UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

KARINA KOHL SILVEIRA

Uma Estratégia Baseada em Programação Orientada a Aspectos para Injeção de Falhas de Comunicação

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Profa. Dra. Taisy Silva Weber
Orientadora

Porto Alegre, junho de 2005

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Silveira, Karina Kohl

Uma Estratégia Baseada em Programação Orientada a Aspectos para Injeção de Falhas de Comunicação / Karina Kohl Silveira – Porto Alegre: Programa de Pós-Graduação em Computação, 2005.

75 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2005. Orientador: Taisy da Silva Weber.

1.Tolerância a Falhas. 2.Injeção de Falhas 3.Programação Orientada a Aspectos. I. Weber, Taisy da Silva. II. Título.

“Stare at yourself
The power can be found
Make a wish
Straight to the skyline
Make your dreams come true
The best bet is yourself”
Hibria

One ring to rule them all.
One ring to find them.
One ring to bring them all.
An in the Darkness bind them”
J.R.R. Tolkien

AGRADECIMENTOS

A Deus e seus Anjos! Que fizeram que minhas lágrimas servissem para limpar os caminhos nas horas que mais tive vontade de desistir. Agradeço por me darem a força pra concluir mais essa etapa.

A minha mãe, dona Noeli, que abriria mão de tudo para me dar suporte nessa caminhada! A pessoa que sorriu, chorou, se alegrou e sofreu comigo mesmo não entendendo um "bit" do que eu estava falando. Mãe, te amo incondicionalmente!!!

Ao meu namorado, Abel. Que depois de agüentar meu estresse do trabalho de graduação, teve que agüentar também o estresse da dissertação. Obrigada por ser meu companheiro, amigo, professor, psicólogo, confidente e antes de qualquer coisa, um namorado maravilhoso! Te amo muito!

Aos amigos! Sem eles não somos nada. Aos grandes amigos que me acompanham desde a graduação: Daniel Gaspary (Chewieeeeee!!!) e Guilherme Drehmer (Boooooo!!!). Que mesmo morando na mesma cidade, os compromissos fazem que a comunicação seja a maior parte do tempo por e-mails, icq ou msn. Outros nomes que não poderiam nunca deixar de constar aqui: Gustavo Hexsel (que me ensinou a gostar de J.R.R. Tolkien e tem sua parcela de culpa pelo meu gosto por Java e joguinhos ;)), Vinicius Pazutti Correia, Ana Gabriela (futura bibliotecária com um futuro brilhante pela frente) e Luciana Clipes, Eduardo Bobsin, Miguel Meggar, Michelle Leonhardt (pelo reencontro após vários anos, pela surpresa de estarmos seguindo o mesmo caminho), Rafael Kunst (pelos papos pelo icq e o compartilhamento de escrita de artigos :)). Vocês são especiais! Um "Viva a internet"! Graças a ela consigo manter todos vocês por perto!

Aos amigos da família Hibria (meninos e meninas)! Obrigada pelos momentos de alegria que me proporcionaram e que muitas vezes me fizeram "descansar a cabeça" para continuar com mais vontade!!

Aos amigos do *hard & metal*. UP THE BANGERS!

Ao CPD da UFRGS, que me liberou por 20hs durante todo o ano para que eu pudesse me dedicar também ao mestrado.

A minha orientadora Taisy Weber. Que me aceitou como orientanda e deixou que eu levasse adiante a minha loucura de orientação a aspectos. Obrigada pelo voto de confiança e pela orientação.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API	Applications Programming Interface
FITA	Fault Injection Tool based on Aspects
FICTA	Fault Injection Communication Tool based on Aspects
JRE	Java Runtime Environment
JVM	Java Virtual Machine
JVMDI	Java Virtual Machine Debug Interface
JVMTI	Java Virtual Machine Tool Interface
POA	Programação Orientada a Aspectos
POO	Programação Orientada a Objetos
QoS	Quality of Services
UDP	User Datagram Protocol
XML	Extensible Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A injeção de falhas permite acelerar a ocorrência de erros em um sistema para que seja possível a validação de seu comportamento sob falhas, assim como a avaliação do impacto dos mecanismos de detecção e remoção de erros no desempenho do sistema. Abordagens que facilitem o desenvolvimento de injetores vêm sendo buscadas com empenho, variando desde a inserção de injetores no *kernel* do sistema operacional até o uso de reflexão computacional para aplicações orientadas a objetos.

Este trabalho explora os recursos da Programação Orientada a Aspectos como estratégia para a criação de ferramentas de injeção de falhas. A Programação Orientada a Aspectos tem como objetivo a modularização de interesses transversais, isto é, interesses que atravessam as unidades naturais de modularização. A injeção de falhas possui um comportamento que abrange os diversos módulos da aplicação alvo, afetando métodos que são executados em diversas classes em diversos pontos da aplicação. Desta forma, a injeção de falhas pode ser encapsulada sob a forma de aspectos.

Para demonstrar a validade da proposta apresentada foi desenvolvida a ferramenta FICTA – *Fault Injection Communication Tool based on Aspects*. O objetivo é a validação de aplicações Java distribuídas, construídas sobre o protocolo UDP e que implementem mecanismos de tolerância a falhas em protocolos de camadas superiores. A importância de instrumentar um protocolo de base é justificada pelo fato da necessidade de validar aplicações, *toolkits* e *middlewares* que implementem tolerância a falhas em camadas superiores, logo, esses protocolos devem lidar corretamente com as falhas de mais baixo nível. A ferramenta abrange falha de colapso e omissão de mensagens do protocolo UDP.

O uso de Programação Orientada a Aspectos na construção de FICTA resultou em uma ferramenta altamente modular, reusável e flexível, que pode ser facilmente inserida e removida da aplicação alvo, sem causar intrusividade espacial no código fonte da aplicação.

Palavras-Chave: Tolerância a Falhas, Injeção de Falhas, Programação Orientada a Aspectos.

FICTA – A Fault Injection Communication Tool Based on Aspect Oriented Programming

ABSTRACT

The fault injection allows us to accelerate the occurrence of failures in a system so that it is possible to validate its behavior under faults, as well as the evaluation of the impact on the mechanisms of detection and removal of failures in the performance of the system. The approaches that may facilitate the development of injectors have been searched with effort, varying from the insertion of injectors in the kernel of the operational system up to the computational reflection for object oriented applications.

This work explores the resources of the Aspect Oriented Programming as a strategy to create tools of fault injection. The Aspect Oriented Programming has as its goal the modularization of the crosscutting concerns, that is to say the interests that cross the natural units of modularization. The fault injection has a behavior that covers the various modules of the target application, affecting methods that are executed in several classes of several areas of the application. Thus, the Fault Injection may be encapsulated under the form of aspects.

To demonstrate the worthiness of the presented proposal, a tool called FICTA - Fault Injection Communication Tool based on Aspects, has been developed. The aim is to validate Java distributed applications built under the UDP protocol so that the fault tolerance mechanisms can be implemented in upper layers. The importance of instrumentate a protocol of base is justified by the necessity of validating applications, toolkits and middlewares that implement fault tolerance in upper layers, then, these protocols must deal correctly with the lower level faults. The tool covers crash and message omission faults of the UDP protocol.

The use of Aspect Oriented Programming in the construction of FICTA resulted in a tool highly modular, reusable and flexible that may be easily inserted and removed from the target application, without causing spatial intrusiveness in the source code of the application.

Keywords: Fault Tolerance, Fault Injection, Aspect Oriented Programming.

1 INTRODUÇÃO

Esse capítulo introdutório tem como objetivo apresentar o contexto do trabalho, assim como as principais motivações e objetivos que se pretende alcançar ao final desse texto.

1.1 Contexto

Aplicações distribuídas estão sujeitas as falhas internas de cada computador participante do sistema distribuído e também a falhas de canais de comunicação que afetam o sistema de troca de mensagens. Para que essas aplicações sejam utilizadas em sistemas de alta disponibilidade, é necessária a implementação de mecanismos de detecção e remoção de erros que permitam o funcionamento correto do sistema mesmo na ocorrência falhas. Assegurar que um sistema distribuído satisfaz a sua especificação é particularmente importante para aplicações com restrições de dependabilidade. Uma etapa fundamental no desenvolvimento de sistemas tolerantes a falhas é a fase de validação, na qual é verificado o comportamento do sistema em vários cenários de falha. Não verificar os mecanismos de detecção e remoção de erros pode permitir a ocorrência de comportamentos inesperados e errôneos na fase operacional da aplicação (SILVEIRA, WEBER, 2005).

