolvedor de criar interfaces multi-plataformas, que não variam de acordo com o sistema operacional, ou seja, a interface gráfica vista pelo usuário no Windows será a mesma vista no Linux ou no Mac [AWT14].

O SWT (Standard Widget Toolkit) é um projeto Eclipse [ECL14] que define um API portável de componentes de interface para várias plataformas. Por utilizar componentes nativos, assim como o AWT, a interface também se adapta ao sistema operacional. O SWT tornou-se bastante utilizado em aplicações Java visto que possui um conjunto mais amplo de componentes de interface quando comparado ao Swing e AWT. O SWT não é nativo do Java e precisa de bibliotecas auxiliares para ser utilizado.

É possível também combinar componentes AWT, Swing e SWT em uma mesma aplicação, contudo, isso geralmente leva a efeitos indesejáveis de ordenação e *layout* de componentes, mesmo que estes efeitos tenham sido minimizados nas últimas implementações da versão Java. Componentes gráficos AWT, Swing e SWT também podem ser utilizados em Applets[3] e expostos ao usuário em um *browser*.

2.7.2 Java RMI (Remote Method Invocation)

O principal meio de comunicação e troca de mensagens entre objetos no Java é através de invocação de métodos. Os mecanismos de instanciação e chamada de métodos são simples quando todos os objetos estão sendo executados dentro da mesma máquina virtual Java, compartilhando a mesma memória. Contudo, quando objetos necessitam trocar mensagens entre diferentes máquinas virtuais, algum mecanismo que permita a comunicação remota deve ser utilizado. Tratando-se de JSE, existem alguns

[3] Um *applet* é um pequeno aplicativo escrito em Java disponibilizado aos usuários normalmente através de um serviço ou aplicação em uma página WEB.

mecanismos que podem ser utilizados para tal: comunicação através de Sockets [SOC13] e CORBA [BOL01] são alguns exemplos.

A tecnologia RMI (invocação remota de métodos) provê um melhor suporte para comunicação remota em Java, tendo em vista a relativa simplicidade de implementação (comparado com CORBA) e o fato de que trabalha com invocação remota de métodos de uma forma orientada a objetos [NIE05], como se os objetos fossem locais (diferentemente de todo o *overhead* necessário para codificar e decodificar mensages na comunicação remota quando trabalhamos diretamente com Sockets).

Aplicações RMI frequentemente são compostas de dois programas separados, um cliente e um servidor. Um programa servidor típico possui objetos remotos, que são registrados em um servidor de nomes chamado RMIRegistry, deixando-os assim, expostos e acessíveis aos clientes. O servidor “chama” o registro para associar um nome com um objeto remoto. Os clientes, por sua vez, obtém referências remotas para esses objetos utilizando o registro e, então, invocam os métodos do objeto remoto no servidor.

O RMI utiliza dois objetos principais como padrão para a comunicação com objetos remotos: *Stubs* e *Skeletons*. O *Stub* é uma representação local de um objeto remoto no lado cliente. *Stubs* são utilizados para esconder toda a comunicação em nível de rede e pelo tratamento da chamada remota, a fim de apresentar um mecanismo de invocação simples para o cliente. Quando o cliente chama um método através do *Stub*, este é responsável por toda a serialização dos objetos e a chamada do método remoto, assim como a deserialização da resposta. Já o *Skeleton* fica no lado do servidor. Cada objeto remoto possui um *Skeleton*. Este objeto é responsável pelo tratamento da chamada no lado servidor (como deserialização da chamada, serialização da resposta e direcionamento da chamada para o objeto real que irá processá-la) [RMI11]. Para um melhor entendimento do processo de chamada de um método remoto analisamos uma sequência hipotética de chamada abaixo:

1) após localizar o objeto remoto no registro, o cliente inicia uma conexão através do *Stub* com a JVM que contém o objeto;
2) o cliente, através do Stub faz a serialização dos objetos (processo conhecido como *Marshalling*) e passa os objetos como parâmetro para a JVM remota através da chamada ao método remoto;

3) o servidor recebe a chamada através do Skeleton, que deserializa os objetos (processo conhecido como *Unmarshalling*) e roteia a mesma para o objeto que irá processá-la;
4) o cliente aguarda o resultado da invocação de método;
5) o servidor conclui a chamada e envia a resposta através do Skeleton, que faz a serialização da resposta a ser enviada para o cliente;
6) o Stub recebe a resposta do servidor e deserializa-a para que, então, o cliente possa verificar o valor de retorno (ou exceção).

É importante observar que toda complexidade da comunicação remota, como processos de serialização e deserialização, é abstraída do cliente. Ou seja, do ponto de vista do cliente, o método está sendo chamado normalmente, como se fosse um método de um objeto local, como exemplificado na Figura 14.

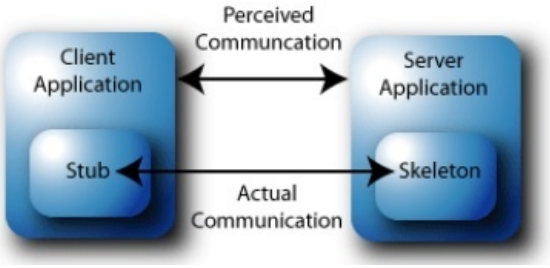

Figura 14– Comunicação RMI

Fonte: [JRMI11]

2.7.3 Java Annotations

Segundo Mendes [MEN11], anotações em Java são marcações (meta-tags*)* que iniciam pelo símbolo “@” e podem ser adicionadas ao código fonte para armazenar informações. Anotações podem marcar pacotes, classes, interfaces, construtores, métodos, atributos, parâmetros de métodos e até mesmo variáveis criadas dentro de métodos. Uma anotação pode ser anexada a um elemento Java no código para propósitos de documentação ou para que o compilador possa identificar o elemento e tomar decisões em tempo de execução. Anotações podem ser utilizada para, por exemplo, documentar que um determinado elemento se tornou depreciado em uma versão da API. De fato, a anotação “*@Deprecated*” da própria API do Java pode ser utilizada para tal fim. Além de

informar ao programador que determinado elemento está depreciado, essa anotação também é utilizada pelo compilador para gerar avisos (*warnings*) quando um elemento depreciado é referenciado.

Anotações podem também ser criadas. A criação de um novo tipo de anotação utiliza o símbolo “@” seguido da palavra reservada “interface” e o nome da anotação. A Figura 15 apresenta um exemplo de criação de uma anotação utilizada para documentar uma classe. O atributo “*@Retention*”, na linha 3, quando utilizado com a opção “*RetentionPolicy.RUNTIME*”, indica que a anotação deve ser retida em tempo de execução, isto é, deve ser mantida pelo compilador e acessível durante a execução do programa. Nas linhas 8 a 11, é possível verificar a anotação sendo utilizada para documentação da classe “*Annotate*”; as linhas 14 a 17 mostram as informações armazenadas na anotação “Documentation” sendo lidas em tempo de execução, através da API de reflexão do Java.

```
1.  import static java.lang.System.out;
2.  import java.lang.annotation.*;
3.  @Retention(RetentionPolicy.RUNTIME)
4.  @interface Documentation{
5.      String author(); String date();
6.      double revision(); String docString();
7.  }
8.  @Documentation(
9.      author = "Michael Scott", date = "July, 2008",
10.     revision = 0.1, docString = "Illustrates the use of annotations"
11. )
12. public class Annotate {
13.     public static void main(String[] args) {
14.         Class<Annotate> c = Annotate.class;
15.         Documentation a = c.getAnnotation(Documentation.class);
16.         out.println("author: " + a.author()); out.println("date: " + a.date());
17.         out.println("revision: " + a.revision()); out.println("docString: " + a.docString());
18.     }
19. }
```

Figura 15 – Exemplo de utilização de meta-tags (annotations) em Java

Fonte: [ZUK06]

2.8 ARQUITETURA CLIENTE SERVIDOR

A arquitetura cliente-servidor pode ser dividida em duas camadas e três camadas[4]. No modelo duas camadas a interface do sistema está no lado cliente, assim como a implementação da lógica de negócio; somente o meio persistente - geralmente um banco de dados - atua no lado servidor. Esse modelo reduz a flexibilidade do sistema visto que alterações na interface ou lógica de negócio impactam o cliente que precisará de uma atualização do programa. Esse modelo também é menos escalável, visto que processos de negócio, mesmo que pesados (que precisam de grandes quantidades de processamento ou memória), rodam na própria máquina do cliente. Esses problemas foram endereçados removendo a lógica de negócio do lado cliente, gerando o modelo arquitetural em três camadas.

Segundo Lobo [LOB09] a arquitetura em três camadas é a mais utilizada atualmente no desenvolvimento de *software*. Nesse modelo arquitetural, uma evolução da arquitetura padrão cliente-servidor em duas camadas, a camada de negócio é separada da camada cliente e da camada de banco de dados. A implementação de rotinas de negócio, ainda segundo o autor, na camada cliente ou na camada de banco de dados pode acarretar nos seguintes problemas:

1) a segurança do sistema fica comprometida com as regras de negócio na camada cliente;
2) a manutenção e alteração das regras de negócio são mais complexas quando estas estão diretamente implementadas no cliente ou banco de dados. No caso de implementação da lógica no cliente, este precisa ser alterado mesmo no caso de modificações feitas puramente na lógica de negócio;
3) a implementação das regras de negócio diretamente no cliente ou no banco de dados em forma de rotinas pode trazer problemas de desempenho.

Além de colaborar na resolução dos problemas apontados acima, deixando a arquitetura do sistema mais segura e as camadas do cliente e banco de dados mais leves e independentes, a arquitetura em três camadas permite maior desacoplamento entre os

[4] Existem ainda outras variações da arquitetura clientes servidor, como modelo em quatro e em "n" camadas [MAN08], que não serão discutidas nesse trabalho.

módulos do sistema, tornando a arquitetura como um todo mais gerenciável, flexível e mais fácil de ser escalada. No que se refere à questão da flexibilidade, um mesmo conjunto de serviços de negócio pode ser acessado por um cliente rodando em um computador *desktop* e um dispositivo móvel, por exemplo, contanto que ambos suportem o protocolo de comunicação utilizado para exposição dos serviços.

Com a utilização das arquiteturas em três camadas, surgiu o termo *thin client* (cliente leve), uma referência à retirada de código de negócio “pesado” dos clientes, deixando a aplicação cliente mais “leve”. A Figura 16 apresenta graficamente um modelo arquitetural de três camadas. A primeira, representa a camada cliente da aplicação, na segunda camada, estão os serviços de negócio (ou a lógica de negócio da aplicação), enquanto a terceira possui os recursos de persistência, gerenciamento e armazenamendo de dados, geralmente um Sistema Gerenciador de Banco de Dados (SGBD).

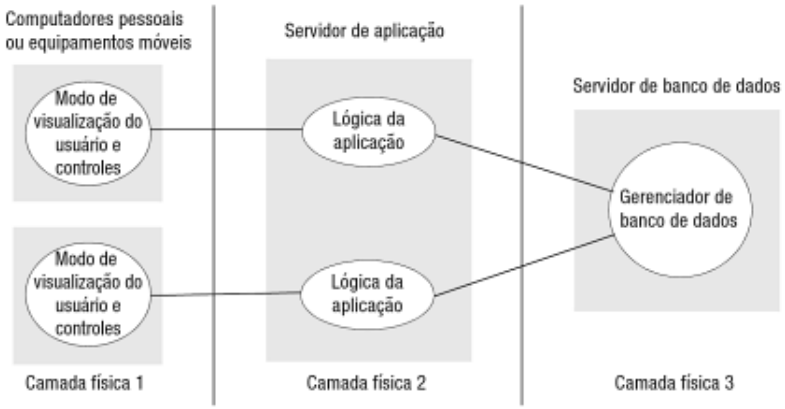

Figura 16– Arquitetura de três camadas

Fonte: [HPO13]

3 *FRAMEWORK JFAULT*

Nesse capítulo é apresentado o *framework* JFault. Primeiramente o *framework* é descrito em nível conceitual, de forma independente de implementação, inclusive seu funcionamento, seus principais componentes e o processo de tolerância a falhas efetuado dinamicamente pelos objetos de meta-nível. A seguir, será demonstrada a implementação do *framework* utilizando a linguagem Java.

3.1 NÍVEL CONCEITUAL

O objetivo do *framework* proposto – JFault – é prover mecanismos não intrusivos que criem dinamicamente objetos de meta-nível em tempo de compilação e execução do sistema, fornecendo suporte para construção de serviços distribuídos e tolerantes a falhas. Inicialmente, o suporte do framework é proposto a nível de falhas de Colapso e Tempo, podendo ser posteriormente estendido em trabalhos futuros.

O objetivo e a principal vantagem da proposta é que o desenvolvedor possa focar na implementação dos objetos que tratam da parte funcional da aplicação, como por exemplo, na criação das interfaces do sistema e da lógica de negócio. Todos os objetos de meta-nível necessários para expor os serviços tolerantes a falhas remotamente, para que esses possam ser distribuídos na arquitetura (caso de aplicações em três camadas), serão criados pelo *framework* podendo ser facilmente adaptados à aplicação com um esforço mínimo de desenvolvimento. Além de prover uma separação clara e em níveis entre a implementação dos requisitos funcionais e a implementação dos mecanismos tolerantes a falhas, o *framework* ainda se propõe a auxiliar na escalabilidade do sistema. Visto que os objetos de meta-nível que toleram falhas nos serviços são expostos remotamente pelo *framework*, esses podem ser utilizados de forma distribuída na arquitetura da aplicação, provendo suporte, assim, para escalabilidade horizontal. Para a identificação dos serviços que devem ser remotamente expostos e tolerantes a falhas, anotações no código devem ser feitas pelo próprio desenvolvedor da aplicação. Os objetos de meta-nível são criados e compilados dinamicamente, através do uso de compilação dinâmica de acesso programático, com a mesma estrutura dos objetos de serviço, capturada com a utilização de reflexão estrutural.

O *framework* JFault cria automaticamente, utilizando os conceitos de reflexão apresentados, dois tipos de componentes que são chaves para o processo de tolerância

a falhas e exposição remota dos serviços: os Stubs tolerantes a falhas e os Proxies. Esses são os objetos de meta-nível criados pelo *framework*. Ambos serão criados e compilados dinamicamente, sem intervenção, implementação ou qualquer esforço do desenvolvedor da aplicação.

3.1.1 Objetos de meta-nível: Stub e Proxy

O Stub é o componente responsável pela comunicação do cliente com o servidor, ou seja, realiza a comunicação com os serviços remotos. O Stub também é o objeto que possui a lógica principal de implementação dos mecanismos tolerantes a falhas selecionados através das meta-tags “JFaultStub” e as meta-tags específicas de cada tipo de falha, adicionadas nos serviços pelo desenvolvedor. Os Stubs no JFault são criados por uma ferramenta utilitária disponibilizada pelo próprio *framework* – o JFault Manager.

O “JFault Manager” obtém as informações necessárias da estrutura das classes de serviço através de reflexão e as demais configurações e técnicas de tolerância a falhas que deverão ser suportadas através das meta-tags adicionadas nas classes. Com a estrutura dos serviços e os tipos de falhas que devem ser toleradas, o *framework* cria o código fonte necessário para construir os mecanismos tolerantes a falhas, responsáveis pela localização remota e chamada dos serviços. Esses componentes são posteriormente compilados pelo *framework,* através da utilização de compilação dinâmica de acesso programático. Os Stubs são então empacotados e disponibilizados pelo *framework* para serem utilizados na aplicação. O desenvolvedor da aplicação deve então adicionar o pacote de Stubs (biblioteca) como uma dependência de seu aplicação e então adaptar o código cliente para utilizá-lo, procedimento que será coberto em mais detalhes nos capítulos que seguem. Embora existam alguns procedimentos manuais para geração e adaptação dos Stubs no sistema, esses só são feitos quando a aplicação é criada, quando um método novo em algum serviço é adicionado ou quando alguma assinatura de método de serviço for alterada. Ou seja, a geração dos Stubs pode ocorrer com mais frequência no início do projeto, mas geralmente tende a ocorrer pouco, uma vez que a estrutura dos serviços a serem disponibilizados pela aplicação é estabilizada.

O Stub é o componente utilizado pelo *framework* para detecção de erros, que é feita de forma concorrente *(Concurrent Detection).* Encontrado o erro, o mesmo é automaticamente compensado por outro serviço e a falhas então é isolada. Outro ponto

importante a ser mencionado é que o Stub é independente da aplicação (serviços no servidor), visto que o é localizado no cliente da aplicação. Assim, o Stub não representa um ponto único de falha quando tratamos de falhas de Colapso e Tempo, pois o mesmo somente virá a falhar se o cliente em si falhar, devido a um problema na própria máquina física do cliente, por exemplo.

O Stub utiliza todos os serviços disponíveis, mantendo uma referência remota para os mesmos internamente (*pool* de serviços). A utilização de diversos serviços idênticos, além de possibilitar a utilização de técnicas de tolerância a falhas por compensação (por meio de componentes redundantes), também auxilia na escalabilidade do sistema visto que várias instâncias do mesmo serviço atuam paralelamente atendendo e processando requisições. Um ponto importante a ser mencionado é que o framework não fornece suporte para criação de serviços que mantenham estado. Assume-se aqui que o sistema que utiliza o *framework* possui algum mecanismo responsável pela persistência dos dados, caso esse requisito seja necessário. Visto que as réplicas (de serviços) não mantêm estado, não é necessário nenhum mecanismo de sincronismo entre elas, mesmo considerando que todas trabalham de forma ativa, recebendo requisições dos clientes.

Os componentes de Proxy criados pelo JFault recebem as requisições dos clientes (mais especificamente dos próprios Stubs) no lado servidor. Os componentes de Proxy são localizados pelo *framework* através da meta-tag “JFaultRemote” e são sempre criados baseados nos próprios serviços - com a mesma estrutura dos serviços - de forma automática, quando o *framework* é iniciado, logo, adaptam-se automaticamente a qualquer alteração feita na implementação dos serviços. Não existe esforço de desenvolvimento ou qualquer esforço de adaptação do sistema para a utilização dos objetos de Proxy: o *framework* é responsável por sua criação e sua exposição para acesso remoto dinamicamente.

Com a utilização de reflexão estrutural, os Proxies são gerados pelo *framework* como uma imagem idêntica do próprio serviço. O *framework* também é responsável por toda implementação dos protocolos utilizados para expor o Proxy para acesso remoto no lado do servidor. Após a geração do código fonte dos Proxies, o *framework* compila os mesmos com a utilização de compilação dinâmica, carregando os objetos recém compilados em memória, através das técnicas de reflexão vistas no Capítulo 2.6, para que possam ser acessados pelo sistema.

Na prática, quando o *framework* é iniciado, os seguintes procedimentos são executados para a criação dos objetos de Proxy:

1) o *framework* inicia analisando todos os serviços da aplicação que estão sendo executados no servidor;
2) para cada serviço marcado com a meta-tag “JFaultRemote”:
    a. utilizando reflexão estrutural o *framework* captura toda a estrutura do serviço (os atributos, assinaturas de métodos, parâmetros, etc) e cria o código fonte do objeto Proxy com toda a implementação de protocolos necessária para expor esse objeto para ser acessado remotamente; a implementação dependerá do protocolo a ser utilizado; alguns exemplos de protocolos que podem ser utilizados são o RMI e o HTTP;
    b. o *framework* compila o código fonte em tempo de execução, utilizando compilação dinâmica de acesso programático, deixando-o disponível dinamicamente para ser utilizado no servidor através de reflexão;
    c. o serviço é instanciado e registrado no servidor com seu caminho de localização (nome remoto e a porta de acesso) configurado na própria meta-tag.

A Figura 17 demonstra graficamente o escopo de atuação do *framework* e a posição do Stub tolerante a falhas e do Proxy. O Stub recebe a requisição do cliente (interface) e, através de algum protocolo de comunicação, envia a requisição para o Proxy que a repassa para o serviço. As figuras em laranja (no meta-nível) representam os objetos criados pelo *framework*, que utiliza reflexão estrutural sobre o serviço da aplicação (representado em preto no nível base) para capturar sua estrutura e compilação dinâmica para compilá-los e deixá-los disponíveis para serem utilizados. Como pode ser visto na figura, o *framework* não atua na interface da aplicação, que deve ser adaptada pelo desenvolvedor para que use o Stub tolerante a falhas para comunicação com o servidor. É importante mencionar que cada serviço possui seu próprio Stub e Proxy, ou seja, se a aplicação possuir 15 serviços, 15 Stubs e 15 Proxies serão gerados pelo *framework*.