Em alguns casos não é possível o teste de um sistema em seu ambiente de utilização. Nestes casos é importante que exista uma forma de validar a implementação e, principalmente, as técnicas de tolerância a falhas utilizadas no sistema em questão. A injeção de falhas é uma das possibilidades para essa validação (CARREIRA; LEITE; WEBER, 1999).

A injeção de falhas é uma técnica de validação experimental usada no teste de mecanismos de tolerância a falhas e pode ser definida como a introdução intencional de falhas em um sistema alvo para observar seu comportamento (ARLAT *et al,* 1990).

Este trabalho utiliza recursos de Programação Orientada a Aspectos – POA (KICKZALES, 1997) como estratégia para a validação de aplicações distribuídas escritas em Java que tenham sido construídas sobre o protocolo UDP e que ofereçam mecanismos de detecção e remoção de erros em protocolos de camadas superiores. A estratégia utiliza os recursos de POA para realizar a interceptação e instrumentação das primitivas de envio e recepção de pacotes UDP simulando falhas de comunicação e

colapso. As falhas injetadas permitem verificar se a aplicação que está sendo validada se comporta conforme especificado na presença de falhas, identificando problemas específicos, como a detecção de erros de projeto e implementação além de identificar violações de especificação dos protocolos.

1.2 Motivação

Sistemas Distribuídos vêm sendo amplamente usados em vários tipos de aplicações. Como exemplos, podem ser citados sistemas que dão suporte a agregados de computadores (tanto os de alto desempenho, quanto os de alta disponibilidade), *grids*, replicação de dados, contêineres para servidores de aplicação e servidores *web*, etc. Normalmente, os tipos de aplicações citadas necessitam um alto grau de dependabilidade. É desejável que os dados e o processamento oferecidos por essas aplicações estejam disponíveis a maior quantidade de tempo possível, com intervalos de indisponibilidade tendendo a zero. Além disso, as aplicações devem ser confiáveis e os dados fornecidos devem ser consistentes.

O grupo de Tolerância a Falhas do Instituto da Informática da UFRGS vem buscando definir e desenvolver técnicas de injeção de falhas para a validação de mecanismos de tolerância a falhas utilizados em ambientes que exigem alta disponibilidade. É esperado que as soluções encontradas facilitem o trabalho de validação dos mecanismos, sendo de ampla utilização e reusabilidade. Também é esperado que as soluções causem pouca intrusividade nas aplicações, fortalecendo a reusabilidade, tornando o procedimento de testes mais rápido e oferecendo retornos em várias fases do ciclo de vida das aplicações, desde seus projetos, desenvolvimento, até o ambiente final de produção.

Exemplos dos trabalhados desenvolvidos pelo grupo de tolerância a falhas são as ferramentas FIONA e a extensão da ferramenta Jaca (MARTINS; RUBIRA; LEME, 2002) para falhas de comunicação. FIONA (SILVA et al, 2004-b) utiliza JVMTI (JVMTI, 2005) e a extensão de Jaca para falhas de comunicação (SILVA et al, 2004-a) utiliza reflexão computacional para realizar a instrumentação das classes que implementam as primitivas de comunicação de mais baixo nível da linguagem Java. Ambas adotam estratégias de implementação que visam minimizar a intrusividade das ferramentas de injeção de falhas nas aplicações sob teste. As ferramentas FIONA e a extensão da ferramenta Jaca influenciaram fortemente o desenvolvimento desse trabalho. Porém, apesar do desenvolvimento no mesmo grupo de pesquisa, este trabalho foi desenvolvido em paralelo e sem interferência das soluções adotadas para a implementação das duas ferramentas citadas. As diferenças entre as ferramentas serão citadas no capítulo 2.

As linguagens para POA são baseadas no conceito de modularização de interesses transversais, possibilitando a interceptação e modificação de construtores, métodos e campos de um sistema, em tempo de execução, sem que exista a necessidade de alteração do código fonte original da aplicação. Isso se torna uma forte motivação para a utilização de orientação a aspectos no desenvolvimento de uma nova estratégia de injeção de falhas.

1.3 Objetivos

O principal objetivo do trabalho é explorar uma nova estratégia para a construção de injetores de falhas, utilizando recursos de POA. O trabalho pretende mostrar como os recursos oferecidos por uma nova tecnologia podem auxiliar na solução de um problema clássico (como o da injeção de falhas), facilitando e ampliando a utilização de injeção de falhas como técnica de validação de sistemas confiáveis.

Ao longo do trabalho é apresentado POA como um paradigma válido para a criação de uma estratégia de validação experimental e que seus conceitos e mecanismos são muito úteis no desenvolvimento de injetores de falhas. É mostrado como conceitos de POA se relacionam com os conceitos de injeção de falhas e como podem ser utilizados em benefício dessa forma de validação experimental. Através desse mapeamento de conceitos é apresentada uma estratégia de validação assim como uma versão inicial da ferramenta chamada FICTA (*Fault Injection Communication Tool based on Aspects*) que tem como objetivo ilustrar e validar a estratégia proposta neste trabalho.

1.4 Resultados Alcançados

Os principais resultados atingidos pelo trabalho foram a apresentação de uma estratégia orientada a aspectos para injeção de falhas em aplicações distribuídas e também um protótipo inicial da ferramenta FICTA, desenvolvido com o objetivo de demonstrar a estratégia apresentada.

FICTA injeta falhas em uma classe da API base da linguagem Java, `java.net.DatagramSocket`, responsável por enviar pacotes UDP pela rede. Dessa forma, FICTA é capaz de validar qualquer aplicação que utilize o protocolo UDP como base e implemente confiabilidade em camadas superiores.

O trabalho resultou em dois artigos aprovados. O primeiro em um evento regional, a 2º Escola de Redes de Computadores, em 2004, onde foram apresentados os primeiros resultados dos estudos do uso de POA para injeção de falhas, além de um protótipo inicial para FICTA (que foi chamado apenas de FITA). Esse primeiro protótipo realizava a instrumentação de protocolos do sistema de comunicação de grupo chamado JGroups (BAN, 1998) e não oferecia flexibilidade para ser utilizado em outros ambientes.

O segundo artigo foi aprovado no VI Workshop de Testes e Tolerância a Falhas, em 2005, evento nacional que ocorre em paralelo com o Simpósio Brasileiro de Redes de Computadores. Nesse artigo são apresentados resultados bem mais próximos dos resultados apresentados neste trabalho. A ferramenta já tem o nome de FICTA e oferece maior flexibilidade e reusabilidade, obtida através do uso de arquivos de configuração (texto e XML (XML, 2005)).

Os resultados alcançados podem ser visualizados através dos experimentos apresentados no capítulo 6 deste texto, onde FICTA é utilizada para demonstrar o comportamento sob falhas do JGroups, porém, dessa vez, instrumentando classes base de Java e não protocolos do próprio JGroups (o que também seria possível, pela

configuração do arquivo XML que define quais métodos serão instrumentados pela ferramenta).

1.5 Estrutura do Texto

O capítulo 2 apresenta os principais conceitos relacionados a injeção de falhas, modelos de falhas, além de apresentar algumas ferramentas e suas principais características. O capítulo 3 apresenta os principais conceitos de Programação Orientada a Aspectos. O capítulo 4 inicia a apresentação da estratégia proposta pelo trabalho, mostrando como POA pode ser utilizada em injeção de falhas, vantagens e desvantagens da abordagem. O capítulo 5 apresenta a ferramenta FICTA, sua arquitetura, modelo de falhas e implementação, validando os conceitos do capítulo 4. O capítulo 6 apresenta resultados dos testes efetuados com a ferramenta, demonstrando efetivamente que a estratégia pode ser utilizada com vantagens. O capítulo 7 apresenta as considerações finais do trabalho e trabalhos futuros que podem ser realizados.

2 INJEÇÃO DE FALHAS

A injeção de falhas é uma das técnicas usadas para a validação experimental de sistemas. Através dessa técnica, são introduzidas falhas, de forma controlada, em um sistema alvo e a resposta do sistema é monitorada sob essa condição. O objetivo é testar a eficiência de mecanismos de tolerância a falhas e avaliar a segurança de operação do sistema, oferecendo algum retorno ao ciclo de desenvolvimento (IYER, 1995).