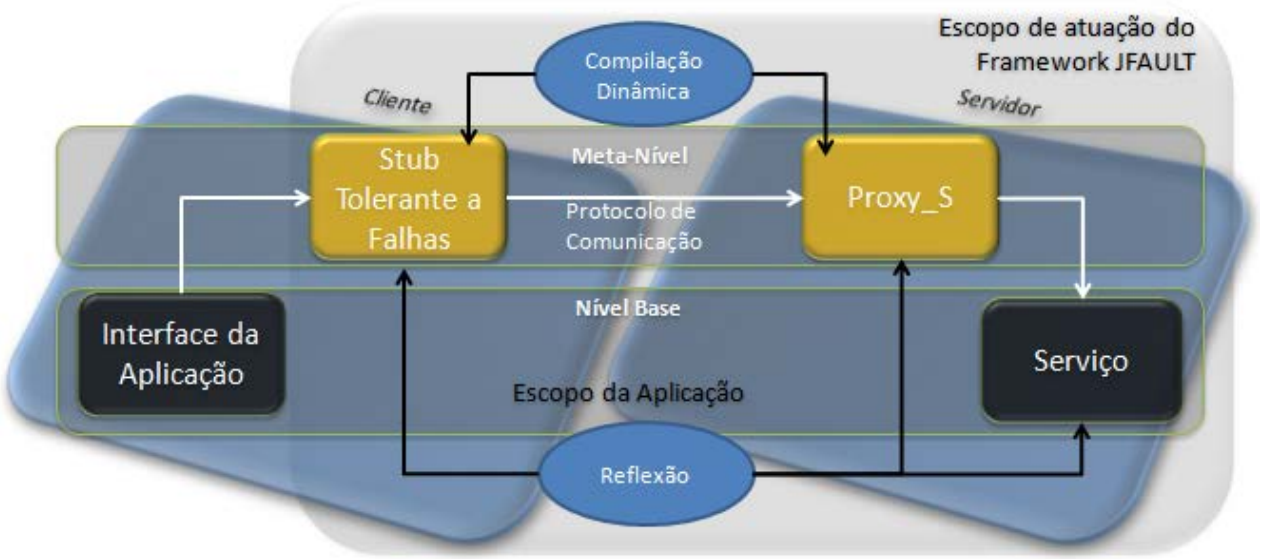

Figura 17 – Objetos de meta-nível: Stubs e Proxies do JFault

3.1.2 Processo de Tolerância a Falhas do JFault

Descreveremos nessa seção como o JFault realiza o tratamento de falhas nas aplicações que utilizarem o *framework*. O *framework* conta com o suporte para tratamento de dois níveis de falhas: Colapso (Crash) e Tempo. Um ponto importante a ser mencionado é que o *framework* parte do princípio que as chamadas aos serviços da aplicação são idempotentes, ou seja, múltiplas requisições ao mesmo recurso usando um método específico devem possuir o mesmo resultado do que uma requisição apenas.

3.1.2.1 Processo de Tolerância a Falhas de Colapso

No *framework,* todo processo de tolerância a falhas de Colapso é executado pelo objeto de meta-nível Stub. O desenvolvedor da aplicação deve usar a meta-tag "JFaultCrashFT" para indicar que um determinado serviço deve suportar falhas de Colapso, indicando, também, o algoritmo a ser utilizado pelo *framework*. Cada Stub possui um conjunto de serviços que podem ser acessados (*pool*) e uma tabela de monitoração de serviços que indica se o serviço está funcional através de requisições de *heartbeat*[5].

[5] *Heartbeat* é um sinal periódico gerado por *hardware* ou *software* para indicar o funcionamento normal ou para sincronizar partes de um sistema.

As requisições de *heartbeat* são feitas pelo Stub ao Proxy para cada serviço existente na aplicação. No lado do servidor, quando o Proxy é criado na inicialização do *framework*, um método de *heartbeat* é injetado na classe. Para cada serviço, o método de *heartbeat* é acessado por um processo paralelo (*thread)* do Stub com determinada frequência, para verificar se o serviço está responsivo. Se uma requisição falhar, o Stub altera o estado do serviço na tabela de monitoração indicando que o serviço colapsou, removendo-o do conjunto de serviços (*pool*), e não mais o utilizando até que fique disponível novamente. A *thread* do Stub continua tentando acessar o método de *heartbeat* dos serviços que colapsaram em intervalos maiores até que o serviço se restabeleça e possa ser utilizado novamente. Quando o serviço responde à requisição de *heartbeat*, o Stub reinsere o mesmo no conjunto de serviços e atualiza seu estado na tabela de serviços indicando que o serviço está novamente disponível. Essa técnica será utilizada pelo *framework* para tentar garantir, o máximo possível, que o cliente (Stub) sempre acesse serviços que estejam responsivos.

Os tempos de resposta de cada método dos serviços também são armazenados na tabela de monitoração e controle dos serviços (isso será coberto em mais detalhes quando o processo de tolerância a falhas de Tempo for abordado). Além de tentar garantir um conjunto de serviços (*pool*) responsivos, o Stub também controlará a chamada aos métodos de serviço. Se o cliente chamar um método através do Stub e o serviço colapsar, o Stub automaticamente redirecionará a chamada para outro serviço, mascarando a falha e tornando-a transparente para o cliente.

Para falhas de Colapso, existem dois algoritmos que podem ser selecionados pelo desenvolvedor da aplicação através da meta-tag “JFaultCrashFT”:

1) LB_ROUND_ROBIN: baseado no algoritmo Round-Robin [KAL09], onde as requisições são feitas de forma uniforme entre os serviços disponíveis. Esse algoritmo atua também na escalabilidade do sistema, visto que distribui o fluxo de requisições de forma homogênea entre todos os serviços;
2) LB_ROUND_ROBIN_WEIGHTED: baseado no algoritmo Round-Robin-Weighted [KAL09], é uma versão aprimorada do algoritmo “Round Robin”. Utilizando-se esse algoritmo é possível definir um “peso” para cada servidor. Considerando dois servidores (A e B), é possível definir, por exemplo, que 80% das requisições devem ser enviadas ao servidor A e 20% ao servidor B. Esse

> algoritmo é útil quando a arquitetura da aplicação possui servidores com diferentes capacidades - mais memória ou processadores por exemplo. Se um dos servidores apresentar uma falha de Colapso, o percentual de requisições que estavam sendo atendidas pelo servidor é distribuído uniformemente entre os outros servidores disponíveis.

A Figura 18 representa de forma gráfica as mensagens de *heartbeat* e a correspondente tabela de monitoração de serviços (vamos omitir por enquanto a informação de tempo de resposta das operações dos serviços). No caso apresentado na figura, se o algoritmo de escolha do Stub for baseado em “LB_ROUND_ROBIN”, todos os serviços estariam sendo chamados intercaladamente de forma uniforme, visto que todos estão operacionais. O algoritmo “LB_ROUND_ROBIN_WEIGHTED” poderia também ser escolhido para rotear as requisições aos servidores utilizando um critério de peso, por exemplo, 50% das requisições direcionadas ao “Servidor 1”, e os outros 50% divididas entre o “Servidor 2” e “Servidor 3”. No caso de nenhum serviço estar disponível, em nenhum servidor, o *framework* ficará impossibilitado de mascarar a falha, nesse caso uma exceção do tipo “JF_CRASH_EXCEPTION” é enviada ao cliente pelo Stub. Se todos os serviços falharem, o cliente deve tratar a exceção conforme os requisitos da aplicação, no geral, exibindo uma mensagem de erro para o usuário. A terceira camada arquitetural, de persistência, foi omitida do diagrama com o propósito de simplificação.

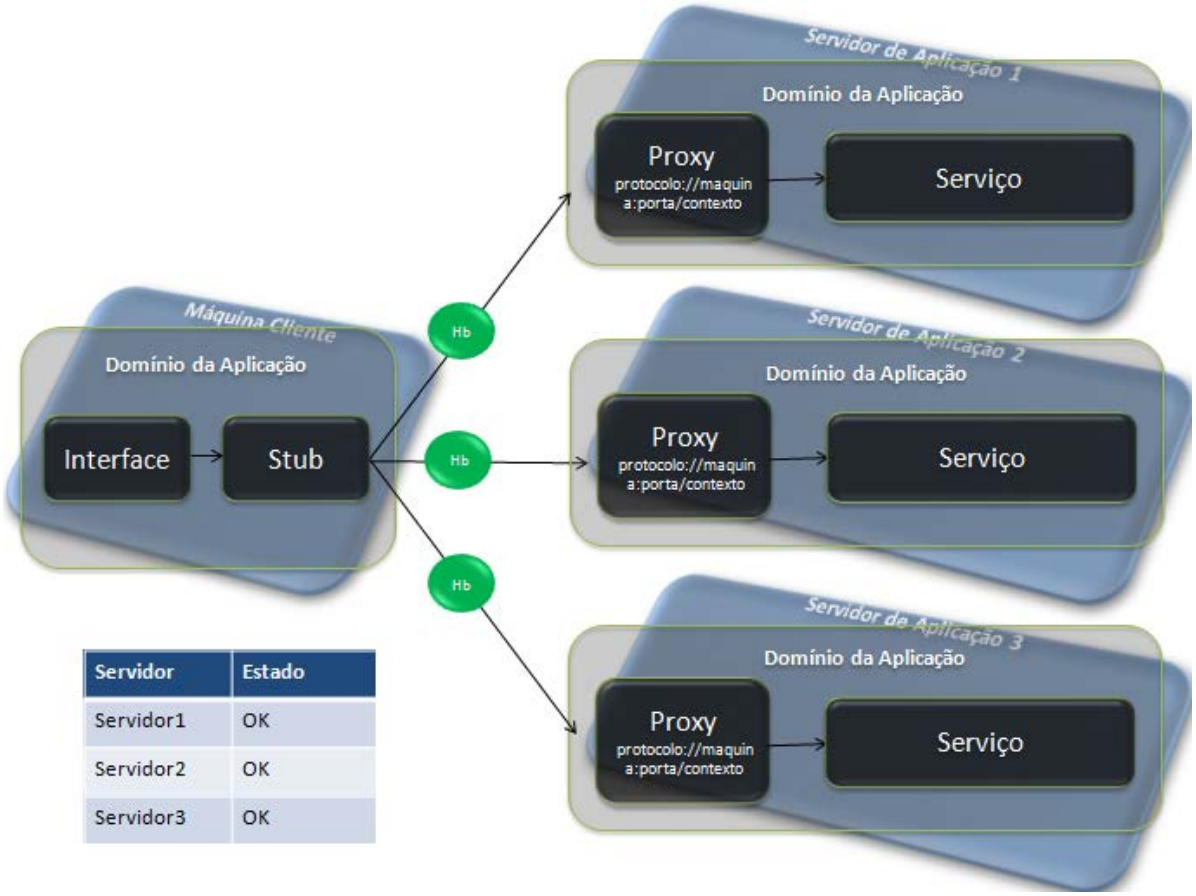

Figura 18 – Processo de tolerância a falhas do tipo Colapso

3.1.2.2 Processo de Tolerância a Falhas de Tempo

Falhas de Tempo são difíceis de ser toleradas, pois quando ocorrem, o tempo estipulado pelo cliente para obtenção da resposta já passou e o simples reenvio da requisição não será suficiente para se tolerar a falha. Para tolerar falhas de Tempo foi adotado um algoritmo que trabalha na sua prevenção[6], similar ao proposto por Krishnam et. al [KRI01] na *middleware* AQuA. A *middleware* AQuA é basedada em CORBA [BOL01] e utiliza os dados históricos dos tempos de resposta das últimas requisições efetuadas aos serviços em cada réplica para tentar prevenir que falhas de Tempo ocorram. Esses tempos são utilizados pela *middleware* posteriormente para tentar selecionar o conjunto de réplicas que possui a maior probabilidade de atender a requisição do cliente dentro do tempo estipulado nos requisitos da aplicação. Quando o conjunto de réplicas é definido, a

[6] Embora alguns autores (como Avizienis et al. em [AVI04]) afirmem que tolerância a falhas e prevenção de falhas são diferentes técnicas para dependabilidade manteremos os termos relacionados seguindo o mesmo modelo proposto no trabalho de Krishnam et. al em [KRI01]

requisição é enviada para todo o conjunto mas somente a primeira resposta é computada, as demais são descartadas. É importante observar que esse algoritmo também auxilia na tolerância a falhas de Colapso em si, visto que, caso uma réplica do serviço colapsar, o cliente ainda poderá obter a resposta de outras réplicas. A *middleware* visa prover suporte a serviços distribuídos, sem estado (*stateless*) e idempotentes, assim com o *framework* JFault. Este último utilizará um algoritmo similar, que chamaremos a partir de agora de "FT_PREVENTION". Dentro de cada Proxy o *framework* injeta um componente que armazena um histórico dos últimos tempos de resposta de cada operação do serviço. Ou seja, cada vez que o Stub efetuar uma operação solicitada pelo cliente, o Proxy captura o tempo de processamento total dessa requisição e armazena esse tempo internamente, em um componente compartilhado por todas as instâncias do mesmo serviço no lado servidor.

Com isso, cada Proxy passa a possuir uma tabela de informações que indica o tempo total de processamento de cada uma das operações acessíveis pelos clientes. Um exemplo dessa tabela de informações pode ser visto na Tabela 2, considerando que o JFault foi configurado para armazenar os últimos cinco tempos de resposta de um serviço chamado "FolhaDePagamento", que contém as operações "incluirOuAlterarDespesa(despesa)" e "fecharAFolha(periodo)".

Tabela 2 – Tempos de Resposta de Serviços do Proxy

Serviço FolhaDePagamento	
Operação	**Tempos de Resposta (Cinco ultimas requisições)**
incluirOuAlterarDespesa(despesa)	0.4 s 0.7 s 0.2 s 0.6 s 1.2s
fecharAFolha(periodo)	18.4 s 15.2 s 19.2 s 19.6 s 25.2s

Para que o serviço seja tolerante a falhas de Tempo ele deve ser marcado no código pelo desenvolvedor com a meta-tag “JFaultTimingFT” em nível geral (de classe), que deve conter também as seguintes informações:

1) ALGORITHM: “FT_PREVENTION”;
2) GENERAL_SLA: esse parâmetro é utilizado em nível geral para determinar o tempo de resposta do serviço para todas as operações; A meta-Tag “JFaultTimingFTSpecificSLA” pode ser utilizada em nível de operação (método) para determinar um tempo de resposta específico a uma operação;
3) REPLICA_QTD: esse parâmetro indica quantas réplicas devem ser selecionadas para atender a requisição do Stub; o número de réplicas precisa ser menor ou igual ao número total de réplicas gerenciadas pelo Stub;

Uma vez que os Proxies contêm as informações de resposta de cada operação dos serviços, o Stub precisa agora capturar essa informação, visto que ele é o componente que contém a inteligência de roteamento das chamadas dos clientes. O Stub captura essas informações através das chamadas de *hearbeat*, alimentando a mesma tabela de monitoramento utilizada pelo *framework* para tratamento de falhas de Colapso. A cada requisição de *hearbeat,* efetuada a cada réplica de serviço sendo executada nos diferentes servidores do sistema, o Proxy retorna um objeto que contém um histórico dos últimos tempos de reposta de cada operação. Dessa forma, o Stub conhece não somente quais serviços se encontram disponíveis, mas também obtém uma visão mais ampla de seus estados internos. Se um servidor apresentar um histórico de respostas maior do que outro, para o mesmo serviço, isso pode indicar que o mesmo esteja com certa sobrecarga (utilização muito alta de CPU ou memória, por exemplo).

A Tabela 3 apresenta uma visão da tabela de monitoração de serviços no Stub, baseada no mesmo serviço de “FolhaDePagamento” mencionado anteriormente, sendo disponibilizada por dois servidores, com a adição dos tempos de resposta das operações disponíveis nos Proxies.

Tabela 3 – Tabela de Monitoração de Serviços no Stub

Serviço FolhaDePagamento			
Servidor	**Status**	**Operação**	**Tempos de Resposta (Cinco últimas requisições)**
SERVIDOR 1	OK	incluirOuAlterarDespesa(despesa)	0.4 s 0.7 s 0.2 s 0.6 s 1.2s
		FecharAFolha(período)	18.4 s 15.2 s 19.2 s 19.6 s 25.2s
SERVIDOR 2	OK	incluirOuAlterarDespesa(despesa)	1.4 s 0.9 s 1.2 s 1.6 s 2.2s
		FecharAFolha(período)	31.4 s 23.2 s 33.2 s 32.6 s 29.2s
SERVIDOR 3	OK	incluirOuAlterarDespesa(despesa)	0.4 s 0.9 s 0.2 s 0.6 s 0.2s
		FecharAFolha(período)	10.4 s 12.2 s 11.2 s 15.6 s 11.2s

Tendo a tabela de monitoração com os tempos de resposta das operações em cada réplica de serviço, o Stub agora precisa somente selecionar os serviços que estarão mais aptos a responder dentro do SLA estipulado pelo cliente, ou seja, os serviços menos suscetíveis a apresentar uma falha de Tempo. A seleção das réplicas é feita da seguinte forma:

1) o Stub analisa os dados históricos de reposta das réplicas (armazenados na tabela de monitoração de serviços) e faz uma média simples dos tempos de

resposta em cada réplica; se alguma réplica ainda não possui informações de tempos de resposta, o Stub considera que o servidor não está sendo muito utilizado e define o tempo da média como 0 (zero);

2) com as médias dos tempos de todas as réplicas em uma lista, o Stub efetua uma ordenação das réplicas, deixando as que possuem os melhores (menores) tempos de resposta no topo da lista;
3) baseado no SLA definido e na quantidade de réplicas que devem ser utilizadas para tolerar possíveis falhas, o Stub seleciona as réplicas a partir do topo da lista.

A sequência de eventos abaixo pode ser vista para melhor entendimento do processo. Nesse exemplo, vamos considerar que uma requisição foi feita pelo cliente para o serviço “FolhaDePagamento”, mencionado acima, que o desenvolvedor configurou o serviço com SLA para todas as operações de 20 segundos e que o número de réplicas configurado para tolerar Falhas de Tempo é dois. Vamos assumir também que os tempos de cada operação (histórico de respostas anteriores) são os mesmos apresentados na Tabela 3:

1°) **Evento 1**: o cliente envia uma requisição ao serviço através do Stub para a operação “FecharAFolha(periodo)”; através das anotações configuradas no serviço, pelo desenvolvedor da aplicação, o Stub verifica que o SLA para essa operação é de 20 segundos e que o número de réplicas a serem escolhidas pelo algoritmo de tolerância a falhas de Tempo é dois.
2°) **Evento 2:** o Stub verifica a tabela de monitoração de serviços “FecharAFolha(periodo)” e faz uma média dos tempos das últimas requisições armazenadas em seu histórico. Baseado nos dados apresentados na Tabela 3, e tendo conhecimento que precisa utilizar duas réplicas para tolerar possíveis falhas, o Stub seleciona as réplicas do “Servidor 3” (melhor média - 12,2) e “Servidor 1” (média 19,52 segundos);
3°) **Evento 3**: o Stub envia a requisição aos dois servidores e aguarda a resposta;
4°) **Evento 4**: o Stub recebe a resposta do “Servidor 3” em 11,5 segundos. Visto que essa resposta não viola o *SLA* (de 20 segundos), ela é retornada pelo Stub ao cliente;
5°) **Evento 5:** a requisição enviada ao “Servidor 1” é cancelada pelo Stub;

.

Ainda tendo em vista a sequência de eventos apresentada acima, no caso de nenhum servidor responder no tempo estipulado, o *framework* retorna através do Stub uma exceção do tipo “JF_SLAEXCEPTION” ao cliente. A mesma exceção é retornada se o número de réplicas especificadas para tolerar falhas for maior do que o número total de réplicas gerenciadas pelo Stub. A Figura 19 pode ser vista para acompanhar em mais detalhes a sequência de eventos apresentada.