A injeção de falhas é uma técnica de validação usada no teste de mecanismos de tolerância a falhas e pode ser definida como a introdução intencional de falhas em um sistema alvo para observar seu comportamento (ARLAT *et al*, 1990). Acelerar a ocorrência de falhas é essencial para validar propriedades como disponibilidade ou confiabilidade de um sistema assim como para avaliar o impacto dos mecanismos de detecção e recuperação no seu desempenho, pois evita a espera por falhas reais, aleatórias e imprevisíveis, na determinação do seu comportamento sob falhas. Nesta técnica, não são provocadas falhas reais de hardware, mas os efeitos de tais falhas são emulados por software.

Para testar um sistema, deve-se forçar o mesmo a determinados estados para garantir que caminhos de execução específicos sejam seguidos. Por outro lado, pode-se forçar o sistema a um comportamento que, sob condições normais, não aconteceria. Estas são as principais motivações para o uso de técnicas de validação (DAWSON; JAHANIAN, MITTON, 1996).

O objetivo da validação é garantir a confiança na capacidade do sistema em fornecer o serviço especificado, ou seja, fazer com que o sistema apresente a característica de dependabilidade (LAPRIE, 1985).

A abordagem de injeção de falhas pode ser aplicada não apenas durante as fases de concepção e projeto do sistema, mas também nas fases de prototipação e operacional. Para que os experimentos tenham sentido é necessário conhecer a especificação dos sistemas para tolerar falhas, incluindo seus mecanismos de detecção e recuperação, para que seja possível especificar ferramentas para injetar falhas, criar erros e defeitos e monitorar seus efeitos (HSUEH; TSAI; IYER, 1997).

Os resultados obtidos através das experiências de injeção de falhas indicam se a aplicação atende à especificação, ou seja, se realmente mascara ou recupera-se das falhas a que se propôs na fase de projeto. Nesse sentido, pode-se afirmar que a técnica

acelera a ocorrência das falhas em um determinado sistema. Com isso, ao invés de esperar pela ocorrência espontânea das falhas, pode-se introduzi-las intencionalmente, controlando, por exemplo, o tipo, a localização e a duração das falhas.

2.1 Classificação da Injeção de Falhas

Técnicas de injeção de falhas têm sido amplamente utilizadas para avaliar as características de dependabilidade dos sistemas ou simplesmente para validar determinados mecanismos de manipulação de falhas.

A escolha entre injeção de falhas por *hardware* ou *software* é dependente do tipo de falhas que se deseja inserir e o esforço requerido para que sejam criadas. Por exemplo, caso o interesse sejam falhas do tipo *stuck-at* (que forçam um determinado valor em determinado ponto de um circuito), um injetor em *hardware* é preferível, pois é possível controlar a localização da falha. A injeção de falhas permanentes usando métodos de *software* ou podem incluir grande atraso ou ser impossível, dependendo da falha. Caso o interesse seja a corrupção de dados a abordagem por *software* deve ser suficiente. Algumas falhas, como as de *bit-flip* (inversão de valores) em células de memória, podem ser inseridas por qualquer método. Em um caso como esse, requisitos adicionais como custo, precisão, intrusividade e possibilidade de nova execução devem guiar a escolha da abordagem (HSUEH; TSAY; IYER, 1997).

2.1.1 Injeção de Falhas por Software

A injeção de falhas por *software* emula falha de *hardware* através de *software* corrompendo o estado do programa em execução. Deste modo, o objetivo da injeção de falhas por *software* é modificar o estado do *hardware/software*, que está sob controle do *software*, levando o sistema a se comportar como se falhas de *hardware* estivessem presentes (KANAWATTI; IYER, 1995).

A injeção de falhas por *software* consiste, usualmente, na interrupção da execução da aplicação e execução do código injetor de falhas. O injetor emula falhas pela inserção de erros em diferentes partes do sistema, como por exemplo, alteração do conteúdo de registradores e posição de memória, alteração de código e *flags* (KANAWATTI; IYER, 1995).

Técnicas de injeção de falhas por *software* são atrativas, pois não necessitam de *hardware* caro. Além do mais, podem ser usadas em aplicações alvo e sistemas operacionais, o que é difícil de ser feito com injeção de falhas por *hardware* (HSUEH; TSAY; IYER, 1997). Os métodos de injeção de falhas por *software* podem ser categorizados baseando-se no momento da injeção das falhas: durante o tempo de compilação ou durante a execução.

2.1.1.1 Injeção de Falhas em Tempo de Compilação

Para a injeção de falhas em tempo de compilação, a instrução do programa deve ser modificada antes da imagem do programa ser carregada e executada. Mais do que injetar falhas no *hardware* do sistema alvo, esse método injeta erros no código fonte ou código montado do programa alvo para emular o efeito de falhas de *hardware*, *software* e transientes. O código modificado altera as instruções do programa alvo, causando a

injeção. A injeção gera uma imagem errônea do *software*. E quando o sistema executa a imagem alterada, a falha é ativada (HSUEH; TSAY; IYER, 1997).

Esse método requer a modificação do programa que irá avaliar o efeito da falha e não necessita nenhum *software* adicional durante a execução. Além disso, não causa nenhuma perturbação ao sistema alvo durante a execução. Pelo efeito da falha ser *hard coded* podem ser usadas para emularem falhas permanentes (HSUEH; TSAY; IYER, 1997).

2.1.1.2 Injeção de Falhas em Tempo Execução

A injeção de falhas em tempo de execução possui a habilidade de injetar uma vasta gama de falhas. São necessários dois elementos: um *software* adicional para injeção de falhas e algum mecanismo para disparar o injetor de falhas, ou seja, a aplicação deve ser preparada para experimentos de injeção de falhas. A atividade de injeção de falhas consiste na execução do código injetor, que emula os efeitos do erro desejado, concorrentemente à aplicação alvo (BARCELOS; WEBER, 2001).

Para injeção de falhas em tempo de execução é necessário algum mecanismo que dispare a falha. Os mecanismos de disparo mais comuns incluem temporizadores, exceções/interrupções e instrumentação de código.

O mecanismo de temporização é uma técnica simples, onde um temporizador expira em um tempo pré-determinado, disparando a injeção. Mais especificamente, o evento do temporizador dispara uma interrupção para invocar a injeção de falhas. Esse método não necessita modificações na aplicação alvo, apenas um temporizador deve ser ligado ao sistema de captura de interrupções. Uma vez que falhas são injetadas com base no tempo e não em eventos específicos ou estado do sistema, o mecanismo de temporização produz efeitos imprevisíveis no comportamento do programa. Entretanto, é um mecanismo apropriado para a simulação de falhas transientes ou falhas de *hardware* intermitentes (HSUEH; TSAY; IYER, 1997).

No caso de exceções/interrupções, o controle é transferido para o injetor de falhas após uma exceção de *hardware* ou *trap* de *software*. Diferente do mecanismo de temporização, as falhas podem ser injetadas sempre que certos eventos ou condições ocorrerem. Por exemplo, uma instrução de *trap* de *software* invoca a injeção de falhas antes da execução de uma determina instrução. Quando o *trap* é executado, gera uma interrupção que transfere o controle para o manipulador de interrupções. Assim como o mecanismo de temporização, deve existir a ligação com o sistema de captura de interrupções (HSUEH; TSAY; IYER, 1997).

Na instrumentação de código, as instruções são adicionadas ao programa alvo, permitindo que a injeção da falha aconteça antes, depois ou no lugar de instruções específicas. Diferentemente da modificação de código, a instrumentação de código realiza injeção de falhas em tempo de execução e adiciona instruções, mais do que simplesmente modifica as instruções originais do código fonte (HSUEH; TSAY; IYER, 1997). O código adicional utilizado na instrumentação de código pode estar executando em paralelo a aplicação alvo, modificando os *bytecodes* (no caso de aplicações Java) da mesma, no instante em que é identificado o momento da injeção.

2.1.2 Injeção de Falhas por Hardware

Uma abordagem bastante comum de injeção de falhas consiste na injeção de falhas físicas no hardware do sistema real. Diversos métodos têm sido utilizados, tas como injeção de falhas em nível de pinos, através de distúrbios externos, *built-in*, dentre outros (KARLSSON *et al*, 1994). Esses métodos, cuja classificação abrange os métodos com contato e sem contato, apresentam a vantagem de gerarem falhas de *hardware* reais, o que os torna mais próximos de um modelo de falhas real.