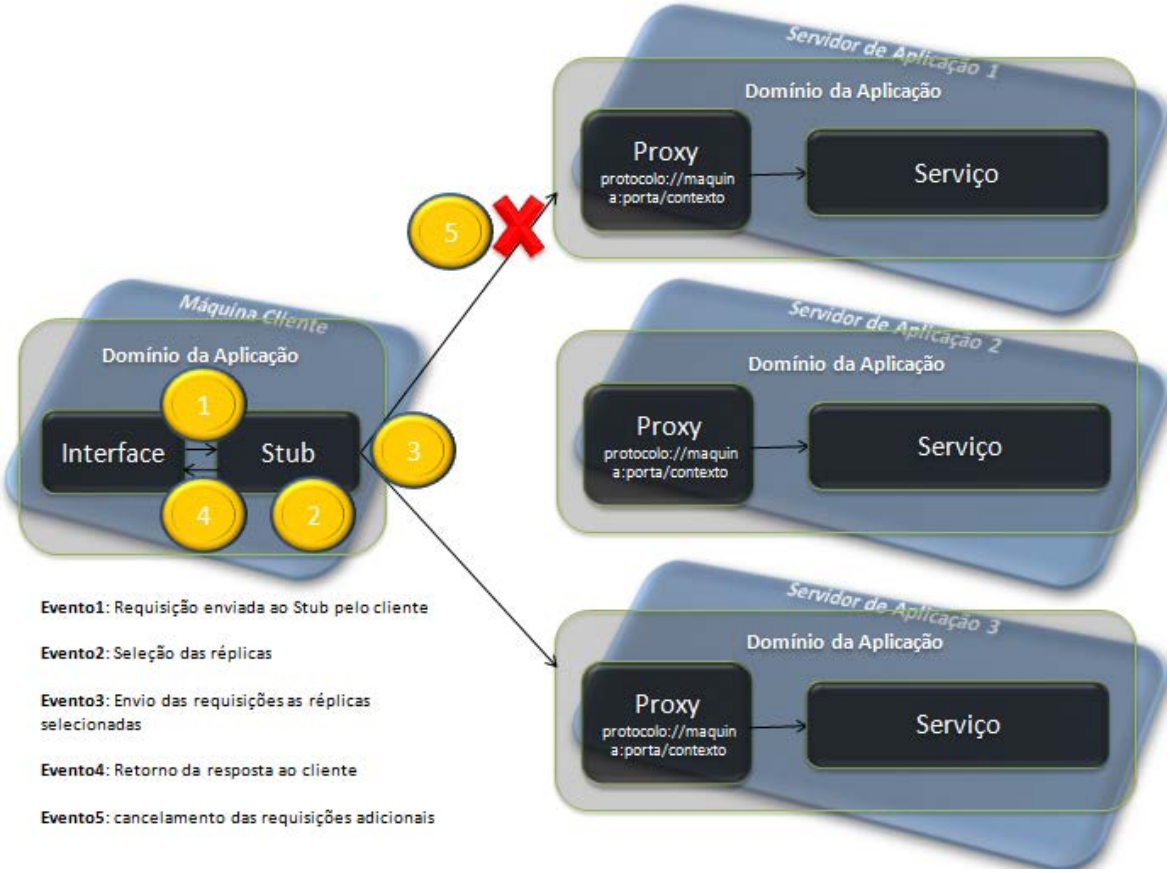

Figura 19 – Processo de tolerância a falhas de Tempo

3.2 IMPLEMENTAÇÃO DO FRAMEWORK EM JAVA

Na seção anterior, foi apresentado os principais componentes do *framework* JFault e como ele pode ser utilizado para construir dinamicamente mecanismos escaláveis e tolerantes a falhas no meta-nível. Nesse capítulo, o objetivo é apresentar a viabilidade da solução demonstrando um exemplo de implementação do *framework* na linguagem de programação Java. A linguagem Java foi escolhida para implementação por possuir os recursos necessários para a construção do *framework*: reflexão, compilação dinâmica de

acesso programático, meta-tags e protocolos para comunicação remota. A Tabela 4 demonstra as tecnologias Java que serão especificamente utilizadas na implementação do *framework*.

Tabela 4 – Recursos necessários de implementação e tecnologias Java envolvidas

Recurso	Tecnologia Java
Reflexão	Java Reflection API (java.lang.reflect)
Compilação Dinâmica de Acesso Programático	Java Compiler API (javax.tools.JavaCompiler, javax.tools.ToolProvider)
Meta-tags	Java Annotations (java.lang.annotation)
Protocolo de Comunicação Remota	RMI - Remote Method Invocation (java.rmi)

Contudo, como já visto anteriormente, esses recursos são comumente encontrados em outras linguagens de programação orientadas a objeto, consequentemente, o *framework* poderia ser implementado em qualquer outra linguagem que suporte tais recursos.

3.2.1 Diagrama de Classes

Nessa seção abordaremos o diagrama de classes do *framework* JFault. O diagrama será apresentado quebrado em pacotes para melhor visualização e organização dos componentes.

3.2.1.1 Pacote de meta-tags (com.jfault.annotations)

Esse pacote contém todas as interfaces de anotações (meta-Tags) que são utilizadas pelo *framework* para identificação dos serviços. As anotações “JfaultCrashFT” e “JFaultTimingFT” são utilizadas pelo *framework* para identificar os tipos de falhas que deverão ser toleradas em cada serviço. A anotação “JFaultTimingFTSpecificSLA ” é usada para determinar um tempo específico de resposta para uma operação – é a única anotação utilizada em nível de método. As anotações “JFaultStub” e “JFaultRemote” serão respectivamente utilizadas para criação dos componentes de Stub e Proxy do *framework*.

A Figura 20 mostra o pacote graficamente e a Tabela 5 mostra as anotações disponíveis, juntamente com as os parâmetros que podem ser utilizados em cada anotação e um a descrição detalhada de cada parâmetro.

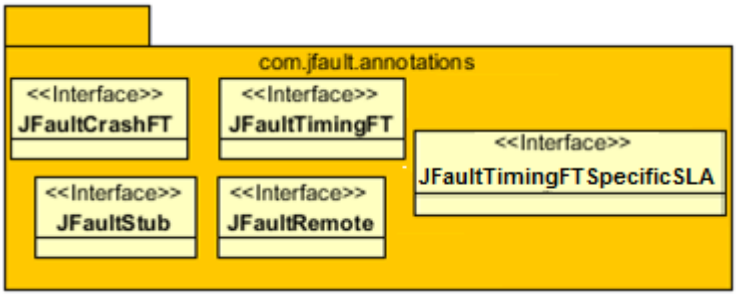

Figura 20 – Pacote de anotações do JFault

Tabela 5 – Anotações Suportadas pelo *JFault*

Anotação	Descrição	Parâmetros	Descrição dos Parâmetros
@JFaultRemote	Utilizada no serviço para expô-lo remotamente para utilização.	remoteReferenceName	Descreve o nome no registro *RMI* do servidor ao qual o serviço em questão será disponibilizado. Esse identificador será utilizado de forma transparente ao cliente pelo *Stub* gerado pelo *framework*.
		remoteReferencePort	Define o número da porta em que o serviço deve ser disponibilizado no registro *RMI* do servidor.
@JFaultStub	Utilizada pelo *framework* para criação do *Stub*, utilizado pelo cliente da aplicação para acesso remoto aos serviços.	remoteHosts	Define o endereço dos servidores (*hosts*) onde o serviço está sendo disponibilizado. Se o algoritmo de balanceamento de carga LB_ROUND_ROBIN_WIGHTED for escolhido a anotação “@” deve ser utilizada após o endereço do servidor, juntamente com o

				percentual de requisições que devem ser enviadas ao mesmo. Exemplo: @JFaultStub(remoteHosts={"localhost@80", "169.254.81.102@20"})
@JfaultCrashFT	Utilizada no serviço para torná-lo tolerante a falhas de colapso. Para que possa ser utilizada, é necessário que ao menos dois servidores de aplicação sejam incluídos no parâmetro remoteHosts da anotação JFaultStub.	algorithm	LB_ROUND_ROBIN	As requisições serão enviadas para todos os servidores de forma uniforme.
			LB_ROUND_ROBIN_WEIGHTED	As requisições serão enviadas aos servidores conforme um "peso", que pode ser definido para cada réplica. 70% das requisições para um servidor e 30% para outro, por exemplo.
@JFaultTimingFT	Utilizada para tolerar falhas de tempo no serviço. O uso dessa anotação sobrescreve o parâmetro de tempo limite de resposta padrão do *framework* para o serviço em questão.	algorithm	FT_PREVENTION	O algoritmo selecionará as réplicas baseado nos tempos anteriores de resposta. O atributo "REPLICA_QTD" define a quantidade de réplicas que será utilizada. O atributo "GENERAL_SLA" define o tempo máximo de resposta da primeira réplica.
		GENERAL_SLA	Long	
		REPLICA_QTD	Long	
@JFaultTimingFT MethodSLA	Utilizada para definir um tempo máximo de resposta do serviço em nível de operação.	METHOD_SLA	Long	O atributo "METHOD_SLA" define o tempo máximo de resposta à nível de operação. Seu uso sobrescreve o atributo "GENERAL_SLA", utilizado a nível de serviço (de classe).

3.2.1.3 Pacote de Exceções (com.jfault.exceptions)

Esse pacote contém todas as exceções que serão utilizadas pelo *framework*. A exceção "JFaultGenericException" é a classe base de todas as exceções e pode ocorrer em qualquer situação que não estiver coberta pelas outras, como erros na criação de uma interface ou falhas em registrar um serviço remoto. A "JFaultIOException" é utilizada em qualquer caso de erro relacionado à escrita e leitura de dados, como por exemplo, a escrita de uma classe Java no disco rígido. As exceções "JFaultCrashException", e

"JFaultSALViolatedException", são enviadas para o cliente pelo Stub respectivamente quando não existem mais serviços que possam atender as requisições e quando o tempo máximo de espera pelo retorno de um serviço foi excedido. A exceção "JFaultUnknowFailException" ocorre quando uma falha desconhecida e não tratada pelo *framework* ocorreu. A Figura 21 mostra o pacote "com.jfault.exceptions".

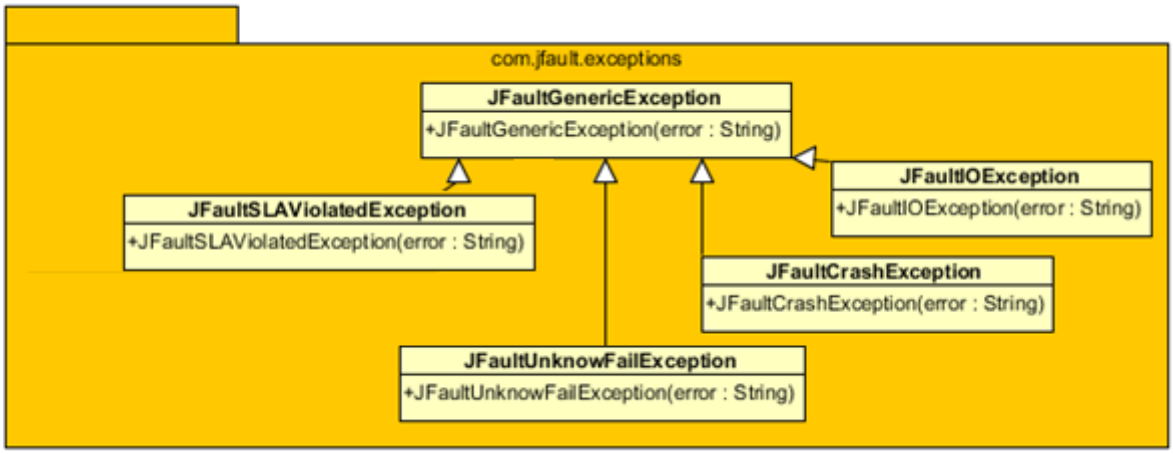

Figura 21 – Pacote de exceções do JFault

3.2.1.4 Classes de controle (com.jfault.framework)

Esse pacote, mostrado graficamente na Figura 22, contém as principais classes de controle do framework. A classe "JFaultLog" é utilizada para encapsular os logs do sistema, que podem ser tanto direcionados para a saída do console da aplicação (System.out) quanto para um arquivo. A classe "JFaultConstants" contém todas as constantes utilizadas no framework.

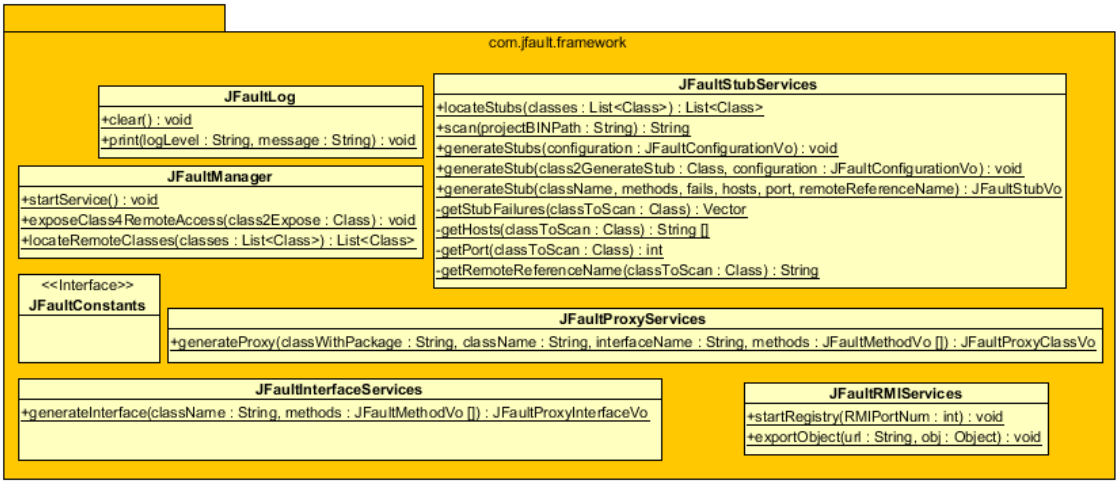

Figura 22 – Pacote de controle do *framework* JFault

A classe “JFaultInterfaceServices” encapsula toda a lógica de criação de interfaces dentro do *framework*. Conforme já abordado, os objetos de meta-nível Stub e Proxy são baseados nas mesmas assinaturas dos serviços que estão sendo expostos pelo *framework* para acesso remoto. No Java, quando utilizamos o protocolo RMI, faz-se necessário que as interfaces de acesso remoto sejam criadas, tanto no lado cliente (Stub) quanto no lado servidor (Proxy). Esse é o objetivo da classe “JFaultInterfaceServices”: o seu método principal “generateInterface()” recebe como parâmetro o nome da classe base e sua estrutura de métodos, extraídos com a utilização de reflexão, para gerar em tempo de execução a assinatura dos métodos das interfaces de Stub e Proxy. O método retorna um objeto do tipo “JFaultProxyInterfaceVo”, que contém todo o código necessário para geração das interfaces. O Apêndice A possui o código da classe para referência.

A classe “JFaultRMIServices” contém a implementação necessária de todos os serviços de exposição remota de objetos. Os dois métodos principais “startRegistry()” e “exportObjet()” são utilizados respectivamente para se iniciar um registro RMI e exportar um objeto para ser acessado remotamente. O Apêndice B possui o código da classe para referência.

A classe “JFaultProxyServices” é utilizada para gerar dinamicamente as classes dos objetos de meta-nível Proxies. O método principal da classe “generateProxy” recebe como parâmetro o nome da classe de serviço da aplicação, o nome da interface e a assinatura de todos os métodos. O retorno do método é um objeto do tipo “JFaultProxyVo” que contém toda informação necessária para que a classe do Proxy possa ser criada e compilada dinamicamente (Apêndice C).

A classe “JFaultStubServices” é utilizada pelo JFaultManager para gerar dinamicamente as classes dos objetos de meta-nível Stubs. Existem vários métodos nessa classe, os principais são: “JFaultGenerateStubs()” e “JFaultGenerateStub()”. O método “JFaultGenerateStubs()” é chamado pelo JFaultManager que envia a configuração do projeto que deve ser analisado para verificação dos Stubs presentes. Uma vez que o *framework* identificou todos os Stubs que devem ser gerados, o método “JFaultGenerateStub()” é chamado internamente na classe, para cada Stub, para que a classe e interface do mesmo possa ser construído, compilado e carregado dinamicamente. Como já visto, a estrutura da classe de serviços que gera o Stub é adquirida através de reflexão (linha 9) e compilada dinamicamente com a API de

compilação dinâmica de acesso programático do Java (linhas 15 e 38). O Apêndice D mostra a implementação do método “JFaultGenerateStub()” em detalhes.

A classe “JFaultManager” representa o serviço que inicia o *framework* no servidor. O processo de inicialização consiste em localizar os objetos de Proxy a serem criados dinamicamente, criar o código fonte necessário para sua construção, compilar o código e expor os objetos para serem acessados remotamente utilizando-se RMI. O Apêndice E apresenta a implementação do método principal da classe JFaultManager: “exposeClass4RemoteAccess()”.

3.2.1.5 Classes de Serviços de Compilação Dinâmica e Reflexão (com.jfault.jvm)

Esse pacote (Figura 23) contém o núcleo (*core*) de serviços do *framework*. A classe “JFaultJvmServices” encapsula toda lógica de compilação dinâmica, efetuada através da API de compilação dinâmica de acesso programático do Java (javax.tools). Atuando em combinação com um *class loader* específico do *framework* (“JFaultClassLoader”), é possível inclusive compilar recursos em tempo de execução com *class paths* adicionados dinamicamente (linha 31 da tabela 17). Isso se torna necessário visto que os Stubs possuem dependências com o projeto da aplicação em si, consequentemente, é necessário que o “JFaultManager” adicione o *class path* da aplicação nas dependências de compilação. A classe “JFaultReflectionServices” contém toda implementação dos serviços de reflexão do *framework*, permitindo inspecionar a estrutura de classes (método “getMethods()”) e criar novas instâncias de objetos (método “createInstance()”), ambos em tempo de execução. O Apêndice F apresenta a implementação dos principais métodos das classes.

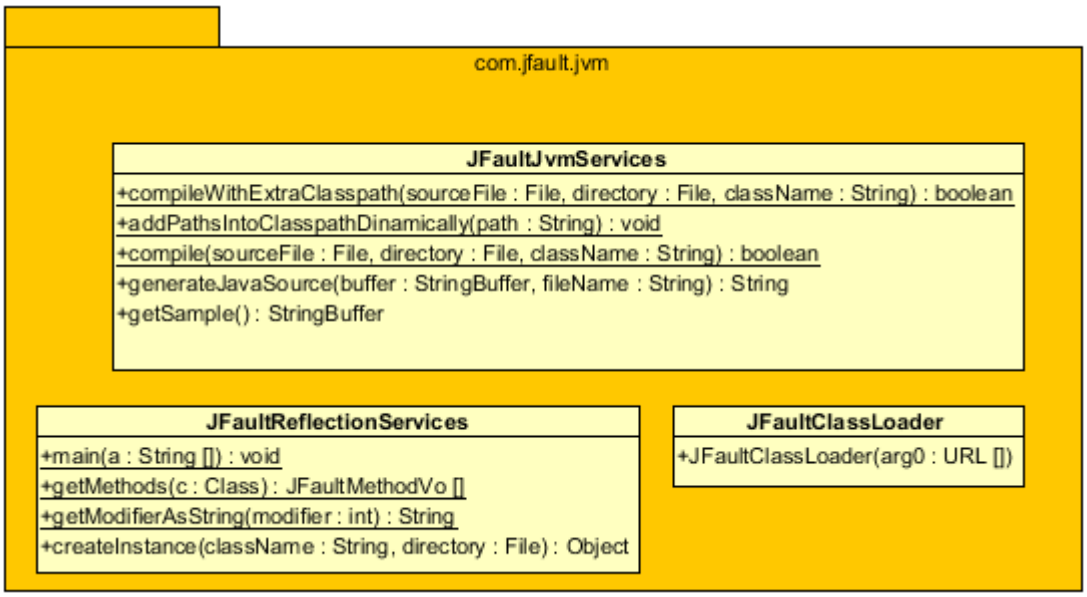

Figura 23 – Pacote de serviços de Compilação Dinâmica e Reflexão

3.2.1.6 Classes de Encapsulamento (com.jfault.vo)

A Figura 24 mostra o pacote que contém as classes que encapsulam os dados usados pelo *framework* e são de menor importância para serem detalhadas. As únicas exceções são as classes “JFaultProxyInterfaceVo”, “JFaultProxyClassVo” e “JFaultStubVo” que, além de encapsular os dados que serão utilizados para criação dos objetos de Proxy e Stub, também contém a lógica de implementação desses objetos de meta-nível que são criados pelo próprio *framework*. As classes de encapsulamento dos Proxies contém a implementação necessária para criação das interfaces e classes de Proxy que serão usadas para exposição remota dos métodos de serviço através de RMI.

A classe “JFaultStubVo”, contém o código Java necessário para tratar as falhas especificadas pelo desenvolvedor da aplicação através das anotações adicionadas nos serviços, além da implementação do método de “hearthBeat()” adicionado dinamicamente pelo *framework* nos Stubs. É importante mencionar que essa classe pode gerar o código fonte dos Stubs com a mesma estrutura dos serviços, pois quando o código Java é requerido, o objeto já possui encapsulada a estrutura das classes de serviço, previamente adquiridas com a utilização de reflexão. O Apêndice G mostra parte do código fonte Java utilizado para criação do método de “heartBeat()”, injetado nos Stubs pelo *framework*. O código é adicionado em um objeto Java do tipo “StringBuffer” através da chamada ao

método “append()”. O código fonte encapsulado nesse objeto é posteriormente compilado dinamicamente para criação dos objetos de meta-nível.

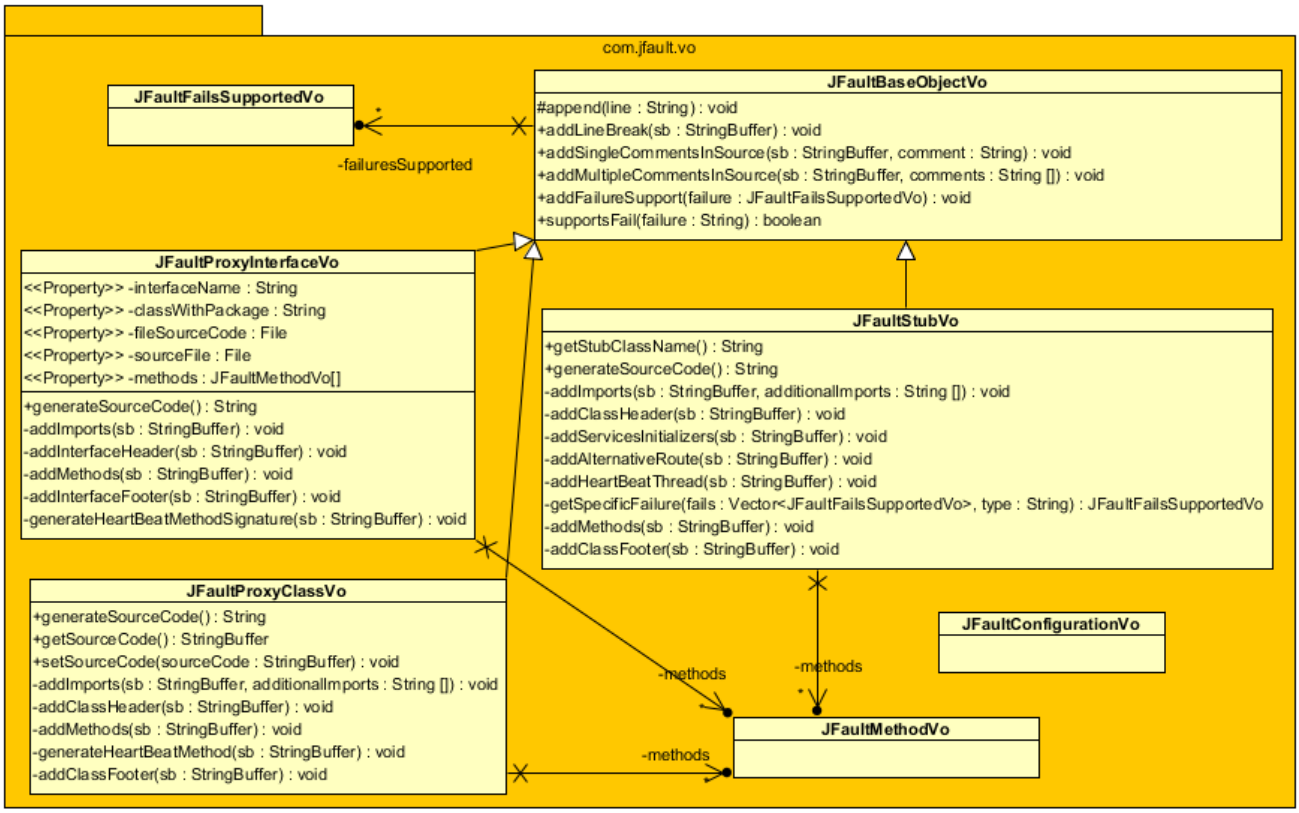

Figura 24 – Pacote de controle do *framework* JFault

3.2.2 JFault Manager Plugin: Criação dos Stubs Tolerantes a Falhas

Conforme visto a nível conceitual, o JFaultManager é o componente fornecido pelo *framework* para criação dos Stubs tolerantes a falhas, que serão utilizados pelo cliente para comunicação com o servidor. A implementação Java desse componente foi realizada através de uma interface gráfica que pode ser acessada pelo desenvolvedor da aplicação através da execução da classe “JFaultManager”, do pacote “jfaultplugin.handlers”. Esse pacote não foi detalhado visto que a implementação das classes que o compõem são simples. Trata-se simplesmente de uma interface que utiliza os serviços do *framework* para criação e compilação dinâmica das classes que contém a lógica para se tolerar as falhas configuradas pelo desenvolvedor da aplicação através das anotações nos serviços.

Para que o JFaultManager possa gerar os Stubs tolerantes a falhas, as seguintes informações sobre o projeto da aplicação devem ser informadas:

1. **Project Path**: diretório raiz do projeto
2. **Project Source Path**: diretório de código fonte do projeto
3. **Project BIN Path**: diretório das classes compiladas do projeto
4. **JFault Meta Level Objects Source Path**: diretório para criação do código fonte dos Stubs gerados
5. **JFault Meta Level Objects BIN Path**: diretório para criação do código compilado dos Stubs gerados
6. **JFault Library Location Path**: diretório para armazenamento da biblioteca gerada pelo JFault
7. **Additional ClassPath**: diretório de bibliotecas adicionais, se necessário

A Figura 25 mostra a interface do JFaultManager e as opções disponíveis para serem utilizadas pelo desenvolvedor da aplicação.

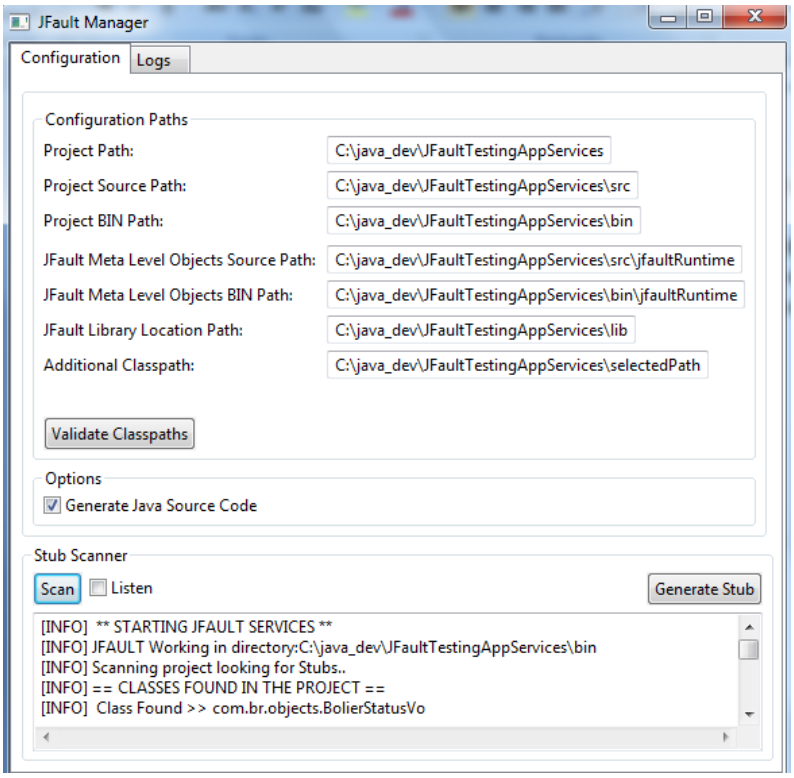

Figura 25 – JFaultManager

Uma vez que as informações dos diretórios são adicionadas, é necessário validá-las, o que pode ser feito clicando-se no botão “Validate Classpaths”. O aplicativo também possui a opção de analisar as classes da aplicação (botão “Scan”), identificando os Stubs

que serão gerados pelo *framework.* Após os diretórios serem validados, os Stubs podem ser gerados nos diretórios configurados pelo desenvolvedor através do uso do botão “Generate Stub”. A opção “Generate Java Source”, quando marcada, disponibilizará também o código fonte Java gerado pelo *framework* para criação do Stub. Todos os *logs* de geração dos Stubs podem posteriormente ser vistos na aba “Logs”, conforme mostra a Figura 26. No final do processo de geração dos Stubs, uma biblioteca (“.jar”) é disponibilizada para o desenvolvedor da aplicação para que possa ser adaptada ao código do cliente. Apesar de existir uma série de procedimentos manuais para criação e adaptação dos Stubs, esse passo será executado pelo desenvolvedor somente na criação dos mesmos, ou quando as assinaturas dos métodos de serviço da aplicação forem alteradas.

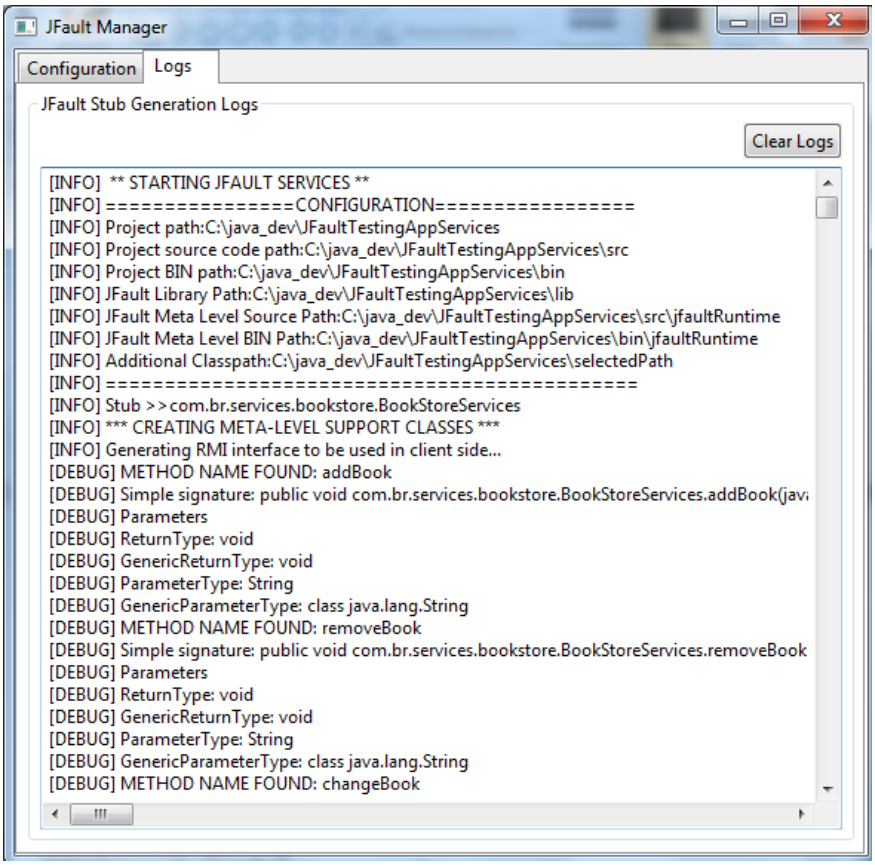

Figura 26 – JFaultManager: Logs de Geração dos Stubs

3.2.3 JFault Manager: Inicialização do Serviço

A inicialização do *framework* será sempre feita no lado servidor da aplicação através da chamada ao método “startService()” da classe “JFaultManager” presente no

pacote “com.jfault.framework”. Conforme já visto, não existe nenhum esforço de implementação, adaptação ou configuração do *framework* nessa etapa – todos os objetos de meta-nivel Proxies serão gerados e compilados dinamicamente pelo *framework*, que fará a exposição remota desses serviços com base nas portas e contextos adicionados pelo desenvolvedor da aplicação nas anotações dos serviços. A Figura 27 mostra a inicialização do *framework* em uma aplicação fictícia, descrevendo os principais passos executados para criação de um dos Proxies da aplicação.

Figura 27 – JFaultManager: Inicialização do *framework* no servidor

3.2.4 Diagramas de Sequência

Essa seção apresenta os diagramas de sequência dos principais processos executados pelo *framework*. A Figura 28 apresenta o processo de criação dos Stubs tolerantes a falhas. Conforme visto anteriormente, esse processo é iniciado pelo próprio desenvolvedor da aplicação através da ferramenta JFaultManager. Após a finalização do processo, uma biblioteca (.jar) é gerada para que o desenvolvedor possa adaptar os Stubs tolerantes a falhas na aplicação.

A geração dos Proxies é feita diretamente no servidor, na própria inicialização do *framework*. A Figura 29 demonstra o processo graficamente – o *framework* é inicializado

através da chamada ao método “startServices()” da classe JFaultManager. Durante a inicialização, o JFaultManager verifica as classes de serviços que devem ser transformadas em Proxies através das anotações no código. Uma vez que essas classes são identificadas, o *framework* constrói dinamicamente todo o código necessário para a geração das interfaces e classes remotas fazendo uso de reflexão, que também é utilizada para instanciação dos objetos após serem compilados pelo *framework*. Após a instanciação dos objetos, os mesmos são expostos para acesso remoto através do protocolo RMI.

3.2.4.1 Criação dos objetos de meta-nível Stubs

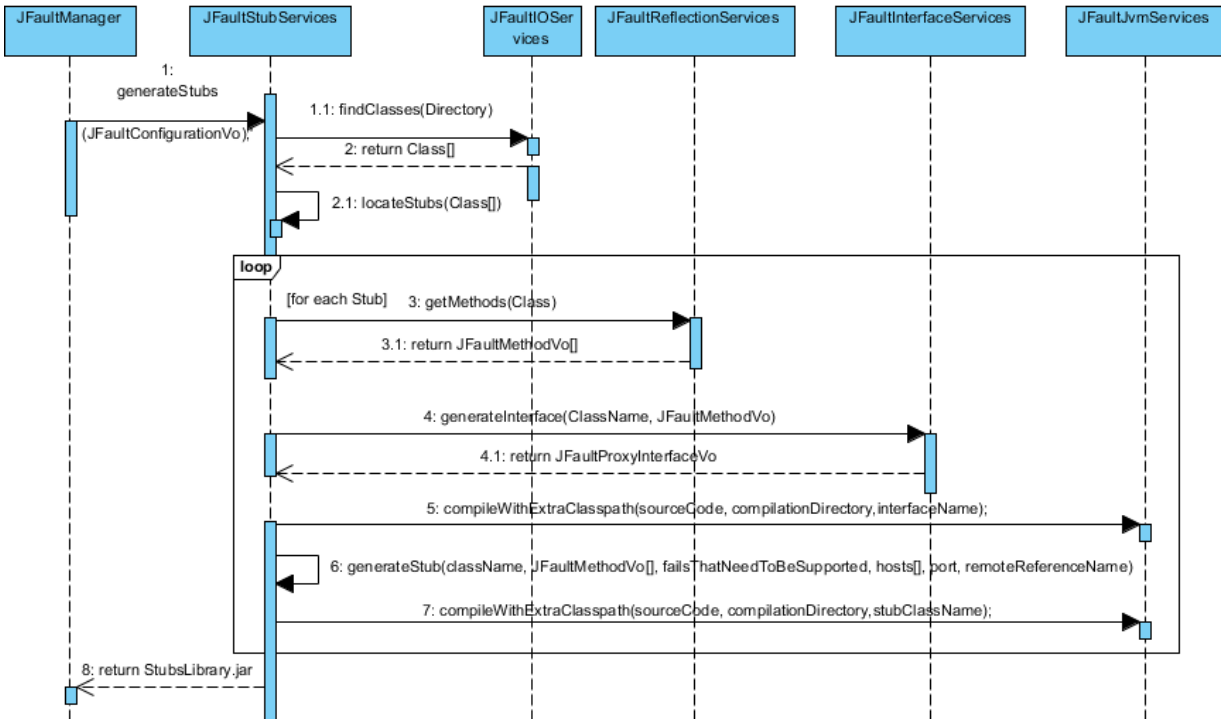

Figura 28 – Diagrama de sequência do processo de criação dos Stubs

3.2.4.2 Criação dos objetos de meta-nível Proxies

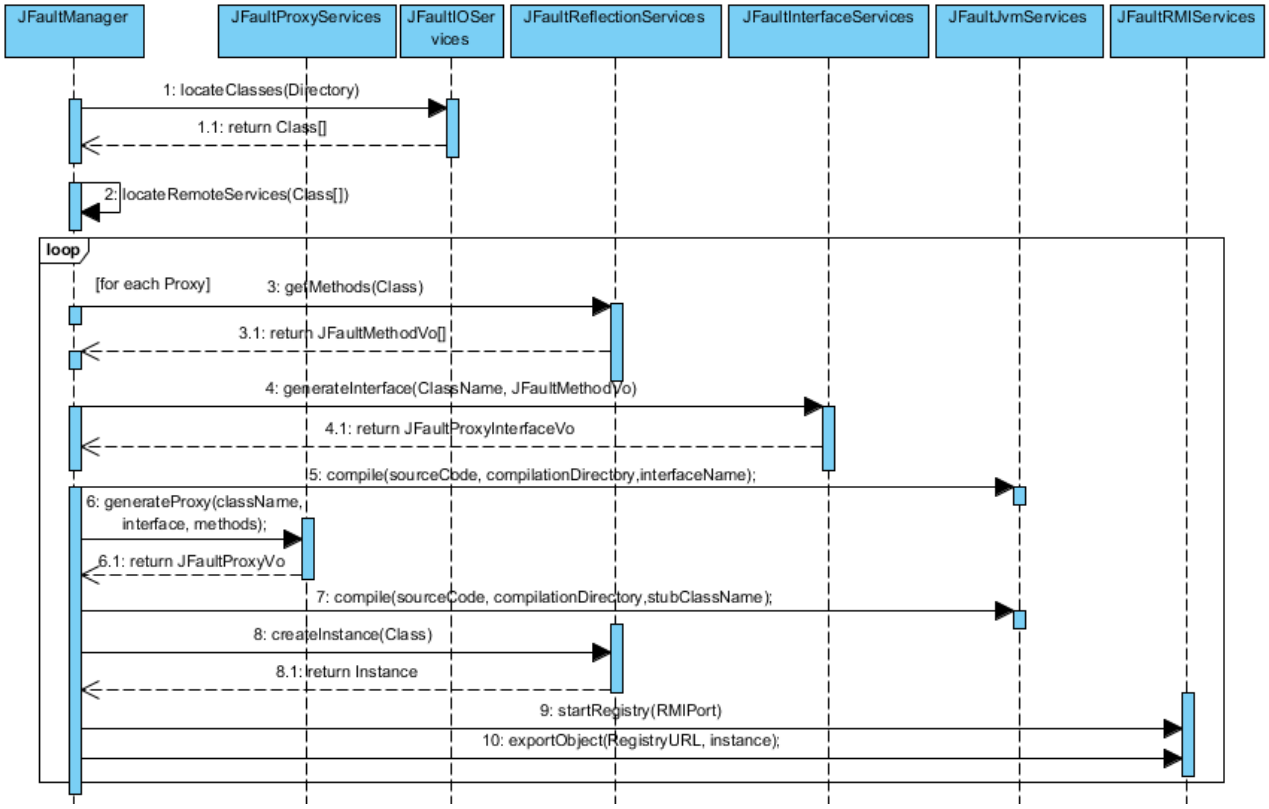

Figura 29 – Diagrama de sequência do processo de criação dos Proxies

3.3 CASO DE USO: JSE BOILER GAUGE

3.3.1 Arquitetura e Implementação da Aplicação

Para demonstrar a aplicabilidade do *framework,* demonstraremos nesse capítulo como ele pode ser utilizado para prover escalabilidade e tolerância a falhas de forma transparente nos serviços de uma aplicação Java para controle de caldeiras. A aplicação é utilizada para controlar temperatura, pressão e quantidade de líquido em quatro diferentes caldeiras. Cada caldeira possui um conjunto de três sensores que são utilizados para realizar a medição. Cada sensor retorna a informação sobre os três atributos a ser monitorados (a redundância é somente utilizada para o caso de falhas). Os três sensores em uma mesma caldeira devem passar exatamente a mesma informação para o serviço, se estiverem em correto funcionamento. A aplicação consiste em um módulo servidor, que contém os serviços que realizam a leitura dos sensores, e um módulo cliente, composto de uma aplicação Java SWT utilizada para apresentar as informações das caldeiras para que possam ser monitoradas. O módulo servidor é montado em três diferentes servidores

de aplicação, também para propósitos de redundância para o caso de algum servidor falhar. O provisionamento de diversos servidores e sensores é importante para garantia do monitoramento das caldeiras no caso de ocorrerem falhas. Se a pressão ou temperatura de alguma caldeira estiver muito alta, a mesma pode explodir. O *framework* será utilizado nesse contexto para prover mecanismos tolerantes a falhas para os serviços de monitoramento, deixando os mesmos escaláveis horizontalmente visto que são dinamicamente expostos para acesso remoto para os clientes. A Figura 30 demonstra a arquitetura da aplicação graficamente.