A injeção de falhas implementada por *hardware* utiliza-se de *hardware* adicional para introduzir falhas no sistema alvo (HSUEH; TSAI; IYER, 1997). Em alguns casos, como por exemplo, injeção de falhas em nível de pinos, a alta complexidade e a alta velocidade dos processadores disponíveis atualmente tornam o projeto deste hardware muito difícil, ou até mesmo impossível. O principal problema não está na injeção de falhas em si, mas está relacionado a dificuldade de controle e observação dos efeitos das falhas nos processadores (CUNHA; RELA, SILVA, 2000).

Outra desvantagem da utilização de injeção de falhas por hardware é a dificuldade de sincronizar a introdução de falhas com o estado do sistema. Além disso, há exigência de hardware dedicado e corre-se o risco de danificar o próprio sistema (BARCELOS WEBER, 2001).

Dependendo das falhas e de suas localizações, os métodos de injeção recaem em duas categorias:

- Injeção de falhas de hardware com contato: o injetor tem contato físico direto com o sistema alvo, produzindo mudanças de voltagem ou corrente, externamente ao chip alvo.

- Injeção de falhas de hardware sem contato: o injetor não tem contato físico direto com o sistema alvo. Ao invés disso, uma fonte externa produz algum fenômeno físico, como uma radiação iônica ou interferência eletromagnética, causando correntes artificiais dentro do chip alvo.

Engenheiros geralmente modelam métodos de *hardware* em modelos de falhas de baixo nível; por exemplo, uma falha de ligação deve ser um circuito pequeno. O *hardware* também dispara falhas e monitora seus impactos, dessa forma oferece grande precisão temporal e baixa perturbação. Normalmente, o *hardware* dispara falhas após um tempo especificado tenha expirado ou depois de ter detectado um evento, como um endereço especificado no barramento de endereços.

2.1.3 Injeção de Falhas de Comunicação em Sistemas Distribuídos

Um sistema distribuído pode ser visto de duas formas: como um modelo físico ou como um modelo lógico. Um modelo físico consiste de vários computadores em posições geográficas distintas, porém interligados por uma rede de comunicação. Um modelo lógico vê uma aplicação distribuída como um conjunto finito de processos e canais de comunicação entre processos (JALOTE, 1994).

Em ambos os modelos, os nodos (físicos ou lógicos) se comunicam através de mensagens. O serviço de comunicação é provido pelo sistema operacional através de

primitivas de *send* e *receive*. Para um processo, falhas de comunicação provenientes do universo físico e falhas emuladas pelo sistema operacional são indistinguíveis. A injeção de falhas de comunicação se dá exclusivamente através da manipulação de mensagens, a serem transmitidas ou recebidas, presentes em um dado nodo do sistema distribuído (LEITE, 2001).

2.2 Intrusividade

A interferência do injetor de falhas da aplicação alvo, chamada de intrusividade, pode ser observada de forma temporal e espacial (DAWSON; JAHANIAN; MITTON, 1996). Na intrusividade temporal, o tempo de execução da aplicação é aumentado devido ao acréscimo das atividades do injetor, que são executadas juntamente com a aplicação alvo. A espacial surge da modificação do código da aplicação alvo. A intrusividade temporal é crítica principalmente em aplicações de tempo real, porém a intrusividade espacial é uma preocupação para todas as classes de aplicação, pois alterações no código fonte podem espalhar *bugs* e mesmo alterar o fluxo de execução da aplicação. Adicionalmente, não é raro que a totalidade ou uma parte considerável do código fonte não esteja disponível para teste. Este trabalho se preocupa em evitar a intrusividade espacial, comum em algumas abordagens de injeção de falhas baseadas em *software*.

2.3 Instrumentação de Código

Este trabalho utiliza instrumentação dinâmica de código para inserir o comportamento errôneo nos sistemas a serem validados. Como citado na seção 2.1.1, um dos mecanismos utilizados para injeção de falhas por software em tempo de execução é a instrumentação de código. A instrumentação de código em tempo de execução é chamada de instrumentação dinâmica.

A instrumentação é o processo de adicionar código em um programa ou classe. Um instrumento é o nome dado a uma unidade de código que pode ser adicionada ou removida de um programa. Instrumentos são simples trechos de códigos que permitem ao desenvolvedor ou usuário obter informação ou medidas sobre a execução do programa (PEARSON, 2003).

O mecanismo de instrumentação de código permite que módulos de programas sejam completamente reescritos em tempo de execução, possibilitando a introdução de novas funcionalidades após a disponibilização da aplicação. É possível a introdução ou remoção de instruções, alterando o uso de classes, variáveis e constantes. A idéia principal é que existe a possibilidade de modificações antes do código ser realmente executado.

Uma propriedade importante da instrumentação é que os próprios códigos instrumentados têm o conhecimento do contexto em que estão sendo aplicados. Por exemplo, um único trecho de código instrumentado é aplicado em diferentes lugares estando apto a fornecer informações precisas de qual aplicação ele está sendo chamado (PEARSON, 2003).

Várias técnicas e abordagens vêm sendo estudadas para que a instrumentação de código evite a alteração do código original, como por exemplo, a reflexão computacional e a programação orientada aspectos.

A alteração direta do código da aplicação a ser instrumentada é a forma mais comum de instrumentação, mas também a mais problemática, pois o código instrumentado é espalhado pela aplicação. Além disso, existe a necessidade de recompilação a cada introdução de código adicional e o mais grave, a introdução não desejada de erros, que pode acarretar a alteração da semântica da aplicação.

A instrumentação de código por reflexão computacional oferece maior flexibilidade à instrumentação de código, pois permite alteração nas classes de uma aplicação independente da disponibilidade do código fonte. Técnicas reflexivas permitem que um programa seja escrito com uma abstração de alto nível para suportar melhor modularização (ROYCHOUDHRY *et al*, 2003). Porém, dependendo da ferramenta utilizada existe a necessidade de que uma cláusula de instanciação seja declarada ao longo da declaração da classe base, como por exemplo, a ferramenta OpenJava (TATSUBORI *et al*, 2000), introduzindo sobrecarga e custos em manutenção e alteração.

A instrumentação de código por POA permite a interceptação e instrumentação de elementos de classes, necessitando apenas o conhecimento das suas APIs. Além disso, preserva a integridade da classe base, isto é, não são necessárias mudanças estruturais. As construções chamadas de Conjuntos de Pontos de Junção (*pointcuts)* de POA identificam claramente os nomes de cada método em cada classe envolvida, dessa forma o programador pode identificar facilmente a parte do código que necessita ser instrumentada após a chamada de cada método (ROYCHOUDHRY *et al*, 2003).

A principal vantagem do uso de POA em relação a reflexão computacional é que, sua funcionalidade é plugável. Por exemplo, se uma funcionalidade necessita ser removida do sistema (ou por não ser mais necessária ou por não ser necessária em determinadas configurações), basta que se remova o aspecto responsável pela funcionalidade. Ao contrário da ferramenta de reflexão computacional OpenJava que cria a necessidade de varrer cada classe afetada e remover as cláusulas de instanciação. A seção 3.5.2 apresenta alguns trabalhos que utilizam POA para instrumentação de código.

2.4 Modelos de Falhas

As falhas são fenômenos aleatórios, imprevisíveis e que podem levar um sistema a um estado errôneo. Se os erros não são tratados, o sistema pode apresentar defeitos. Um defeito ocorre sempre que um serviço não é prestado de acordo com a sua especificação. Dependendo do tipo de falha, ela recebe uma classificação. Em um contexto distribuído, as falhas que são consideradas são baseadas no modelo de falhas para sistemas distribuídos definido por Cristian (CRISTIAN, 1991). Esse modelo classifica as falhas em quatro categorias: colapso, omissão, temporização e bizantinas.

As falhas de colapso são falhas que causam a parada de um componente ou a perda de seu estado interno. As falhas de omissão ocorrem quando um componente não responde a algumas entradas. A falha de temporização, ou também chamada falha de

desempenho, acontece quando a resposta do servidor é correta, porém, fora do intervalo de tempo especificado para que ela aconteça. As falhas bizantinas são falhas arbitrárias, que fazem com que o componente se comporte de maneira completamente arbitrária durante a falha (JALOTE, 1994).

Dependendo do estado do componente no seu reinício, se distinguem vários tipos de comportamentos de falhas de colapso. O colapso de amnésia é considerado quando o componente reinicia em um estado inicial pré-definido que não depende das entradas anteriores ao colapso. Quando, no reinício, alguma parte do estado é igual ao estado anterior ao colapso e o resto do estado é reiniciado para um estado pré-definido, é chamado de colapso de amnésia parcial. O colapso de pausa é caracterizado quando o componente reinicia no estado que tinha anterior ao colapso. E, finalmente, o colapso de parada ocorre quando um componente em colapso nunca reinicia (CRISTIAN, 1991).