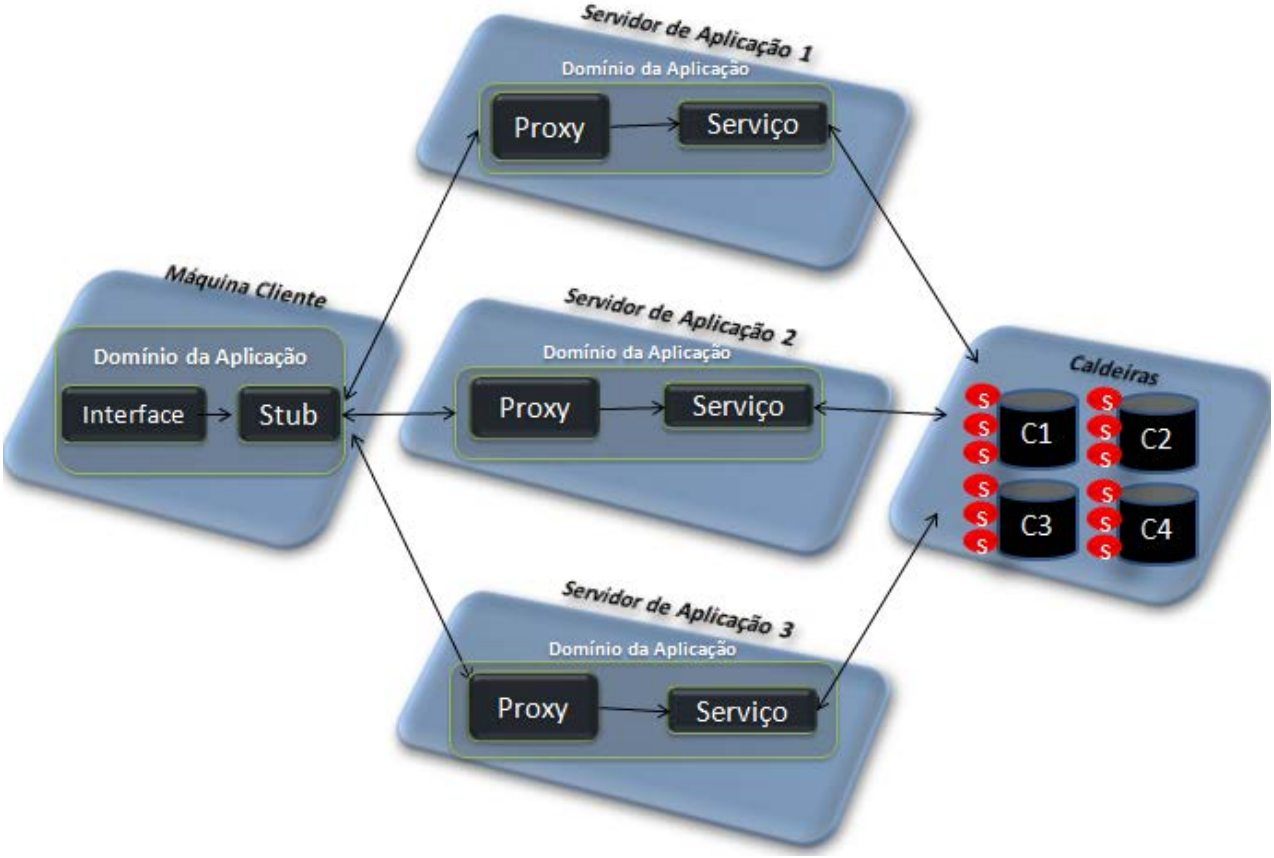

Figura 30 – Boiler Gauge: Arquitetura da Aplicação

A interface da aplicação cliente mostra graficamente a medida dos sensores de cada um dos atributos para as quatro caldeiras. A interface acessa os serviços remotamente através do Stub que controla todo o processo de tolerância a falhas. Em outras palavras, se ocorrerem falhas nos sensores ou em algum servidor de aplicação, a aplicação cliente continuará operando corretamente. Como visto anteriormente, a quantidade de componentes redundantes que devem estar operacionais depende de quantas falhas devem ser toleradas e o tipo de falha. Para falhas de Colapso, é suficiente

que um servidor e um sensor em cada caldeira estejam disponíveis: consequentemente duas falhas nos servidores e sensores podem ser toleradas. A Figura 31 mostra a interface da aplicação.

Figura 31 – Boiler Gauge: Interface da Aplicação

O serviço “BoilerMonitoringService” possui apenas um método (“getStatus()”) que é chamado pelo cliente. O método coleta as informações dos três atributos de cada caldeira e as retorna encapsuladas no objeto “BoilerStatusVo”. A Figura 32 mostra a implementação do serviço com suporte para tolerar falhas de Colapso e de Tempo (definido pelas anotações das linhas 13 e 14 respectivamente). Os valores de temperatura, pressão e quantidade de líquido em cada caldeira estão sendo gerados de forma fictícia, dentro do próprio objeto “BoilerStatusVo”, aleatoriamente.

```
1.  package com.br.services.monitoring;
2.  import java.net.InetAddress;
3.  import java.util.ArrayList;
4.  import com.br.objects.BolierStatusVo;
5.  import com.jfault.annotations.JFaultCrashFT;
6.  import com.jfault.annotations.JFaultRemote;
7.  import com.jfault.annotations.JFaultStub;
8.  import com.jfault.annotations.JFaultTimingFT;
9.  import com.jfault.framework.JFaultConstants;
10.
11. @JFaultRemote(remoteReferenceName="boiler", remoteReferencePort=1030)
12. @JFaultStub(remoteHosts={"localhost", "169.254.69.0"} )
13. @JFaultCrashFT(algorithm=JFaultConstants.LB_ROUND_ROBIN)
14. @JFaultTimingFT(algorithm=JFaultConstants.FT_PREVENTION, general_sla=30000,
15. replica_qtd=2)
16. public class BoilerMonitoringServices {
17.     private BoilerStatusVo boiler1 = new BoilerStatusVo();
18.     private BoilerStatusVo boiler 2 = new BoilerStatusVo();
19.     private BoilerStatusVo boiler 3 = new BoilerStatusVo();
20.     private BoilerStatusVo boiler 4 = new BoilerStatusVo();
21.     private ArrayList<BoilerStatusVo> statusContainer = new ArrayList<BoilerStatusVo>();
22.     public BoilerMonitoringServices(){
23.             boiler1.setBoilerID(1);boiler1.setBolierName("BOILER 1");
24.             boiler2.setBoilerID(2); boiler2.setBolierName("BOILER 2");
25.             boiler3.setBoilerID(3); boiler 3.setBolierName("BOILER 3");
26.             boiler4.setBoilerID(4); boiler 4.setBolierName("BOILER 4");
27.            statusContainer.add(boiler1);statusContainer.add(boiler2);
28.             statusContainer.add(boiler3); statusContainer.add(boiler4);
29.     }
30.     public BoilerStatusVo getStatus(int boilerNrb){
31.         try{
32.             System.out.println("Server "+InetAddress.getLocalHost().getHostAddress()+" replying
33.                 to the request: "+System.currentTimeMillis());
34.         }catch(Exception e){
35.             System.err.println("Unexpected Error: "+e);
36.         }
37.         return statusContainer.get(boilerNrb);
38.      }
39. }
```

Figura 32 – BoilerGauge: Implementação do Serviço de Monitoramento

Uma vez que o serviço foi definido, o desenvolvedor deve gerar o Stub tolerante a falhas através da ferramenta JFaultManager. O *framework*, através dessa ferramenta, utilizará a informação adicionada na anotação "*@Stub*" para identificar o endereço dos servidores onde o serviço será disponibilizado. A informação de porta e contexto para conexão remota será capturada pelo *framework* através da anotação "*@Remote*". Essas informações, em conjunto com as informações das falhas a serem toleradas, será utilizada pelo JFaultManager que criará e compilará dinamicamente todo o código necessário para acesso e tolerância a falhas dos serviços remotos. O código dos Stubs tolerantes a falhas é disponibilizado em formato de biblioteca (.jar), que deve ser adicionada como dependência de projeto na aplicação cliente. Após adicionar a biblioteca ao projeto, o Stub deve estar disponível para ser utilizado. A Figura 33 mostra o código da *thread* Java da interface de monitoramento que efetua as chamadas ao serviço. Como pode ser visto nas linhas 9 a 20, a chamada aos métodos é feita como se o desenvolvedor estivesse acessando a classe de serviços localmente, com a mesma forma de instanciação da classe (linha 3) e assinatura de método.

```
1.  final Thread timeThread = new Thread() {
2.  public void run() {
3.      final BoilerMonitoringServices_JFStub service = new BoilerMonitoringServices_JFStub();
4.      while (true) {
5.          Display.getDefault().syncExec(new Runnable() {
6.              @Override
7.              public void run() {
8.                  try {
9.                      tank.setValue(service.getStatus(0).getBoilerGauge());
10.                     tank2.setValue(service.getStatus(1).getBoilerGauge());
11.                     tank3.setValue(service.getStatus(2).getBoilerGauge());
12.                     tank4.setValue(service.getStatus(3).getBoilerGauge());
13.                     gauge.setValue(service.getStatus(0).getBoilerPressure());
14.                     gauge2.setValue(service.getStatus(1).getBoilerPressure());
15.                     gauge3.setValue(service.getStatus(2).getBoilerPressure());
16.                     gauge4.setValue(service.getStatus(3).getBoilerPressure());
17.                     thermo.setValue(service.getStatus(0).getBoilerTemperature());
18.                     thermo2.setValue(service.getStatus(1).getBoilerTemperature());
19.                     thermo3.setValue(service.getStatus(2).getBoilerTemperature());
20.                     thermo4.setValue(service.getStatus(3).getBoilerTemperature());
21.                 } catch (Exception e) {
```

```
22.                     e.printStackTrace();
23.                     error=true;
24.                 }
25.             }
26.         });
27.         try {
28.             Thread.sleep(1000); // hold on for 1 seg before next call
29.             if (error==true){
30.                 break;
31.             }
32.         } catch (InterruptedException e) {
33.             e.printStackTrace();
34.         }
35.     }
36.     // all available services have failed
37.     JOptionPane.showMessageDialog(null, "There was an error contacting the server, please
38.         contact system administrator.");
39.     }};
```

Figura 33 – BoilerGauge : Chamada do serviço de monitoramento através do Stub

3.3.2 Experimentos

Nessa seção serão apresentados alguns experimentos executados com a utilização do *framework*. A configuração das máquinas utilizadas nos experimentos pode ser vista na Tabela 6. A versão do Java utilizada foi JDK1.7.0.21.

Tabela 6 – Configuração das máquinas utilizadas nos experimentos

Recurso	Configuração	Memória	Sistema Operacional	IP
Servidor 1	Intel® Celeron ® CPU 550 @ 2.00 GHz	2 GB	Windows 7 Professional – 32 bits	192.168.124.1
Servidor 2	Intel® Core I5-2410M CPU @ 2.30 GHz	6 GB	Windows 7 Home Premium – 64 bits	192.168.0.111

3.3.2.1 Tempo de Inicialização do Framework

Conforme visto anteriormente, a utilização de compilação dinâmica pode ser muito útil em algumas situações, mas adiciona um tempo extra de processamento (overhead) que pode impactar a aplicação. No caso do *framework* JFault, essa questão deve ser levada em consideração principalmente na inicialização do serviço, visto que nessa etapa o *framework* gera vários objetos de meta-nível e os compila dinamicamente – mais especificamente dois objetos (um interface e uma classe de Proxy) por serviço. Esse experimento visa demonstrar o tempo extra de processamento adicionado pelo processo de compilação dinâmica do *framework*, variando a quantidade de serviços disponibilizados pela aplicação iniciando-o nas duas máquinas mostradas na Tabela 6. O gráfico a seguir (Figura 34) mostra os tempos de inicialização do *framework* (em segundos no eixo vertical) variando conforme mais serviços vão sendo adicionados na aplicação (de um até 80 serviços, no eixo horizontal).

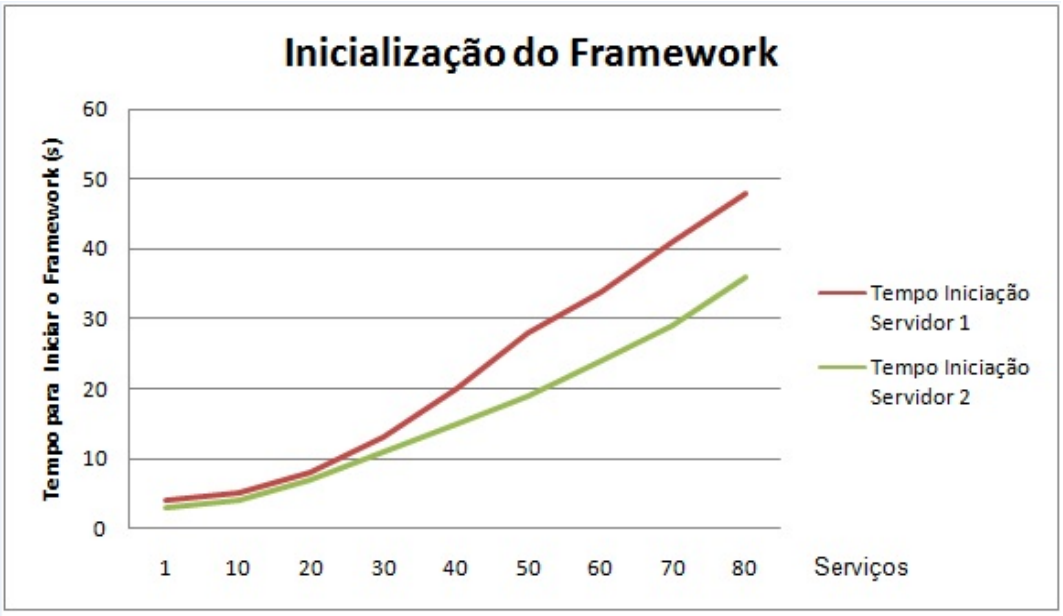

Figura 34 – Tempos de Inicialização do Framework

Podemos ver no gráfico que com 80 serviços (cada serviço com cinco operações) o *framework* é inicializado em aproximadamente 50 segundos no “Servidor 1” e mais rapidamente – aproximadamente 35 segundos – no “Servidor 2”, que possui uma capacidade superior de processamento. A conclusão do experimento é que o tempo de inicialização do *framework* cresce à medida que mais serviços vão sendo adicionados,

mas que o tempo despendido na criação e inicialização dos objetos de meta-nível é relativamente pequeno e bastante aceitável na maioria dos casos. Foi mensurado também, separadamente, o tempo despendido pelo *framework* com as tarefas de reflexão, compilação dinâmica e exposição remota de serviços. Inicializando a aplicação com 80 serviços, no “Servidor 1”, o *framework* levou aproximadamente seis segundos para capturar a estrutura dos serviços através de reflexão e aproximadamente três segundos para expor os serviços remotamente. O restante do tempo, aproximadamente 42 segundos, é despendido nas tarefas de compilação dinâmica, o que leva à conclusão de que quase 85% do *overhead* adicionado pelo *framework* na inicialização da aplicação advêm do emprego da compilação dinâmica para criação dos objetos de meta-nível.

3.3.2.3 Simulação de Falhas de Colapso no Servidor

Esse experimento visa demonstrar a atuação do *framework* no caso de ocorrerem falhas de Colapso nos serviços utilizados pelo cliente. Utilizando a aplicação “Boiler Gauge”, manteremos a arquitetura do sistema de forma similar à mostrada na Figura 44, exceto pela quantidade de servidores utilizada. Para execução desse experimento, a máquina “Servidor 1” foi utilizada como servidor, ou seja, contém uma instância do *framework* executando a aplicação que provém os serviços do sistema BoilerGauge. A máquina “Servidor 2” foi utilizada como cliente e servidor, também contém uma réplica dos componentes de serviço e foi utilizada adicionalmente como cliente da aplicação, executando a interface de monitoramento do sistema. O serviço de leitura dos sensores foi implementado de forma fictícia, retornando números baseados na data atual dos servidores. Para que as mesmas informações de sensores sejam retornadas pelas duas máquinas, os relógios foram sincronizados.

Inicialmente foi testado o dispositivo de tolerância a falhas do *framework* utilizando o mecanismo de balanceamento de carga “Round Robin”. Após a inicialização dos serviços de negócio da aplicação no servidor através do *framework*, a aplicação cliente foi inicializada. Como pode ser visto nos *logs* do *framework*, na Figura 49, quando o cliente é inicializado, ele gera uma instância do Stub que primeiramente tenta criar as instâncias de serviços conectando remotamente nas mesmas (linhas 2 e 3). Se essa conexão falhar em algum servidor, a instância não é adicionada na lista de serviços disponíveis (*pool*), contudo, no caso de tratamento de falhas de Colapso, o *framework* mantém o funcionamento dos serviços se existir ao menos uma instância funcional. Em outras

palavras, mesmo se houverem falhas em um ou mais serviços na inicialização do cliente, contanto que exista ao menos uma instância funcionando corretamente, o mesmo continua operando como se nenhum problema houvesse ocorrido, visto que o Stub tolerará as falhas.

Nas linhas 4 e 9 podemos ver a *thread* de monitoramento dos serviços verificando as instâncias de tempos em tempos (inicialmente de segundo em segundo). Nas linhas 5 a 8, podemos ver os *logs* das chamadas do cliente aos serviços, sendo uniformemente distribuídas visto que o algoritmo *"Round-Robin*" foi selecionado (Figura 35).

1. *[INFO] ** FRAMEWORK JFAULT STARTING **
2. *[INFO] ** Refreshing instance 192.168.124.1:1030 **
3. *[INFO] ** Refreshing instance 192.168.0.111:1030 **
4. *[INFO] ** [HEARTBEAT] Instance 0 for BoilerMonitoringServices_JFInterface is responding fine and will be kept in the pool **
5. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
6. *[INFO] ** Instance 1 for BoilerMonitoringServices_JFInterface is being called **
7. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
8. *[INFO] ** Instance 1 for BoilerMonitoringServices_JFInterface is being called **
9. *[INFO] ** [HEARTBEAT] Instance 1 for BoilerMonitoringServices_JFInterface is responding fine and will be kept in the pool **

Figura 35 – BoilerGauge : chamada aos serviços através do Stub

O próximo passo do experimento consiste em simular uma falha de Colapso em um dos servidores. Para essa simulação, a máquina "Servidor 1" foi abruptamente desligada enquanto o cliente enviava requisições solicitando dados de medição das caldeiras. O resultado pode ser visto nos *logs* do *framework*, mostrados na Figura 36. Nas linhas 2 e 3 podemos ver as duas instâncias ainda em pleno funcionamento, antes do desligamento da máquina. Na linha 4, após o desligamento da máquina, podemos verificar a constatação do *framework* de que a "Instância 1" dos serviços não mais responde às requisições e está sendo removida da lista de serviços. Nas linhas 6 e 7 podemos ver o cliente dando prosseguimento às chamadas, usando agora somente a "Instância 0", normalmente, como se nenhum problema houvesse ocorrido. As chamadas roteadas a essa instância são reenviadas pelo *framework*, ou seja, o *framework* garante que

nenhuma requisição do cliente será perdida se pelo menos uma das instâncias de serviço estiver funcional.

1. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is responding fine and will be kept in the pool **
2. *[INFO] ** Instance 1 for BoilerMonitoringServices_JFInterface is being called **
3. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
4. *[ERROR] ** Instance 1 for BoilerMonitoringServices_JFInterface has failed and removed from the pool **
5. *[ERROR] ** ONE OR MORE INSTANCES OF BoilerMonitoringServices_JFInterface HAVE FAILED, JFAULT IS TRYING TO RECOVER **
6. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
7. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **

Figura 36 – BoilerGauge: Tolerância a Falha de Colapso

Como já visto, o *framework* é responsável também por continuar o monitoramento das instâncias que apresentaram problemas, através da *thread* de monitoramento (*Heart Beat).* A Figura 37 mostra o serviço sendo novamente reestabelecido automaticamente pelo *framework* (linhas 2 a 6) e inserido na tabela de serviços (linha 7), após o reestabelecimento da máquina “Servidor 1” e inicialização dos serviços no servidor. Nas linhas 8 e 9 podemos ver o *framework* novamente utilizando os dois serviços.

1. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
2. *[INFO] **
3. *[INFO] ** REFRESHING POOL SERVICES: TRYING TO AUTO RECOVER INSTANCES ***
4. *[INFO] **
5. *[INFO] ** Refreshing instance 192.168.124.1:1030 **
6. *[INFO] ** Refreshing instance 192.168.0.111:1030 **
7. *[INFO] ** Instance 1 for BoilerMonitoringServices_JFInterface has been recovered and will be added in the pool **
8. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
9. *[INFO] ** Instance 1 for BoilerMonitoringServices_JFInterface is being called **

Figura 37 – BoilerGauge : Recuperação do Serviço

Na segunda parte desse experimento foi verificado o funcionamento de falhas de Colapso quando o *framework* é configurado para utilizar o algoritmo de balanceamento de carga baseado em pesos (LB_ROUND_ROBIN_WEIGHTED). O experimento é idêntico ao anterior exceto pela configuração do balanceamento de carga no serviço de medição que não será mais uniforme. Como pode ser visto na Figura 38 a máquina “Servidor 1” foi configurada para receber somente 20% da carga e a máquina “Servidor 2” receberá os outros 80% (configurações após a caractere “@” na anotação JFaultStub – linha 2).

```
1. @JFaultRemote(remoteReferenceName="boiler", remoteReferencePort=1030)
2. @JFaultStub(remoteHosts={"192.168.124.1@80", "192.168.0.111@20"} )
3. @JFaultCrashFT(algorithm=JFaultConstants.LB_ROUND_ROBIN_WEIGHTED)
4. public class BoilerMonitoringServices { . . .
```

Figura 38 – BoilerGauge : Utilização não uniforme de balanceamento de serviço

Após a inicialização do cliente, instanciação e utilização do Stub para envio de algumas requisições ao serviço, podemos ver as chamadas sendo corretamente roteadas conforme mostra a Figura 39.

```
1.  [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 1, DISTRIBUTION: 20% **
2. [INFO] ** Instance 1 for BoilerMonitoringServices_JFInterface is being called **
3. [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 1, DISTRIBUTION: 20% **
4. [INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is responding fine and will be kept in
   the pool **
5. [INFO] ** Instance 1 for BoilerMonitoringServices_JFInterface is being called **
6. [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 0, DISTRIBUTION: 80% **
7. [INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
8. [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 0, DISTRIBUTION: 80% **
9. [INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
10. [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 0, DISTRIBUTION: 80% **
11. [INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
12. [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 0, DISTRIBUTION: 80% **
13. [INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
14. [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 0, DISTRIBUTION: 80% **
15. [INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
16. [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 0, DISTRIBUTION: 80% **
```

17. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
18. *[INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 0, DISTRIBUTION: 80% **
19. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
20. *[INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 0, DISTRIBUTION: 80% **
21. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
22. *[INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 1, DISTRIBUTION: 20% **
23. *[INFO] ** Instance 1 for BoilerMonitoringServices_JFInterface is being called **
24. *[INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 1, DISTRIBUTION: 20% **

Figura 39 – BoilerGauge: Distribuição das Requisições de Forma não Uniforme

Na Figura 40 podemos ver o comportamento do *framework* quando uma falha de Colapso é simulada em uma das instâncias. Nas linhas 1 e 2 podemos ver o *framework* removendo a instância colapsada do *pool* de serviços redirecionando 100% das chamadas à outra (linhas 3 a 6).

1. *[ERROR] ** Instance 0 for BoilerMonitoringServices_JFInterface has failed and removed from the pool **
2. *[ERROR] ** ONE OR MORE INSTANCES OF BoilerMonitoringServices_JFInterface HAVE FAILED, JFAULT IS TRYING TO RECOVER **
3. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
4. *[INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 1, DISTRIBUTION: 100% **
5. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
6. *[INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 1, DISTRIBUTION: 100% **

Figura 40 – BoilerGauge: Redirecionamento de Carga após Colapso

Uma vez que o serviço é recuperado, o *framework* automaticamente reestabelece a comunicação com o mesmo, assim como o balanceamento de carga original, como pode ser visto na Figura 41.

1. *[INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
2. *[INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 0, DISTRIBUTION: 100% **
3. *[INFO] **
4. *[INFO] ** REFRESHING POOL SERVICES: TRYING TO AUTO RECOVER INSTANCES ***
5. *[INFO] **
6. *[INFO] ** Instance 1 for BoilerMonitoringServices_JFInterface is being called **

```
7.  [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 1, DISTRIBUTION: 20% **
8.  [INFO] ** Instance 1 for BoilerMonitoringServices_JFInterface is being called **
9.  [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 1, DISTRIBUTION: 20% **
10. [INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
11. [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 0, DISTRIBUTION: 80% **
12. [INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
13. [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 0, DISTRIBUTION: 80% **
14. [INFO] ** Instance 0 for BoilerMonitoringServices_JFInterface is being called **
15. [INFO] ** LB WEIGHTED ALGORITHM SELECTED INSTANCE: 0, DISTRIBUTION: 80% **
16. . . . More 5 times
```

Figura 41 – BoilerGauge : Restabelecimento do Serviço

3.3.2.4 Simulação de Falhas de Tempo no Servidor

Esse experimento visa demonstrar a atuação do *framework* no caso do serviço estar configurado para tratar falhas de Tempo, ou seja, como o framework atua para preveni-las e como o *framework* reage no caso do servidor não responder à requisição no tempo estipulado. Foi utilizada a mesma aplicação “Boiler Gauge” e a arquitetura idêntica ao experimento demonstrado na Seção 3.3.2.3. A Figura 32 mostra os detalhes da configuração do serviço, que utilizará duas réplicas para tentar prevenir falhas de Tempo, sendo que o *SLA* configurado (tempo máximo de resposta) é de 30 segundos.

O primeiro experimento demonstra unicamente o funcionamento do *framework* quando o serviço está configurado para tratar falhas de Tempo, em outras palavras, o objetivo é somente mostrar a atuação do *framework* na comunicação entre o cliente e o servidor para completo entendimento do algoritmo utilizado. Um dos serviços, sendo executado na máquina de endereço IP “192.168.124.1”, foi modificado para que responda com um atraso (delay) de 100 milissegundos – assim será possível verificar o correto funcionamento do algoritmo de seleção de réplicas do *framework*. Após a inicialização dos serviços no servidor e da aplicação cliente, podemos ver na Figura 42 os *logs* de comunicação do Stub após algumas requisições. Note que no caso, a operação “getStatus()” está sendo chamada de tempos em tempos por um processo do cliente (Figura 33). É importante mencionar também que a tabela de monitoração dos serviços está vazia inicialmente, visto que o framework foi recentemente iniciado nos servidores.

Na linha 1, podemos ver a requisição feita a operação “getStatus()”, uma vez que a requisição é realizada ao Stub o mesmo analisa a tabela de monitoração dos serviços

para verificar as réplicas que estão respondendo mais rapidamente (linhas 2 a 9). No caso, como nenhuma requisição foi enviada ainda ao serviço a tabela se encontra vazia, logo, duas (parâmetro “replica_qtd” informado na anotação “@JFaultTimingFT”) réplicas quaisquer são selecionadas pelo Stub para atender a requisição. Após selecionar as réplicas, o Stub realiza a chamada às duas réplicas, mas aguarda somente pelo resultado da primeira. Como pode ser visto nas linhas 10,11 e 12, uma vez que a primeira resposta é recebida, o Stub cancela a chamada às outras réplicas (cancela do ponto de visto do cliente, o processo ainda continua sendo executado no servidor).

```
1.  [INFO] Request being made to getStatus operation
2.  [INFO] ============== MONITORING TABLE ================
3.  [INFO] Service: BoilerMonitoringServices_JFInterface, endpoint:[192.168.124.1:50803]
4.  [INFO] Operation: getStatus
5.  [INFO] Respose Times: []
6.  [INFO] Service: BoilerMonitoringServices_JFInterface,endpoint:[192.168.0.111:49433]
7.  [INFO] Operation: getStatus
8.  [INFO] Respose Times: []
9.  [INFO] ================================================
10. [INFO] Calling Service 192.168.124.1 asynchronously
11. [INFO] Calling Service 192.168.0.111 asynchronously
12. [INFO] Service 192.168.124.1 has responded within the SLA, the requests made to other
13. instances are being cancelled
14. [INFO] Request being made to getStatus operation
15. [INFO] ============== MONITORING TABLE ================
16. [INFO] Service: BoilerMonitoringServices_JFInterface, endpoint:[192.168.124.1:50803]
17. [INFO] Operation: getStatus
18. [INFO] Respose Times: [15]
19. [INFO] Service: BoilerMonitoringServices_JFInterface,endpoint:[192.168.0.111:49433]
20. [INFO] Operation: getStatus
21. [INFO] Respose Times: [118]
22. [INFO] ================================================
23. [INFO] Calling Service 192.168.124.1 asynchronously
24. [INFO] Calling Service 192.168.0.111 asynchronously
25. [INFO] Service 192.168.124.1 has responded within the SLA, the requests made to other
26. instances are being cancelled
27. [INFO] Request being made to getStatus operation
28. [INFO] ============== MONITORING TABLE ================
```

29. *[INFO] Service: BoilerMonitoringServices_JFInterface, endpoint:[192.168.124.1:50803]*
30. *[INFO] Operation: getStatus*
31. *[INFO] Respose Times: [15,13]*
32. *[INFO] Service: BoilerMonitoringServices_JFInterface,endpoint:[192.168.0.111:49433]*
33. *[INFO] Operation: getStatus*
34. *[INFO] Respose Times: [118,122]*
35. *[INFO] ==========*
36. *[INFO] Calling Service 192.168.124.1 asynchronously*
37. *[INFO] Calling Service 192.168.0.111 asynchronously*
38. *[INFO] Service 192.168.124.1 has responded within the SLA, the requests made to other*
39. *instances are being cancelled*

Figura 42 – BoilerGauge : Tabela de monitoração para falhas de Tempo

Na Linha 14 podemos ver uma segunda chamada sendo feita ao serviço “getStatus()”, nesse caso, os Proxies já possuem um histórico de tempo para essa operação (repassada ao Stub através da chamada constante ao método de *heartbea*t*)* que é de 15 milissegundos para a réplica sendo executada no servidor de IP “192.168.124.1” e 118 milissegundos para a réplica sendo executada no servidor “192.168.0.111” (linhas 18 e 21). O comportamento do *framework* nesse caso será de selecionar as duas réplicas que estão respondendo mais rapidamente as requisições entre todas as réplicas disponíveis (no caso do experimento possuímos somente duas), ordenar essa lista e efetuar as requisições, como já mencionado, aguardando somente a primeira que responder. Nas linhas 31 e 34 podemos ver o histórico de tempos de chamadas aumentando à medida que mais requisições são realizadas (as últimas cinco medidas de tempo por operação são armazenadas), quando existe mais de uma medida de tempo o *framework* faz uma média simples para seleção das réplicas.

O próximo passo desse experimento consiste em simular uma falha de Colapso em uma das réplicas. Para isso, a máquina de IP “192.168.124.1” (que possuía os menores tempos de resposta do serviço) foi abruptamente desligada. Podemos ver o comportamento do framework nesse caso analisando os *logs* do Stub disponíveis na Figura 43.

1. *[INFO] Request being made to getStatus operation*
2. *[INFO] ========== MONITORING TABLE ==========*
3. *[INFO] Service: BoilerMonitoringServices_JFInterface, endpoint:[192.168.124.1:50803]*
4. *[INFO] Operation: getStatus*

```
5.  [INFO] Respose Times: [0, 16, 0, 0, 0]
6.  [INFO] Service: BoilerMonitoringServices_JFInterface,endpoint:[192.168.0.111:49433]
7.  [INFO] Operation: getStatus
8.  [INFO] Respose Times: [109, 125, 109, 109, 109]
9.  [INFO] ================================================
10. [INFO] Calling Service 192.168.124.1 asynchronously
11. [INFO] Calling Service 192.168.0.111 asynchronously
12. [INFO] java.rmi.ConnectException: Connection refused to host: 192.168.124.1;
13. [INFO] Service 192.168.0.111 has responded within the SLA, the requests made to other
14. instances are being cancelled
```

Figura 43 – BoilerGauge : Simulação de falha de Colapso

Como pode ser visto na linha 12, o *framework* identificou uma falha de Colapso durante o processo de chamada assíncrona dos serviços, contudo, visto que a segunda réplica selecionada foi capaz de responder a requisição do cliente dentro do tempo máximo estipulado, a falha foi tolerada, ou seja, não produziu nenhum impacto ao cliente. Nesse caso, é importante mencionar que assim que a réplica estiver disponível novamente, será automaticamente reinserida no conjunto de serviços para ser utilizada pelo *framework*, todavia, com a tabela de monitoramento vazia, ou seja, sem históricos prévios de tempos de resposta.

O último passo do experimento consiste em injetar uma falha de Tempo nos dois serviços sendo utilizados no experimento. Para isso, foi feita uma pequena alteração de código para que ambos os serviços demorem mais do que 30 segundos para responder. Nesse caso, após uma requisição do cliente, é possível ver que o *framework* não consegue tolerar a falha, visto que nenhum serviço foi capaz de responder as requisições dentro do *SLA* estipulado. Os *logs* do *framework* para esse cenário podem ser vistos na Figura 44.

```
1. [INFO] Calling Service 192.168.124.1 asynchronously
2. [INFO] Calling Service 192.168.0.111 asynchronously
3. [ERROR] com.jfault.exceptions.JFaultSLAViolatedException: getStatus operation did not respond
   within SLA, SLA configured: 30000 milliseconds
```

Figura 44 – BoilerGauge: Serviços não respondendo dentro do SLA

Considerando que nessas circunstâncias o *framework* não foi capaz de prevenir e nem tolerar a falha, uma exceção será submetida para o cliente, que deve tratá-la como melhor lhe convier. No caso da aplicação “BoilerGauge”, como pode ser visto na Figura 45, a exceção é tratada pela interface do sistema, que mostra uma mensagem de erro ao usuário.

Figura 45 – Boiler Gauge: Erro na chamada aos serviços

3.3.2.5 Impacto de Desempenho na Comunicação

O objetivo desse experimento é de avaliar o impacto de desempenho do *framework* na comunicação entre o cliente e o servidor, em outras palavras, visa verificar se a utilização do *framework* adiciona algum tempo extra (*overhead*) na comunicação entre o cliente e o servidor, e caso positivo, se esse tempo é aceitável. Para esse experimento foi utilizada a mesma aplicação “Boiler Gauge” dos experimentos anteriores, com a mesma arquitetura. Durante esse experimento, partiremos do princípio que o serviço responde imediatamente a requisição, ou seja, todo tempo medido entre a requisição e a resposta é relacionado com a comunicação.

Os testes foram realizados em duas rodadas: na primeira rodada o cliente permaneceu fazendo requisições ao servidor durante um minuto, se comunicando com os

servidores através do *framework*, sendo que todos os tempos das requisições foram coletados. Os testes foram feitos com o suporte do *framework* a falhas de Colapso e posteriormente com suporte às de Tempo. Na segunda rodada, o mesmo teste foi executado, porém, ao invés de utilizar o *framework* foi feito uma pequena alteração no cliente para que o mesmo se comunique com o servidor utilizando diretamente o protocolo RMI. Esses tempos de requisições também foram coletados. No final uma média simples entre os tempos foi realizada para os dois cenários e então os tempos foram comparados. É importante salientar que nenhuma falha foi injetada durante os testes, visto que o experimento tem o objetivo somente de avaliar o tempo e o *overhead* de comunicação utilizando o *framework*.

A Figura 46 apresenta um gráfico demonstrando a média dos tempos coletados. Como pode ser verificado, a utilização do *framework* levou a um pequeno acréscimo de tempo na comunicação. Com o suporte a falhas de Tempo, foram adicionados aproximadamente 294 milissegundos a cada requisição; mais tempo se comparado às requisições enviadas através do *framework* configurando o Stub para suportar somente falhas de Colapso (151 milissegundos). Os tempos de resposta das requisições enviadas sem a utilização do *framework* ficaram na média em 135 milissegundos, esse fato se deve ao próprio tempo de rede e operações de *Marshalling e Unmarshalling* do protocolo RMI.

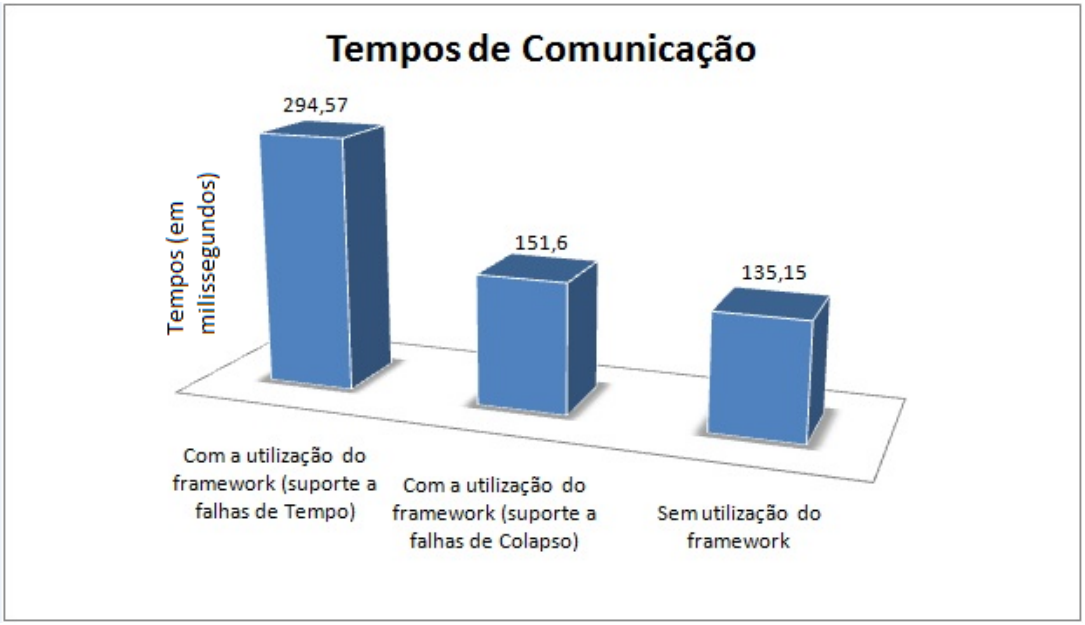

Figura 46 – Tempos de Comunicação

A conclusão do experimento é que a utilização do framework adiciona um tempo relativamente pequeno à comunicação, que parece bastante aceitável para a maioria das aplicações.