As falhas definidas acima formam uma hierarquia, onde as falha de colapso é a mais simples e restritiva (ou bem definida) e a bizantina é a menos restritiva. Um outro tipo de falha pode ser considerado tipo de falha chamado de falha de resposta, a qual é resultado de uma computação é incorreto. As falhas de resposta podem ser vistas como um subconjunto das falhas bizantinas, porém, diferente das outras. Nesse tipo de falha, um componente simplesmente produz uma saída incorreta para as entradas fornecidas. (JALOTE, 1994). A figura 2.1 ilustra a relação entre os tipos de falhas.

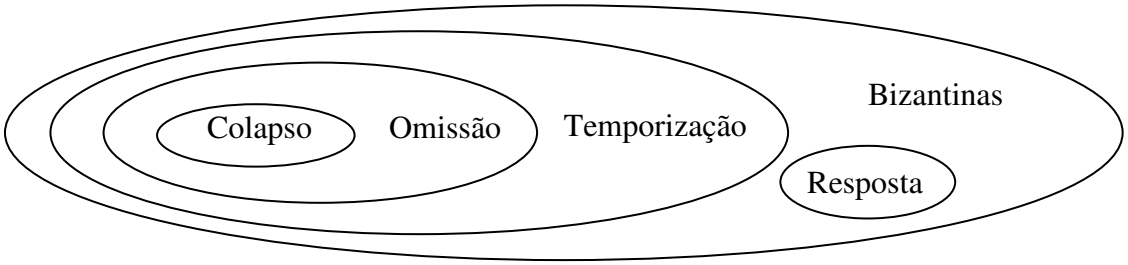

Figura 2.1 - Classificação de Falhas

Em componentes sem estado, colapsos de pausa e de parada se comportam como subconjuntos de comportamentos de falhas de omissão. Em geral, os comportamentos de colapso de amnésia total e parcial não são subconjuntos de falhas de omissão (CRISTIAN, 1991).

Como exemplos de falhas de colapso podem ser citados: um sistema operacional entra em colapso, seguido por um reinício e um servidor de banco de dados que entra em colapso seguido de uma recuperação do estado do banco de dados que reflete todas as transações efetivadas antes do colapso. Um serviço de comunicação que ocasionalmente perde, mas não atrasa mensagens é um exemplo de serviço que sofre falha de omissão. Uma transmissão excessiva de mensagens ou atraso no processamento das mesmas é considerada uma falha de desempenho (temporização). A alteração de uma mensagem em um canal de comunicação sujeito a ruído é exemplo de falha de resposta (CRISTIAN, 1991).

2.5 Ferramentas de Injeção de Falhas

Uma ferramenta de injeção de falhas por software geralmente é um trecho de código que usa todos os ganchos possíveis do processador para criar um comportamento errôneo de maneira controlada (CARREIRA; SILVA, 1998). As ferramentas de injeção de falhas executam o sistema teste e geram ou simulam falhas, monitoram e verificam como o sistema se comporta. Diferentes estratégias e abordagens já foram usadas na criação de ferramentas de injeção de falhas para sistemas distribuídos. A seguir são apresentadas algumas dessas abordagens, em ordem cronológica.

FIAT (*Fault-Injection-based Automated Testing*) (SEGALL, 1988) combina técnica de injeção de falhas por *software* e *hardware* para avaliar a dependabilidade de sistemas distribuídos de tempo real, tolerantes a falhas. A ferramenta injeta erros diretamente na memória, ajustando e limpando *bytes* nas imagens das aplicações alvo. FIAT é composta de dois componentes ligados por uma rede de comunicação. O componente FIM (*Fault Injection Management*) é o controle global, suporta todas as fases do experimento, realiza a coleta e analise de dados e controla o experimento em tempo de execução. O componente FIRE (*Fault Injector Receptors*) executa sob controle do FIM e oferece a plataforma de execução para o sistema distribuído de tempo real sob teste.

FINE (*Fault Injection moNItoring Environment*) (KAO; IYER, 1993) tem o objetivo de observar como acontece a propagação das falhas no *kernel* do Unix e avaliar a dependabilidade do sistema e seus mecanismos de detecção e remoção de erros. DEFINE (KAO *et al*, 1994) é uma evolução do FINE que inclui capacidade de observar aplicações distribuídas. O injetor modifica imagens executáveis da aplicação alvo para emular falha de memória, como por exemplo, pela inserção de interrupções de software em localizações específicas de memória no segmento de texto.

CSFI (*Communication Software Fault Injection*) (CARREIRA; MADEIRA, 1995) foi uma das primeiras ferramentas desenvolvidas para a injeção de falhas de comunicação e seu principal objetivo era avaliar o impacto de falhas em sistemas paralelos. A CSFI foi desenvolvida para um sistema Transputer T805.

A ferramenta ORCHESTRA (DAWSON; JAHANIAN; MITTON, 1996) foi desenvolvida para teste de dependabilidade de protocolos distribuídos. A ferramenta injeta falha através da inclusão de uma camada extra, a chamada PFI (*Protocol Fault Injection*), na pilha de protocolos, abaixo do protocolo sob teste. A injeção de falhas é realizada através da manipulação de mensagens, filtrando, interceptando, analisando, perdendo, atrasando, reordenando, duplicando, modificando e introduzindo mensagens.

FIRE (*Fault Injection using a Reflective Architecture*) (ROSA; MARTINS, 1998) utiliza reflexão computacional para minimizar a intrusividade no teste de aplicações distribuídas orientadas a objetos. FIRE utiliza injeção de falhas por *software* ativada em tempo de execução. A reflexão computacional foi utilizada para implementar uma biblioteca de injeção e monitoração que é ligada a aplicação alvo.

A ferramenta ComFIRM (*Communication Fault Injection through OS Resources Modification*) (DREBES *et al*, 2005) foca apenas em falhas de comunicação e está localizada dentro do *kernel* do sistema operacional Linux, na camada mais baixa do

subsistema de manipulação de mensagens de rede. O código da ferramenta é inserido diretamente dentro do *kernel* do sistema operacional, reduzindo a intrusão temporal do sistema em teste. A ferramenta ComFIRM foi desenvolvida no âmbito do grupo de Tolerância a Falhas da UFRGS e é fortemente inspirada na ferramenta ORCHESTRA.

INFIMO (*INtrusivelles Fault Injection Module*) (BARCELOS *et al*, 2004) é um *toolkit* baseado em Linux que tem como objetivo analisar de forma experimental a intrusividade do injetor de falhas sobre os protocolos alvo com restrições de tempo real. INFIMO suporta falhas de comunicação, as quais podem ocasionar omissão de pacotes. A ferramenta se baseia na alteração de bibliotecas de protocolo de comunicação e em recursos de depuração do sistema operacional para interceptar as chamadas de sistema. Essa ferramenta também foi desenvolvida pelo grupo de Tolerância a Falhas da UFRGS.

Jaca (MARTINS; RUBIRA; LEME, 2002) é uma ferramenta extensível, baseada em reflexão computacional, com a finalidade de validar aplicações orientadas a objetos baseadas em Java. Como plataforma de suporte para reflexão computacional, Jaca utilizada o protocolo de meta objetos não padrão Javassist (CHIBA, 2004), que permite que os *bytecodes* sejam transformados durante o tempo de carga. Jaca teve seu modelo de falhas estendido para suportar falhas de comunicação UDP (SILVA, 2004-a), porém, essa extensão necessita uma fase de pré-processamento do código da aplicação (código fonte ou *bytecodes*) para a injeção de falhas de comunicação. Isto é necessário para que a aplicação possa se referir às classes de comunicação que suportam a injeção de falhas.

A ferramenta FIONA (SILVA, 2004-b) utiliza JVMTI para injeção de falhas de comunicação. JVMTI é uma nova interface de programação nativa da plataforma Java que permite o desenvolvimento de ferramentas de monitoramente e *debugging*, permitindo a instrumentação de aplicações Java.

Tabela 2.1 - Quadro Comparativo Jaca X FIONA X FICTA

Ferramenta	**Estratégia**	**Ferramenta de Apoio na Construção**
Jaca	Reflexão Computacional	Javassist
FIONA	Programação Nativa em Java	JVMTI
FICTA	Programação Orientada a Aspectos	AspectWerkz

Este trabalho apresenta uma nova estratégia para a construção de injetores de falhas, baseada em Programação Orientada a Aspectos. Essa estratégia resultou na ferramenta FICTA (SILVEIRA; WEBER, 2005), que foi fortemente inspirada na extensão da ferramenta Jaca para falhas de comunicação e FIONA no sentido que possui um modelo de falhas semelhante e também injeta falhas em aplicações Java distribuídas baseadas no protocolo UDP, pelos mesmos motivos que as ferramentas citadas. A tabela 2.1 apresenta um quadro comparativo entre as estratégias utilizadas pelas ferramentas Jaca, FIONA e FICTA, todas desenvolvidas dentro do mesmo grupo de pesquisa, porém, em paralelo e sem interferência nas decisões de projeto de cada uma. As estratégias de implementação utilizadas em todas as ferramentas visam uma diminuição da intrusão espacial das ferramentas no código fonte da aplicação. As ferramentas Jaca e FICTA utilizam apenas programação em alto nível (reflexão

computacional e orientação a aspectos) enquanto que FIONA utiliza programação nativa em Java, tendo alguns trechos de seu código escritos na linguagem C.

A estratégia de injeção de falhas desse trabalho diferencia-se de FIONA, Jaca e também das demais por usar POA para realizar a interceptação e instrumentação de *bytecodes* das classes que implementam os métodos de comunicação das aplicações alvo distribuídas. A POA é um paradigma de programação relativamente novo, complementar a orientação a objetos e que tem como objetivo a melhor modularização de interesses transversais que estão espalhados em vários módulos de um sistema. A utilização de uma estratégia orientada a aspectos permite que os aspectos sejam carregados junto com o sistema alvo (os aspectos e as classes são combinados em tempo de carga) e a instrumentação das primitivas de comunicação é realizada efetivamente em tempo de execução.

3 PROGRAMAÇÃO ORIENTADA A ASPECTOS

Esse capítulo apresenta o paradigma de orientação a aspectos, o conceito de interesses transversais, os elementos da orientação a aspectos, como a orientação a aspectos está sendo utilizada e duas ferramentas utilizadas para o desenvolvimento de software orientado a aspectos.

3.1 Introdução

As linguagens de programação evoluíram de códigos de máquina e linguagens de montagem para uma variedade de paradigmas como programação procedural, funcional, lógica e orientada a objetos. O avanço das tecnologias de programação aperfeiçoou a capacidade dos desenvolvedores de software de alcançar uma clara separação de interesses, ou seja, "a capacidade de identificar, encapsular e manipular apenas as partes do software que são relevantes a um conceito, meta ou propósito específico" (OSSHER; TARR, 2001). Um sistema complexo pode ser visto como uma implementação combinada de vários interesses, como por exemplo, persistência, *logging*, depuração, segurança, etc.

A Programação Orientada a Objetos - POO - se tornou o paradigma de programação dominante, onde um problema é decomposto em objetos que abstraem comportamentos e dados em uma única entidade. A tecnologia de objetos reduz a complexidade de escrever e manter aplicações complexas como aplicações distribuídas e interfaces gráficas de usuário. Embora a tecnologia de POO ofereça grande capacidade para separação de interesses, ela continua tendo dificuldade de localizar aqueles que não se encaixam naturalmente em um único módulo de programa ou em módulos fortemente relacionados. Interesses podem estar num amplo intervalo que envolve desde noções de alto nível, assim como segurança e qualidade de serviços, até noções de baixo nível como buferização, *caching* e *logging*. Alguns interesses, como por exemplo, XML *parsing* se encaixam em apenas alguns objetos, alcançando uma boa coesão. Outros, como *logging*, podem interagir com vários módulos que não se relacionam ente si (LEE, 2003).

Os interesses transversais tendem a afetar vários módulos de implementação de um sistema, isto é, seus códigos se "espalham", dificultando a manutenção e compreensão do sistema (LADDAD, 2002). Interesses transversais podem gerar problemas como espalhamento e emaranhamento de código. Esses problemas acabam

gerando diminuição de reuso de código, péssima qualidade de código e dificuldade de ampliação do sistema. A Programação Orientada a Aspectos - POA - vai de encontro a resolução desses problemas.

3.2 O que é Programação Orientada a Aspectos?

A Programação Orientada a Aspectos – POA - (KICKZALES, 1997) é uma tecnologia para a separação de interesses transversais em unidades chamadas de aspectos. O principal objetivo de POA é auxiliar os desenvolvedores a manter código relacionado em um único ponto. Um aspecto é uma unidade modular de implementação transversal e encapsula os comportamentos que afetam diversas classes em módulos reusáveis. Tarefas relacionadas, como segurança e *logging*, podem ser extraídas do código fonte e colocadas separadamente em um aspecto e podem ser automaticamente integradas por um combinador de aspectos (*aspect weave*r) em uma forma final executável. Como resultado, um único aspecto pode contribuir na implementação de procedimentos, módulos ou objetos, aumentando a reutilização dos códigos.

A POA é uma proposta de solução para os problemas causados pela ineficiência dos atuais paradigmas de programação em prover a modularidade adequada dos interesses multidimensionais no desenvolvimento de software. Ela propõe uma separação clara dos interesses multidimensionais num sistema, de forma a evitar o entrelaçamento de interesses distintos.

Na Programação Orientada a Objetos – POO - a unidade natural de modularização é a classe, e um comportamento transversal está espalhado em várias classes. Trabalhar com código que aponta para responsabilidades que atravessam o sistema gera problemas que resultam na falta de modularidade. A POA complementa a POO por facilitar um outro tipo de modularidade que expande a implementação espalhada de um interesse dentro de uma simples unidade. É possível dizer que a POO implementa a modularização de interesses comuns enquanto POA implementa a modularização de interesses transversais.

A POA faz para interesses transversais o que a POO faz para encapsulamento e herança, oferecendo mecanismos que capturam explicitamente estruturas que atravessam o sistema. Um exemplo do modo como aspectos cruzam classes é apresentado na figura 3.1 (KICKZALES *et al*, 2001). É apresentado um diagrama de classes de um editor simples de figuras. A caixa nomeada de `RastreiaMovimento` mostra um aspecto que encapsula alguns métodos que entrecortam as classes `Ponto` e `Linha`. Os métodos em cinza são comuns as duas classes e podem ser encapsulados na forma de um aspecto.

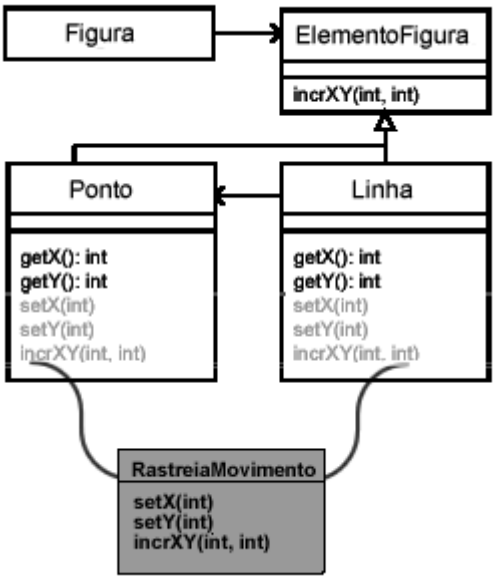

Figura 3.1 - Exemplo de como um aspecto atravessa classes de um sistema

Outro exemplo comum de utilização de POA é o de um interesse que realiza o *Logging* de exceções disparadas pela aplicação. O código relativo ao *Logging* se encontra espalhado através das classes da aplicação, tornando o código de difícil manutenção, pois estará afetando várias classes do sistema, que deverão ser atualizadas uma a uma em caso de alteração ou remoção da classe de *Logging*. Utilizando POA, o código relativo ao *Logging* se encontra em uma única unidade conceitual simplificando a manutenção, melhorando o reuso e reduzindo os custos de modificações. Da mesma forma, um mecanismo de injeção de falhas, pode ser visto com um interesse que atravessa a aplicação a ser validada. A utilização de aspectos reúne todo o código de injeção de falhas em um único aspecto, trazendo todos os benefícios citados acima para o caso de *Logging*.