4 TRABALHOS RELACIONADOS

Esse capítulo apresenta alguns trabalhos relacionados ao tema de pesquisa proposto. Os primeiros quatro trabalhos são *frameworks* que, da mesma forma que o JFault, propõem-se a fornecer mecanismos auxiliares para tolerância de falhas e comunicação remota de processos. Os demais trabalhos são relacionados a artigos que abordam como as arquiteturas de meta-níveis podem ser utilizadas para tolerar falhas. No Capítulo 5 é apresentado como esses trabalhos se relacionam com a presente proposta do *framework* JFault e quais são suas características potencialmente originais.

4.1 FRAMEWORKS SIMILARES

4.1.1 Framework FT-Java

4.1.1.1 Descrição e Funcionamento

Thomas et al. [TOM03] criaram um *framework* Java (FT-Java) para construção de sistemas distribuídos tolerantes a falhas. O modelo proposto pelos autores suporta diferentes tipos de padrões de tratamento de falhas do tipo Fail-Stop: Máquinas de Estado Replicadas (*Replicated State Machines*) e Ações Reiniciáveis (*Restartable Actions*), quando um módulo é reiniciado voltando para um estado anteriormente armazenado. O *framework* utiliza reflexão Java para implementar os mecanismos tolerantes a falhas.

No caso do modelo de Máquinas de Estado Replicadas, cada módulo Fail-Stop (FS) precisa estender uma classe abstrata chamada “FSObject”, sendo que as operações dos módulos devem ser implementadas como métodos na interface “FSInterface”. Várias instâncias do mesmo módulo são compostas em um grupo, fornecendo a visão de que o grupo é um módulo único. Operações invocadas no grupo são distribuídas a todas as instâncias (*multicast*). A utilização de reflexão é feita quando os módulos são criados e

replicados (com a utilização do método *newInstance*); no caso da replicação, o estado do objeto também é replicado (mantido).

Já no modelo de Ações Reiniciáveis (proposto, mas não implementado), os módulos Fail-Stop (FS) armazenam o seu estado interno (estado de *checkpoint*) para que os módulos que eventualmente vierem a falhar possam recuperar o seu estado.

O framework proposto por Thomas et al. utiliza o protocolo RMI para suportar a criação e invocação de objetos distribuídos. Todos os módulos FS devem ser subtipos da classe “Activatable” do RMI, assim como todas as FSInterfaces precisam estender a interface “Remote” do RMI.

O serviço “HostFTManager”, em cada máquina (*host*), é responsável por gerenciar as JVMs e monitorá-las usando RMI. Esse serviço envia uma notificação ao cliente em caso de falhas (se todas as réplicas de um serviço vierem a falhar).

O método “newInstance” da classe “FSObject” é utilizado para criação de módulos FS. Para que o cliente crie um módulo, uma lista de máquinas (*hosts*) deve ser passada, assim como o nome da aplicação proprietária do módulo e lista de máquinas que possuirão os serviços de *backup* desse módulo. O cliente recebe como retorno um objeto de Proxy para efetuar as chamadas aos módulos FS.

4.1.1.2 Comparação com o JFault

O *framework* FT-Java suporta tolerância a um número mais limitado de falhas, detendo-se unicamente ao modelo de falhas do tipo Fail-Stop. Existe também a necessidade de programar alguns aspectos da comunicação (como estender e implementar requisitos do protocolo RMI) que são absolutamente transparentes no caso do JFault. O modelo de anotações utilizado pelo JFault deixa os serviços mais simples, ou seja, os serviços de negócio no caso do JFault não possuem absolutamente nenhum código ou requisito de implementação que não seja relativo à lógica de negócio (todo processo de tolerância a falhas e exposição remota de objetos é feito através de anotações e tratados no meta-nível). Contudo, o framework FT-Java possui um mecanismo interessante de sincronismo de réplicas, que mantém e replica o estado de um módulo em várias réplicas para o caso de falhas. Mecanismo de sincronismo de réplicas não existe no JFault, que parte do princípio que os serviços não precisam manter estado (são *stateless*), o estado deve ser mantido e gerenciado pela camada de persistência.

4.1.2 Framework Disal

4.1.2.1 Descrição e Funcionamento

Biely et al. [BIE13] desenvolveram o *framework* Disal para implementação de sistemas tolerantes a falhas baseado em pseudocódigo (*pseudocode).* Disal é implementado como uma biblioteca Scala [SCA14] e consiste de duas partes principais: uma linguagem de domínio específico (Domain Specific Language-*DSL*), em que os algoritmos são expressos, e uma camada de mensagens que lida com questões de baixo nível, como o gerenciamento de conexões, de processos paralelos e serialização.

Usando DSL, é possível escrever algoritmos utilizando uma linguagem muito similar às utilizadas para descrever algoritmos em produções textuais científicas. Segundo os autores, a utilização de uma linguagem similar poderia tornar as implementações mais claras e convincentes visto que refletiriam a mesma linguagem. O *framework* pode converter a linguagem gerada em pseudocódigo para Java ou C++.

4.1.2.2 Comparação com o JFault

Embora ambos os *frameworks* tenham a mesma finalidade – suportar a criação de aplicações distribuídas e tolerantes a falhas – o framework JFault se apresenta como uma melhor opção dentro de seu escopo de atuação (tolerância de falhas em processos de negócio em aplicações três camadas). Nesse escopo, o framework proposto neste trabalho é preferível, pois elimina a necessidade de programação dos mecanismos de tolerância a falhas. Ou seja, o desenvolvedor não precisará escrever nenhuma linha de código para que os processos de negócio tolerem falhas (tudo é feito através de anotações). Contudo, fica evidente que o framework Disal é mais flexível e pode ser utilizado para implementação de diversos algoritmos para comunicação de processos e tolerância a falhas.

4.1.3 Framework Akka

4.1.3.1 Descrição e Funcionamento

O *framework* Akka [AKK13] fornece suporte para criação de aplicações distribuídas e tolerantes a falhas. Esse *framework* trabalha com um modelo de atores (entidades que se comunicam entre si enviando mensagens de forma assíncrona). Toda comunicação é

orientada a eventos e cada ator também pode ser um supervisor de processos. O modelo de tolerância a falhas, de acordo com a documentação, suporta falhas do tipo Colapso (Crash). O supervisor de processos é responsável por verificar se os processos os quais está supervisionando estão respondendo e atua tentando reinicializá-los caso estejam apresentando algum problema.

Existem duas estratégias no *framework* Akka para tolerância de falhas: “OneForOne” e “AllForOne”. Na estratégia “OneForOne”, o supervisor (também chamado de Fault Handler) de processos tentará reiniciar somente o componente que colapsou. Na “AllForOne”, o supervisor irá reiniciar todos os componentes que estão sob sua supervisão (os que colapsaram e os que estão operacionais). A estratégia “AllForOne” deve ser utilizada quando um certo conjunto de componentes estão agrupados e possuem dependências entre si. Nesse caso, é necessário que todos sejam reiniciados para que o estado correto de todos os componentes possa ser garantido. A documentação desse *framework* deixa claro também que no mínimo duas máquinas diferentes são necessárias para a correta implementação do modelo de tolerância a falhas.

4.1.3.2 Comparação com o JFault

O *framework* Akka atua somente em processos que se comunicam de forma assíncrona, fornecendo um bom suporte para falhas de Colapso em componentes que enviam, recebem e processam mensagens. Da mesma forma que o framework Disal, também provém vários recursos para implementação (que pode ser feita na linguagem Java ou Scala) de processos que necessitam se comunicar de forma remota e tolerar possíveis falhas que possam ocorrer. Contudo, também possui sua linguagem própria, ou seja, o desenvolvedor precisa aprender a programar no *framework* e sobretudo escrever o código necessário para que os processos se comuniquem e tolerem falhas. O JFault, por sua vez, atua somente em processos síncronos e não contém uma linguagem própria (tudo é feito através de anotações). O JFault também possui um modelo nativo de suporte a tolerância a falhas mais robusto, todavia, menos flexível.

4.1.4 Framework JGroups

4.1.4.1 Descrição e Funcionamento

O *framework* JGroups [JGR14] auxilia no processo de criação de grupos de processos (*cluster*) de forma que um ou mais processos podem juntar-se em grupos para comunicar-se entre si de forma confiável. O *framework* suporta o mecanismo de comunicação assíncrona, tanto do tipo ponta-a-ponta *(one-to-one)* quanto *multicast* (*one-to-many*), sendo que esses modelos podem ser implementados utilizando-se tanto o protocolo UDP quanto TCP. O JGroups também possui funcionalidades referentes à tolerância a falhas, sendo capaz de reconhecer e remover do Cluster processos que apresentarem falhas de Colapso. O *framework* ainda suporta protocolos de ordenação, mecanismos de criptografia e compressão de mensagens.

4.1.4.2 Comparação com o JFault

O JGroups [JGR14] se tornou um framework bastante popular e amplamente utilizado para comunicação assíncrona entre processos distribuídos. Assim, como o *framework* Akka e o Disal, o JGroups também é mais flexível se comparado ao JFault, visto que possui uma linguagem específica e vários mecanismos de suporte a diferentes protocolos de rede (UPD e TCP), protocolos de ordenação, controle de fluxo, criptografia, entre outros. Contudo, assim como os *frameworks* vistos anteriormente, o JFault tem a vantagem de suportar mais tipos de falhas nativamente e é mais simples de ser utilizado em cenários onde pode ser aplicado. Em outras palavras, a aplicabilidade do JFault é mais simples, porém mais restrita, se comparado ao JGroups.

4.2 Trabalhos Similares sobre Arquiteturas de Meta-níveis

Zorzo et al. [ZOR96] realizaram em seus estudos um experimento utilizando Open C++ para demonstrar como os mecanismos de tolerância a falhas podem ser aplicados em um sistema remoto que ordena dados (*sorting*) utilizando-se arquiteturas de meta-níveis e protocolos de meta-objetos. De acordo com a proposta, quando o cliente invoca os métodos de ordenação, a chamada é “interceptada” pelo objeto de meta-nível que, por sua vez, executa a requisição no objeto de nível-base (que é efetivamente o que faz a ordenação) tratando as possíveis falhas que possam ocorrer durante o processo. O

experimento também avaliou o impacto no desempenho do sistema comparando os tempos de resposta das requisições sendo enviadas diretamente ao método de ordenação do objeto de nível-base e ao método do objeto reflexivo (meta-nível). A conclusão do trabalho é que as técnicas de reflexão utilizando meta-objetos fornecem uma separação mais clara de responsabilidades na arquitetura da aplicação, trazendo também uma pequena perda de desempenho, que é geralmente aceitável.

Xu et al. [XUJ96] implementaram em seu trabalho mecanismos de tolerância a falhas em uma aplicação de ordenação de dados utilizando técnicas usuais (em C++) e técnicas reflexivas baseadas em protocolos de meta-objetos (em Open C++) a fim de compará-las. Ambos experimentos utilizaram dois diferentes esquemas (*schemas*), ou seja, diferentes métodos de tolerância a falhas: RB (*Recovery Blocks*) [RAN75] e NVP (*N-Version programming*) [AVI85]. Os autores descrevem como os mecanismos de tolerância a falhas podem ser inseridos utilizando-se técnicas usuais de programação e técnicas de reflexão. Na técnica reflexiva é demonstrado como diferentes esquemas (*schemas*) de tolerância a falhas podem ser selecionados dinamicamente, visto que a implementação é separada do código da aplicação em si (dos mecanismos de ordenação, no caso), pois estão no meta-nível. Concluída a execução do experimento os autores completam afirmando que os métodos reflexivos de tolerância a falhas fornecem um *design* mais limpo e simples, contudo, adicionando uma determinada perda de desempenho (*overhead*) imposto pelas operações reflexivas que não são geralmente motivo de preocupação.

Killijan et al. [KIL98] desenvolveram um protocolo de meta-objetos que utiliza reflexão em tempo de compilação para construção de sistemas tolerantes a falhas chamado FT_MOP (Fault Tolerance MetaObject Protocol). O protocolo permite a execução e o controle de estado de objetos CORBA [BOL01] atuando no meta-nível para prover mecanismos de tolerância a falhas aos objetos da aplicação (nível-base). Em outras palavras, o protocolo fornece meios para anexar dinamicamente estratégias de tolerância a falhas em objetos CORBA a partir de meta-objetos. Os serviços da aplicação, assim como os mecanismos de comunicação em grupo e os serviços remotos CORBA, são considerados nessa arquitetura como serviços básicos (*basic layer*) ou simplesmente serviços a nível de aplicação (*application level*).

5 CONSIDERAÇÕES FINAIS

5.1 CARACTERÍSTICAS POTENCIALMENTE ORIGINAIS DO JFAULT

Os *frameworks* atuais estudados como parte dessa pesquisa são similares ao JFault no que se refere ao objetivo do mesmo: criar mecanismos para comunicação remota de processos (ou objetos) e tolerar possíveis falhas em que neles possam ocorrer. A originalidade da proposta está na quantidade de falhas suportadas nativamente e à baixa intrusão do *framework* no código da aplicação. Diferentemente dos *frameworks* atuais, no JFault não é necessário aprender como os mecanismos de tolerância a falhas e o processo de comunicação distribuída funcionam. Não é necessário se programar no *framework*, o código da aplicação é desenvolvido como se o acesso aos serviços de negócio fosse realizado utilizando-se objetos locais. As únicas alterações necessárias no código da aplicação são a inclusão das anotações que indicam os objetos que serão expostos remotamente (e quais falhas devem ser toleradas) e a alteração do cliente da aplicação para se utilizar o Stub gerado pelo *framework*. Cabe ressaltar também que o JFault é de aplicabilidade mais específica se comparado aos *frameworks* pesquisados.

No que se refere aos trabalhos pesquisados relativos a arquiteturas de meta-níveis, a proposta apresenta uma potencial inovação no que se refere à prática de criação dos objetos de meta-nível. A utilização das técnicas de reflexão combinadas às técnicas de compilação dinâmica de acesso programático permite que os objetos de meta-nível responsáveis pelo processo de tolerância a falhas possam ser criados de forma dinâmica, sem esforço de desenvolvimento e com baixo esforço de adaptação dos objetos à aplicação. A facilidade de adaptação se deve ao uso de reflexão estrutural, que permite que os objetos de meta-nível possam obter a mesma estrutura das classes de serviço, enquanto o artifício da compilação dinâmica é utilizado para compilar os objetos criados em tempo de execução.

5.2 CONTRIBUIÇÕES DO TRABALHO

A principal contribuição da pesquisa está em demonstrar uma visão de como as técnicas de reflexão – já bastante utilizadas em arquiteturas de meta-níveis – em conjunto com técnicas de criação e compilação de objetos em tempo de execução (*runtime*) podem

ser utilizadas para o provimento de mecanismos de tolerância a falhas de sistemas de forma pouco intrusiva. O trabalho também comprovou a viabilidade da solução, apresentando como as técnicas utilizadas podem ser aplicadas na prática em diversas linguagens de programação, com foco na tecnologia Java, linguagem na qual o *framework* foi implementado. Também foi demonstrado como a criação dinâmica de objetos de meta-nível específicos no lado cliente (Stubs) e no lado servidor (Proxies) pode ser utilizada para estabelecer um canal de implementação, que pode ser usado para o desenvolvimento de complexos mecanismos de suporte a requisitos não funcionais – e os meios para alcançá-los. No caso do *framework*, tolerância a falhas.

A biblioteca Java (.jar) do *framework* JFault, incluindo seu código fonte, pode ser acessada em [JFT14].

5.3 LIMITAÇÕES DO TRABALHO

1) O *framework* não possui nenhum mecanismo de controle distribuído de transações. Em outras palavras, em caso de falhas, parte-se da premissa de que os métodos das aplicações são idempotentes;
2) o *framework* não suporta serviços que mantenham estado. Parte-se da premissa de que sua utilização será focada em aplicações que mantêm estado na camada de persistência (geralmente um banco de dados);
3) o *framework* não possui nenhum tipo de controle de concorrência atuando no meio persistente (banco de dados). O controle de concorrência no nível persistente pode ser obtido com a utilização de outros *frameworks*, como o Hibernate [HIB14], no caso do Java, por exemplo;
4) a implementação do *framework* JFault é direcionada a aplicações JSE que necessitem tolerar falhas, possuindo seus serviços de negócio separados do cliente (modelo três camadas). Embora aplicações *desktop* (AWT/Swing/SWT) sejam o foco de atuação do *framework,* os serviços expostos por RMI podem potencialmente ser usados por qualquer cliente que suporte o protocolo, incluindo, mas não limitado a aplicações executadas em dispositivos móveis, por exemplo.

5.4 TRABALHOS FUTUROS

1) **Controle e replicação de estados**: o framework não possui atualmente nenhum controle de estado entre os serviços. O mesmo pode ser aprimorado para que os serviços mantenham estado e sincronizem-se entre si para garantir a consistência de suas réplicas;
2) **Interoperabilidade**: o protocolo utilizado para comunicação remota de serviços foi implementado utilizando-se RMI, protocolo suportado somente pela linguagem de programação Java. O *framework* pode ser melhorado com a implementação de um protocolo mais interoperável como, por exemplo, HTTP;
3) **Intrusividade**: por mais que a intrusividade do *framework* seja baixa, ainda assim existe a necessidade de adaptação dos Stubs tolerantes a falha ao código da aplicação. A criação de Aspectos [COS08] pode ser utilizada para eliminar completamente a intrusividade do *framework*. Um Aspecto pode ser criado junto com a Stub para redirecionar a requisição à interface remota através de um artifício de *cross-cutting* [COS08], removendo a necessidade de alterações no código da aplicação;
4) **Suporte a outros tipos de falhas**: o suporte do *framework* se restringe a falhas de Colapso e Tempo. O mesmo pode ser aprimorado para tolerar outros tipos complexos de falhas, como falhas Bizantinas, por exemplo.

REFERÊNCIAS BIBLIOGRÁFICAS

[AKK13] AKKA. “Building powerful concurrent & distributed applications more easily”. Capturado em: http://akka.io, Novembro 2014.

[AVI85] AVIZIENIS, Algirdas. “The N-Version Approach to Fault-Tolerant Software”. In: IEEE Trans. Software Eng., 1985, pp. 12, 1491-1501.

[AVI00] LAPRIE, Jean-Claude; RANDELL, Brian; AVIZIENIS, Algirdas. “Fundamental Concepts of Dependability”. In: Proceedings 3rd Information Survivability Workshop, 2000, pp. 2-5.

[AVI04] LAPRIE, J.; RANDELL, B.; AVIZIENIS, A; LANDWEHR, C. “Basic concepts and taxonomy of dependable and secure computing”. In: IEEE Trans. Dependable Security Computing 1, 2004, pp. 1,11–33.

[AWT14] WIKIPEDIA. “Abstract Window Toolkit and Swing”. Capturado em: http://en.wikipedia.org/wiki/Abstract_Window_Toolkit, Novembro 2014.

[BAR05] BARTH, J.; GOMI E. “A Meta-Level Architecture for Adaptive Applications”. In: Adaptive and Natural Computing Algorithms, 2005, pp. 329-332.

[BER98] ROBBEN, B.; JOOSEN, W.; MATTHIJS, F; VANHAUTE, B.; VERBAETEN P. “Building a Metalevel Architecture for Distributed Applications”. Technical report CW265, Department of Computer Science, K.U. Leuven, Belgium, 1998, 17p.

[BIE13] BIELY, M.; DELGADO P.; MILOSEVIC, Z.; SCHIPER, A. “Distal: A framework for implementing fault-tolerant distributed algorithms”. In: Annual IEEE/IFIP International Conference, 2013, pp.1,8, 24-27.

[BOL01] BOLTON, F. “Pure Corba”. Pearson Education, 2001, 944p.

[COL07] COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. “Sistemas Distribuídos: Conceitos e Projeto”. Bookman, 2007, 784p.