3.3 Conceitos Básicos

Uma linguagem orientada a aspectos possui três elementos críticos para a separação de interesses transversais: um modelo de pontos de junção, uma forma de identificar *join point*s e um meio de especificar comportamentos nos *join point*s (ELRAD *et al*, 2001). O modelo de *join point* oferece a forma de construção comum para definir a estrutura de interesses transversais e descrever os ganchos onde modificações devem ser adicionadas. Além disso, também existe a necessidade de um elemento que encapsule a combinação de *join point*s e seus comportamentos e também um método de unir essas unidades no programa.

A linguagem AspectJ (KICKZALES *et al*, 2001) surgiu juntamente com o paradigma de POA para complementar os conceitos do mesmo. Dessa forma surgem os elementos *join points*, *pointcuts*, *advices*, introduções e aspectos. Um *join point* é um ponto bem definido no fluxo do programa, por exemplo, uma chamada de método. Um *pointcut* separa certos *join points* e alguns valores nesses *join points*. Um *advice* é o código que é executado quando um *join point* é alcançado. Esses elementos definidos

por AspectJ servem de base para outras linguagens orientadas a aspectos e são definidos a seguir (KICKZALES *et al*, 2001) (BONÉR, 2005).

Join points

Join points são pontos bem definidos no fluxo do programa, podendo ser de diferentes tipos: de chamada de método ou construtor, de chamada de execução de método ou construtor, de *get* e *set* de campos, de execução de manuseio de exceções e de inicialização estática e dinâmica.

Pointcuts

Umas das principais características de uma linguagem para orientação a aspectos é a possibilidade de selecionar determinados *join points*. Um *pointcut* seleciona *join points* específicos filtrando um subconjunto de todos os *join points*, baseado em um critério definido. O critério pode ser nome de funções explicitas ou especificadas por curingas. *Pointcuts* podem ser compostos usando operadores lógicos e podem identificar *join points* de diferentes classes.

Advices

Os *pointcuts* são utilizados na definição dos *advice*s. Um *advice* é utilizado para definir o código adicional que deve ser executado nos *join point*s. Os seguintes tipos de advices podem ser encontrados:

• *Before advice*: o código executa quando o *join point* é alcançado, porém, antes da execução do corpo do método, construtor, exceção, etc., ocorrer.

• *After advice*: o código executa após a execução de um *join point* ser completada, porém, antes da saída do *join point*.

• *Around advice*: o código executa quando o *join point* é alcançado, substituindo o código original do *join point* se as condições especificadas no *advice* são satisfeitas. O fluxo de controle de um método modificado por um *around advice* é apresentado na figura 3.2.

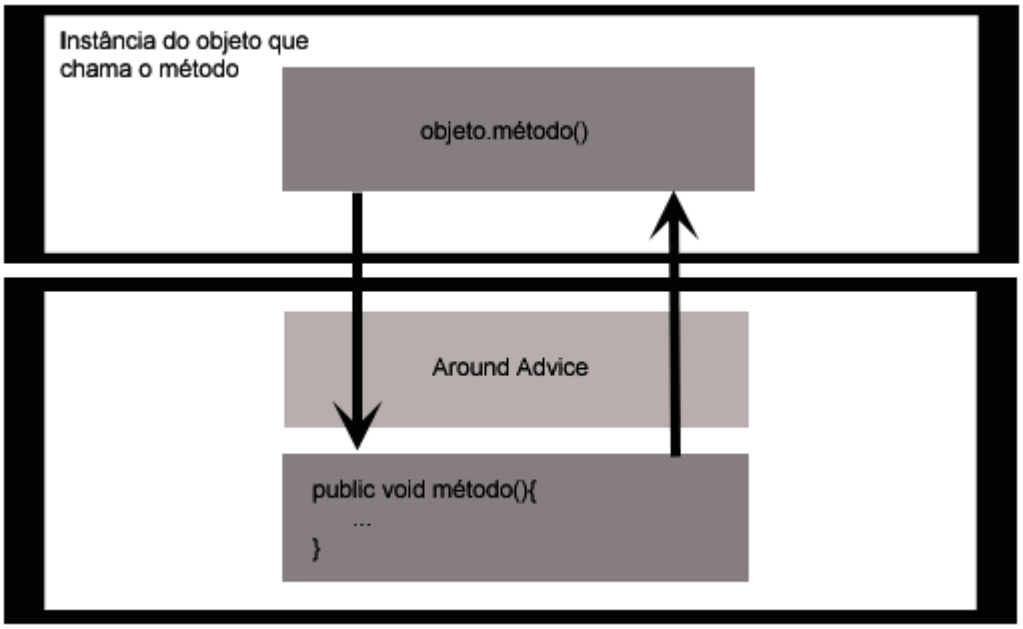

Figura 3.2 - Fluxo de controle de um método modificado por um *around advice*

Introduções

Uma introdução insere novos membros nas classes e, portanto, pode alterar a relação de herança entre classes, tornando possível estender uma classe com uma nova interface, super classe, métodos e campos.

Aspectos

Os aspectos são as unidades modulares de interesses transversais e são definidos em termos de *join points, pointcuts, advices* e introduções. Um aspecto é similar a uma classe tendo um tipo, estendendo outras classes e outros aspectos. Ele pode ser abstrato ou concreto e ter campos, métodos e tipos como membros. Ele encapsula comportamentos que afetam múltiplas classes em módulos reusáveis.

A herança de aspectos assemelha-se a herança de uma classe Java regular. Por exemplo, os *pointcuts* do aspecto abstrato são herdados pelo aspecto concreto, tornando possível a criação de componentes reusáveis de aspectos e bibliotecas.

3.4 Weaving

O principal trabalho de uma implementação de linguagem para POA é garantir que códigos de aspectos e de classes executem juntos de forma apropriada e coordenada. Essa coordenação é chamada de *weaving* e envolve a garantia de que os *advices* aplicáveis executem nos *join points* apropriados (KICKZALES *et al*, 2001).

O *weaving* realiza o processo de instrumentação das classes, modificando o *bytecode* de modo a incluir interfaces e as chamadas aos *advice*s. Os integradores de aspectos (*aspect weavers*) devem processar os componentes dos sistemas com os aspectos, os compondo de forma apropriada para produzir a operação total desejada do sistema. A figura 3.3 ilustra, de forma simplificada, o processo de *weaving*.

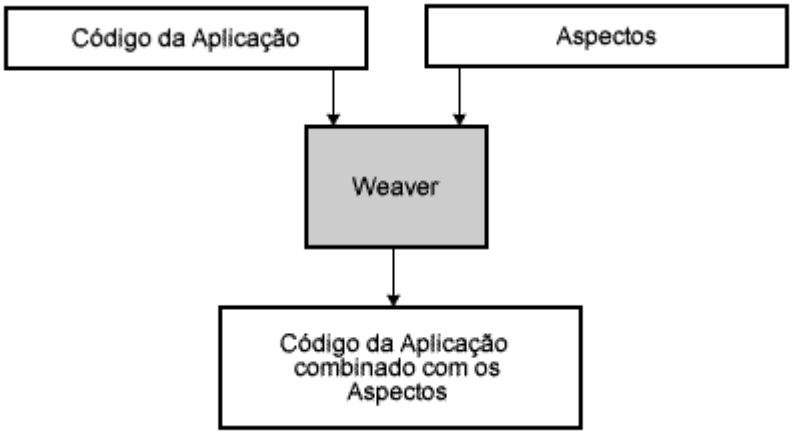

Figura 3.3 - Weaving

O processo de *weaving* pode ser estático ou dinâmico. Com o *weaving* estático, os aspectos são integrados resultando em uma única classe, como é feito, por exemplo, na linguagem AspectJ (KICKZALES *et al*, 2001). Os aspectos são entidades em tempo de compilação. Em *weaving* dinâmico, os aspectos sempre são classes separadas, que são chamadas por algum *framework* que intercepta dinamicamente o uso de algum objeto, como é realizado, por exemplo, pelo AspectWerkz (BONER, 2004-a). Nesse caso, os aspectos são entidades em tempo de execução.

3.5 Ferramentas para Programação Orientada a Aspectos

Existem várias linguagens orientadas a aspectos, dentre elas podem ser citadas AspectJ (KICKZALES *et al*, 2001) e AspectWerkz (BONER, 2005). O AspectJ é a linguagem orientada a aspectos mais utilizada pois surgiu para complementar os conceitos de POA. O AspectWerkz é um *framework* recente que permite a especificação de aspectos através de XML (XML, 2005) e Meta-Atributos.