[COS08] COSTA, R. “Universo Java: Domine os Principais Recursos Oferecidos por Esta Linguagem de Programação”. Universo dos Livros, 2008, 272p.

[DAN05] DANTAS, M. “Computação Distribuída de Alto Desemprenho”. Axcel Books, 2005, 277p.

[ECL14] ECLIPSE. “SWT: Standard Widget Toolkit”. Capturado em: http://www.eclipse.org/swt/, Novembro 2014.

[HIB14] HIBERNATE. “Framework Hibernate”. Capturado em: http://hibernate.org/orm/, Outubro 2014.

[HPO13] HPOREMUHLE, A. “Sistemas Distribuídos: Conceitos e Projeto”. 5ª edição, Bookman, 2013, 1055p.

[JAL94] JALOTE, P. “Fault Tolerance in Distributed Systems”. Prentice Hall, Universidad of Michigan, 1994, 432p.

[JDB14] ORACLE. “Java Database Connectivity”. Capturado em: http://docs.oracle.com/Javase/tutorial/jdbc/overview/, Outubro 2014.

[JFT14] JFAULT. “Framework JFault (Biblioteca e Código Fonte)”. Capturado em: https://www.dropbox.com/l/NriwzgzR4ROTjESaPfAAMs, Dezembro 2014.

[JGR14] JGROUPS. “Framework Jgroups”. Capturado em: http://www.jgroups.org/, Outubro de 2014.

[JSE14] ORACLE. “JSE: Java Standard Edition”. Capturado em: http://www.oracle.com/technetwork/*Java*/*Java*se/tech/index.html, Outubro 2014.

[KAL09] KAMALAPUR, S.; DESHPANDE, N. “Distributed Systems”. Technical Publications, 2009, 567p.

[KOG08] KOGENT SOLUTIONS INC. “C# 2008 Programming: Covers .Net 3.5 Black Book”. Dreamtech Press, 2008, 1808p.

[KIL98] KILLIJIAN, M.; FABRE, J.; GARCIA, J; CHIBA, S. “Development of a Metaobject Protocol for Fault-Tolerance using Compile-Time Reflection”. In: OOPSLA’98 Workshop on Reflective Programming in C++ and Java, 1998, pp. 61-65.

[KRI01] KRISHNAMURTHY, S.; SANDERS, W.; CUKIER, M. “A Dynamic Replica Selection Algorithm for Tolerating Timing Faults”. In: Proceedings Int´l Conference Dependable Systems and Networks, 2001, pp. 107-116.

[KUR06] KURNIAWAN, B. “Java 6 New Features: a tutorial series”. Brainysoftware, 2006, 314p.

[LAB10] LABRADOR, M.; PEREZ, A; WIGHTMAN P. “Location-Based Information Systems: Developing Real-Time Tracking applications”. CRC Press, 2010, 287p.

[LEM07] LEMOS, R.; GACEK, C.; ROMANOVSKY, A. “Architecting Dependable Systems v4”, 2007, Springer, 433p.

[LOB09] LOBO, E. “Guia Prático de Engenharia de Software”. Universo dos Livros Editora, 2009, 128p.

[LOT10] LOTAR, A. “Como Programar com ASP.NET e C# - 2ª Edição: Dicas, truques, exemplos, códigos, conceitos, tutoriais”. Novatec Editora, 2010, 656p.

[MAN08] MANNINO, M. “Projeto, Devenvolvimento de Aplicações e Administração de Banco de dados, 3ª edição”. McGraw Hill Brasil, 2008, 717p.

[MAP87] MAES, P. “Concepts and Experiments in Computational Reflection”. In: Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages and Applications, 1987, pp. 147–155.

[MEN11] MENDES, D. “Programação Java em Ambientes Distribuídos”. Novatec Editora, 2011, 495 p.

[MIC14] MICROSOFT. “GenerateInMemory Property. Exemplos de compilação dinâmica em C#, C++ and VB”. Disponível em: http://msdn.microsoft.com/en-us/library/system.codedom.compiler.compilerparameters.generateinmemory(v=vs.110).aspx, Setembro 2014.

[MI214] MICROSOFT. “Compiler namespace. Pacote de compilação dinâmica do Framework .NET”. Disponível em: http://msdn.microsoft.com/en-us/library/system.codedom.compiler(v=vs.110).aspx, Setembro de 2014.

[MUL05] WIKIPEDIA. “Multicast”. Disponível em: http://pt.wikipedia.org/wiki/Multicast, Outubro de 2014.

[NIE05] NIEMEYER, P.; KNUDSEN, J. “Learning Java - 3rd edition”. O´reilly Media Inc, 2005, 980p.

[PFL03] PFLEENGER, C.; PFLEENGER, S. “Security in Computing”. Prentice Hall Professional, 2003, 746p.

[RAN75] RANDELL, B. “System Structure for Software Fault Tolerance”. IEEE Trans. Software Eng., 1975, vol. SE-1 no. 2, pp. 220-232.

[RMI11] SMATANIK, V. “Java RMI–Software architecture document”. Department of Information and Computing Sciences, 2011, 44p.

[ROM01] ROMANOVSKY, A. “Advances in Exception Handling Techniques”. Springer, 2001, 288p.

[SAM11] SAMPAIO, C. “Java Enterprise Edition 6 - Desenvolvendo Aplicações Corporativas”. Brasport, 2011, 276p.

[SCA14] SCALA. “Linguagem Scala”. Capturado em: http://www.scala-lang.org/, Dezembro de 2014.

[SCO08] SCOTT, M. “Programming Language Pragmatics”. Morgan Kaufmann, 2008, 944p.

[SHA07] SHANKAR, P.; SRIKANT, Y. “The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition”. CRC Press, 2007, 784p.

[SOC13] ORACLE. “Java Sockets”. Capturado em: http://docs.oracle.com/javase/tutorial/networking/sockets/, Novembro de 2014.

[TAN06] TANENBAUM, A.; STEEN, M. “Distributed Systems: Principles and Paradigms – 2nd Edition”. Pearson Prentice Hall, 2006, 686p.

[TOE12] TOEROE, M.; TAM, F. “Service Availability”. John Wiley & Sons, 2012, 336p.

[TOM03] THOMAS, V.; MCMULLEN, A; GRABA, L. “FT-Java: A Java-Based Framework for Fault-Tolerant Distributed Software”. Springer Berlin Heidelberg, 2003, pp. 899-911.

[VAL06] VALPEREIRO, F; PINHO, L. “POSIX Trace Based Behavioural Reflection”. In: 11th Ada-Europe International Conference on Reliable Software Technologies, 2006, pp. 5-9.

[VER11] VERMA, A; KUMAR, M. “Dependability of Networked Computer-based Systems”. Springer, 2011, 185p.

[XUJ96] XU, J.; RANDELL, B.; ZORZO, A. “Implementing Software Fault Tolerance in C++ and OpenC++: an object-oriented and reflective approach”. In: Proceedings of the International Workshop on Computer-Aided Design, Test and Evalution for Dependability, 1996, pp. 224-229.

[ZOR96] ZORZO, A.; XU, J.; RANDELL, B. “Experimental Evaluation of Fault-Tolerant Mechanisms for Object-Oriented Software”. In: Proceedings of XXIII Integrated Seminars on Software and Hardware, 1996, pp. 457-468.

[ZUK06] ZUKOWSKI, J. “Java 6 Plataform Revealed”. Apress, 2006, 240 p.

APÊNDICE A

Principais métodos da classe JFaultInterfaceServices

```
1.  import static java.lang.System.out;
2.  public class JFaultInterfaceServices {
3.      public static JFaultProxyInterfaceVo generateInterface(String className, JFaultMethodVo[]
4.          methods) throws JFaultGenericException{
5.      JFaultProxyInterfaceVo interface2Expose = null;
6.      try {
7.          // defining inteface name
8.          String interfaceName = className+JFaultConstants.JFAULT_RMI_INTERFACE_SUFIX;
9.
10.         // creating interface Object
11.         interface2Expose = new JFaultProxyInterfaceVo();
12.         interface2Expose.setInterfaceName(interfaceName);
13.         interface2Expose.setMethods(methods);
14.         // generate source that will be encapsulated within the object
15.         interface2Expose.generateSourceCode();
16.         JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Creating runtime resource...");
17.         // creating Java file to be compiled
18.         File interfaceFileSource = JFaultIOServices.createFile(
19.             interface2Expose.getSourceCode().toString(), interfaceName,
20.            JFaultConstants.COMPILE_DIRECTORY_VALUE);
21.         interface2Expose.setSourceFile(interfaceFileSource);
22.     }catch (Exception e) {throw new JFaultGenericException(e.getMessage()); }
23.     return interface2Expose;
24.   }
25. }
```

APÊNDICE B

Principais métodos da classe JFaultRMIServices

```
1.  public class JFaultRMIServices {
2.      public static void startRegistry(int RMIPortNum) throws JFaultGenericException{
3.          try {
4.              JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "STARTING RMI REGISTRY");
5.              Registry registry = LocateRegistry.getRegistry(RMIPortNum);
6.              registry.list( );
7.          } catch (Exception e) {
8.              try { // No valid registry at that port, try to create it
9.                  Registry registry = LocateRegistry.createRegistry(RMIPortNum);
10.                 FaultLog.print(JFaultConstants.LOG_LEVEL_INFO, " RMI Registry Started");
11.             } catch (Exception e2) { throw new JFaultGenericException(e.getMessage());
12.             }
13.         }
14.     }
15.     public static void exportObject(String url, Object obj) throws JFaultGenericException{
16.         try {
17.             Naming.rebind(url, (Remote)obj);
18.             JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, " Exposing Object "+obj+" in RMI
19.                 Registry to be located via URL: "+url+"....");
20.         } catch (Exception e) { throw new JFaultGenericException(e.getMessage());
21.         }
22.     }
23. }
```

APÊNDICE C

Principais métodos da classe JFaultProxyServices

```
1.  public class JFaultProxyServices {
2.      public static JFaultProxyClassVo generateProxy(String classWithPackage, String className,
3.              String interfaceName, JFaultMethodVo[] methods) throws JFaultGenericException{
4.          JFaultProxyClassVo proxyClass = new JFaultProxyClassVo();
5.          String proxyClassName = className+ JFaultConstants.JFAULT_RMI_PROXY_SUFIX;
6.          try {
7.              JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Gathering Proxy information...");
8.              proxyClass.setClassName(proxyClassName);
9.              proxyClass.setRemoteInterface(interfaceName);
10.             proxyClass.setInnerClassName(className);
11.             proxyClass.setMethods(methods);
12.             proxyClass.setClassWithPackage(classWithPackage);
13.             // genering proxy code
14.             proxyClass.generateSourceCode();
15.             JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Creating runtime resource...");
16.             JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, ">> Proxy Classname:
17.                 "+proxyClassName);
18.             File proxyFileSource = JFaultIOServices.createFile(proxyClass.getSourceCode().toString(),
19.                 proxyClassName, JFaultConstants.COMPILE_DIRECTORY_VALUE);
20.                 proxyClass.setSourceFileCode(proxyFileSource);
21.         }catch (Exception e) {throw new JFaultGenericException(e.getMessage());
22.         }
23.         return proxyClass;
24.     }
25. }
```

APÊNDICE D

Principais métodos da classe JFaultStubServices

```
1.  public class JFaultStubServices {
2.      public static void generateStub(Class class2GenerateStub, JFaultConfigurationVo conf){
3.          try {
4.              boolean compilationResult = false;
5.              // gathering class components and behavior
6.              JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "** META-LEVEL CLASSES ***");
7.              String className = class2GenerateStub.getSimpleName();
8.              // gathering class structure via Reflection
9.              JFaultMethodVo[] methods = JFaultReflectionServices.getMethods(class2GenerateStub);
10.             // building RMI interface to be exposed in runtime
11.             JFaultProxyInterfaceVo rmiInterface =
12.                 JFaultInterfaceServices.generateInterface(className, methods);
13.
14.             JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Compiling Stub interface...");
15.             compilationResult=JFaultJvmServices.compileWithExtraClasspath(
16.                 rmiInterface.getSourceFile(),
17.                 new File(JFaultConstants.COMPILE_DIRECTORY_VALUE),
18.                 rmiInterface.getInterfaceName());
19.
20.              if (compilationResult) {
21.                  JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, " Compiled sucessfully!");
22.                  } else {
23.                      JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Compilation Failted!");
24.                  }
25.
26.             // Generation Java Source code for developer analises if necessary
27.             if (conf.isGenenerateSourceCode()) {
28.                 JFaultIOServices.createFile(rmiInterface.getSourceCode().toString(),
29.                 rmiInterface.getInterfaceName(), conf.getJfaultMetaLevelObjectsSourcePath());
30.             }
31.             JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Generating Client stub...");
32.             JFaultStubVo stub = generateStub(className, methods,
33.                 getStubFailures(class2GenerateStub),
34.                 getHosts(class2GenerateStub),getPort(class2GenerateStub),
```

```
35.                 getRemoteReferenceName(class2GenerateStub));
36.             JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Compiling Stub class...");
37.             // compiling interface built by JFault
38.             compilationResult =
39.                  JFaultJvmServices.compileWithExtraClasspath(stub.getSourceFileCode(),
40.                  new File(JFaultConstants.COMPILE_DIRECTORY_VALUE),
41.                  stub.getStubClassName());
42.              if (compilationResult) {
43.                 JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, " Compiled sucessfully!");
44.             } else {
45.                 JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Compilation failed!");
46.             }
47.             if (conf.isGenenerateSourceCode()) {
48.                 JFaultIOServices.createFile(stub.getSourceCode().toString(),
49.                     stub.getStubClassName(), conf.getJfaultMetaLevelObjectsSourcePath());
50.             }
51.         }catch (Exception e) {
52.             e.printStackTrace();
53.         }
54. }
```

APÊNDICE E

Principais métodos da classe JFaultManager

```
1.  public class JFaultManager{
2.      public static void exposeClass4RemoteAccess(Class class2Expose){
3.          try {
4.              JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Generating RMI interface...");
5.              String className = class2Expose.getSimpleName();
6.              String classWithPackage = class2Expose.getName();
7.              JFaultMethodVo[] methods = JFaultReflectionServices.getMethods(class2Expose);
8.              // building RMI interface to be exposed in runtime
9.              JFaultProxyInterfaceVo rmiInterface =
10.                 JFaultInterfaceServices.generateInterface(className, methods);
11.             File compilationDir= new File(JFaultConstants.COMPILE_DIRECTORY_VALUE);
12.             JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Compiling resources...");
13.             JFaultJvmServices.compile(rmiInterface.getSourceFile(),
14.                 compilationDir, rmiInterface.getInterfaceName());
15.             JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Generating RMI Proxy...");
16.             JFaultProxyClassVo rmiProxyClass =
17.                 JFaultProxyServices.generateProxy(classWithPackage, className,
18.              rmiInterface.getInterfaceName(), methods);
19.             JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Compiling resources...");
20.             JFaultJvmServices.compile(rmiProxyClass.getSourceFileCode(),
21.                 compilationDir,
22.                  rmiProxyClass.getClassName());
23.             File binDir= new File(JFaultConstants.BIN_DIRECTORY_VALUE);
24.             Object obj =
25.                  JFaultJvmServices.createInstance("jfaultRuntime."+rmiProxyClass.getClassName(),
26.                   binDir);
27.              // gathering port and name to expose via RMI
28.             int rmiPort = ((JFaultRemote)
29.                 class2Expose.getAnnotation(JFaultRemote.class)).remoteReferencePort();
30.             String rmiName = ((JFaultRemote)
31.                 class2Expose.getAnnotation(JFaultRemote.class)).remoteReferenceName();
32.             JFaultRMIServices.startRegistry(rmiPort);
33.             String registryURL = "rmi://localhost:"+rmiPort+"/"+rmiName;
34.             JFaultRMIServices.exportObject(registryURL, obj);
```

```
35.
36.             JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Object
37.                     "+rmiProxyClass.getClassName()+"" + " exposed for remotely at: "+registryURL);
38.         } catch (Exception e) {
39.             e.printStackTrace();
40.         }
41. }
```

APÊNDICE F

Principais métodos da classe JFaultJvmServices

```
1.  public class JFaultJvmServices{
2.      public static boolean compileWithExtraClasspath(File sourceFile, File directory,
3.              String className) {
4.          boolean compilationResult = false;
5.              try {
6.                  // Compile source file in runtime
7.                  JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO,
8.                      "Compiling source class " + className);
9.                  JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
10.                 String classpath = System.getProperty("java.class.path");
11.                 JFaultClassLoader jFaultClassLoader = (JFaultClassLoader) ClassLoader
12.                     .getSystemClassLoader();
13.                 URL[] urls = jFaultClassLoader.getURLs();
14.                 classpath = classpath + ";";
15.                 String url = "";
16.                 for (int i = 0; i < urls.length; i++) {
17.                     url = urls[i].getPath();
18.                     if (url.startsWith("/"))
19.                         url = url.substring(1);
20.                     classpath = classpath + url + ";";
21.                  }
22.                 classpath = classpath.replaceAll(Matcher.quoteReplacement("/"),
23.                 Matcher.quoteReplacement("\\"));
24.                 JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO,
25.                     "Runtime Compilation Classpath " + classpath);
26.                 List<String> optionList = new ArrayList<String>();
27.                 optionList.addAll(Arrays.asList("-classpath", classpath));
28.                 StandardJavaFileManager sjfm = compiler.getStandardFileManager(
29.                     null, null, null);
30.                  Iterable fileObjects = sjfm.getJavaFileObjects(sourceFile);
31.                  JavaCompiler.CompilationTask task = compiler.getTask(null, null,
32.                      null, optionList, null, fileObjects);
33.                  compilationResult = task.call();
34.                  sjfm.close();
```

```
35.          } catch (Exception e) {e.printStackTrace(); }
36.       return compilationResult;
37. }
```

Principais métodos da classe JFaultReflectionServices

```
1.  public class JFaultReflectionServices{
2.     public static JFaultMethodVo[] getMethods(Class c) throws JFaultGenericException{
3.        JFaultMethodVo[] jfMethods = null;
4.        JFaultMethodVo jfMethod = null;
5.        try {
6.           Method[] methods = c.getDeclaredMethods();
7.           if (methods != null){
8.              jfMethods = new JFaultMethodVo[methods.length];
9.              for (int i = 0; i < methods.length; i++) {
10.                Class<?>[] pType  = methods[i].getParameterTypes();
11.                Type[] gpType = methods[i].getGenericParameterTypes();
12.                for (int j = 0; j < pType.length; j++) {
13.                   jfMethod = new JFaultMethodVo();
14.                   jfMethod.setMethodName(methods[i].getName());
15.                   jfMethod.setReturnType(methods[i].getReturnType());
16.                   jfMethod.setParameterTypes(methods[i].getParameterTypes());
17.                   jfMethod.setModifier(getModifierAsString(methods[i].getModifiers()));
18.                   jfMethods[i] = jfMethod;
19.                }
20.           }
21.       } catch ( Exception exc ) {
22.          exc.printStackTrace();
23.          JFaultGenericException jfe = new JFaultGenericException(exc.getMessage());
24.          throw jfe;
25.       }
26.       return jfMethods;
27.    }
28.    public static Object createInstance(String className, File directory) {
29.       Object instance = null;
30.       try {
```

```
31.             // creating instance in runtime
32.             JFaultLog.print(JFaultConstants.LOG_LEVEL_INFO, "Creating instance " + className);
33.             URLClassLoader classLoader = URLClassLoader
34.                 .newInstance(new URL[] { directory.toURI().toURL() });
35.             Class<?> cls = Class.forName(className, true, classLoader);
36.             instance = cls.newInstance();
37.         } catch (Exception e) {
38.             e.printStackTrace();
39.         }
40.         return instance;
41.     }
42. }
```

APÊNDICE G

Principais métodos da classe JFaultStubVo

```
1.  public class JFaultStubVo{
2.  private void addHeartBeatThread(StringBuffer sb) {
3.      addLineBreak(sb);
4.      append("private class HeartBeatChecker extends Thread {");
5.      append("HeartBeatChecker(String name) {");
6.      append("super(name);");
7.      append("}");
8.      addLineBreak(sb);
9.      append("public void run() {");
10.     // all necessary code to verify the state of the services and gather monitoring information
11.     append("}");
12. }
```