Em março de 2005, AspectJ e AspectWerkz (ASPECTJ+ASPECTWERKZ, 2005) divulgaram sua união. As equipes irão produzir uma única plataforma de Programação Orientada a Aspectos complementando suas forças e experiências. O primeiro resultado dessa colaboração será o AspectJ 5, que estende a linguagem AspectJ para suportar estilo de desenvolvimento baseado em anotações e XML, em adição ao estilo baseado em código tradicional de AspectJ. Além disso, AspectJ 5 também irá dar suporte total as novas funcionalidades da plataforma Java 5.

Neste texto, as ferramentas serão apresentadas separadamente, visto que todo o trabalho de pesquisa e desenvolvimento da estratégia e da ferramenta FICTA foi realizado no período anterior a união das equipes. A ferramenta AspectJ será apresentada de forma sucinta, como ilustração, pois foi desenvolvida juntamente com o paradigma de orientação a aspectos para complementar seus conceitos. A ferramenta AspectWerkz será apresentada mais detalhadamente por ter sido a ferramenta utilizada desse trabalho.

3.5.1 AspectJ

A linguagem AspectJ é uma implementação de programação orientada a aspectos para Java. AspectJ acrescenta algumas novas construções de linguagem, as já existentes: *join points*, *pointcuts*, *advice*, introducções e aspectos. Essas construções dão suporte ao modelo de *join point*. *Join points* são pontos bem definidos na execução do programa, enquanto *pointcuts* são coleções de *join points*. *Advice*s são construções semelhantes a métodos que podem ser anexadas aos *pointcuts*. Aspectos são unidades modulares de implementação cruzada que encapsulam *pointcuts, advices* e declarações simples de membros de classe Java (KICKZALES *et al*, 2001)

```
aspect NomeAspecto{

      pointcut nomePointcut() : execution(* Classe.metodo(..))

      Object around(): nomePointcut(){

            //código do advice

      }
}
```

Figura 3.4 - Exemplo de definição de aspecto em AspectJ

Em AspectJ os aspectos são definidos em declarações de aspectos, que tem formato parecido com declarações de classes. A figura 3.4 exemplifica a definição de um aspecto em AspectJ, ilustrando a definição do aspecto (semelhante a definição de uma classe, porém utilizando a palavra *aspect*), a definição de um *pointcut* e de um *around advice* definido para o *pointcut*.

3.5.1.1 Weaving

AspectJ trabalha com *weaving* estático. É oferecida uma implementação baseada em compilação, que realiza quase todo o trabalho de *weaving*, em tempo de compilação (KICKZALES *et al*, 2001). Isso causa problemas como, por exemplo, algumas partes do programa não serem afetadas pelos *advices* e serem compiladas como se fosse por um compilador Java padrão. Para que todas as classes sejam atingidas, todos os códigos fonte Java e códigos fonte de aspectos desenvolvidos em AspectJ devem ser compilados ao mesmo tempo, com o compilador chamado de `ajc` do AspectJ.

Para o atual compilador `ajc`, controle de código significa ter *bytecodes* para qualquer aspecto e todos os códigos que ele deve afetar disponíveis durante a compilação. Isto significa que se alguma classe `Cliente` contém código com a expressão `new Pessoa().método` o atual compilador de AspectJ irá falhar ao inserir os advices, a menos que a classe `Cliente` seja compilada juntamente com a classe `Pessoa`. Isso também significa que *joint points* associados a métodos nativos não são atingidos por *advices*.

Os primeiros experimentos para teste da estratégia baseada em aspectos proposta nesse trabalho foram desenvolvidos utilizando AspectJ. O resultado alcançado foi um protótipo de injetor de falhas chamado apenas de FITA – *Fault Injection Tool based on Aspects* (SILVEIRA, WEBER, 2004). Esse protótipo era específico para o *toolkit* de

comunicação de grupo JGroups (BAN, 1998) e possibilitava apenas a instrumentação de classes que implementavam os protocolos desses sistema, pelo obstáculo imposto pelo compilador de AspectJ. Foram esses problemas de *weaving* que fizeram que a opção de ferramenta para esse trabalho fosse o AspectWerkz, definida na próxima seção.

3.5.2 AspectWerkz

Aspectwerkz é um *framework* dinâmico para Programação Orientada a Aspectos em Java. O *framework* utiliza modificação de *bytecodes* para integrar classes e aspectos em tempo de compilação, tempo de carga de classes ou tempo de execução, utilizando APIs padrão em nível de JVM. Aspectos, introduções e advices são escritos em classes Java comuns. Os aspectos são definidos usando anotações de Java 5, Doclets de Java 1.3/1.4 ou arquivos de definição em XML (BONNER, 2004).

```
Public class Aspecto {

      public Object execute(JoinPoint joinPoint) throws Throwable {

            // código que deve ser executado antes de devolver o
            // fluxo de controle para o método afetado pelo around
            // advice

            Object result = joinPoint.proceed();

            return result;

      }

}
```

Figura 3.5 - Exemplo de *around advice* em AspectWerkz

```
<aspectwerkz>
 <system id="id" >
   <package name="pacote">
    <aspect class="FaultInjector">
            <pointcut name="pntCt"
                  expression="call(* Classe.metodo(..)) "/>
          <advice name="advc" type="around" bind-to="pntCt "/>
      </aspect>
   </package>
 </system>
</aspectwerkz>
```

Figura 3.6 - Declaração XML do *around advice*

A figura 3.5 exemplifica um aspecto para ser utilizado com o *framework* AspectWerkz. A figura 3.6 mostra como esse *advice* é definido no XML. O *around advice* é normalmente identificado por retornar um objeto do tipo `Object` da API padrão Java. Esse objeto pode ser uma chamada ao método `proceed()`, que tem como objetivo devolver o fluxo de execução para o método original que está sendo instrumentado pelo *advice*. Além disso, o método `proceed()` também é inserido na chamada do método original, no momento do weaving. Isso é necessário, pois caso o *advice* não encontre as condições necessárias para a sua execução (definidas no seu XML, por exemplo), o fluxo de controle deve ser imediatamente repassado para o método original.

3.5.2.1 Weaving

AspectWerkz permite *weaving* dinâmico e estático e integra as classes em nível de *bytecode*. AspectWerkz oferece os mecanismos de integração *online* e *offline*, equivalentes aos conceitos de *weaving* dinâmico e estático, respectivamente. Ambos são similares e suportam os requisitos de integração com servidores de aplicação, JREs e JVMs. Para algumas situações, entretanto, o modo *online* oferece algum poder adicional sobre o modo *offline*, como pode ser visto a seguir.

Offline Weaving

O modo *offline* permite que os aspectos sejam integrados na aplicação antes da aplicação ser disponibilizada. O modo *offline* é realizado através de um pós-processador. Isto adiciona um passo de compilação extra ao passo de construção, porém a aplicação pode ser iniciada normalmente, sem nenhum requisito especial. É uma boa opção para usuários que não possuem controle total sobre o disparo dos processos ou não podem pensar num atraso extra quando a transformação das classes ocorre em tempo de execução.

Online weaving

O *weaving* dinâmico é chamado de modo *online* e permite a integração dos aspectos em tempo de carga ou de execução dos aspectos de forma transparente. O *weaving* dinâmico é permitido pela modificação da arquitetura de carga de classes Java para permitir que a integração seja realizada no instante anterior a classe ser carregada na JVM. Isso é realizado ligando o *framework* no carregador de classes do sistema, diretamente após o carregador de classe *bootstrap*, dessa forma, torna-se possível realizar transformações de *bytecode* em todas as classes carregadas por todos os carregadores de classes precedentes.

A ferramenta realiza a instrumentação dos *bytecodes* do sistema alvo em dois momentos, o momento de carga da aplicação e no momento da execução da aplicação. A redefinição das classes é feita utilizando *HotSwap*, em Java 1.4, ou JVMTI, em Java 5. No mecanismo padrão da linguagem Java, uma classe é carregada (e reside) no carregador de classes de sistema. Através de *HotSwap* ou JVMTI, é carregado um carregador de classes de sistema modificado que substitui o carregador padrão da arquitetura Java.

O *HotSwap*, também chamado de redefinição de classes, é uma característica adicionada em Java 1.4 e faz parte da API JVMDI (JVMDI, 2005). O objetivo é permitir a submissão de novo código para uma classe já carregada na máquina virtual Java (JVM) que está executando. No mecanismo de *HotSwap* tradicional uma primeira JVM dispara a aplicação alvo em uma segunda JVM com o mecanismo de carga de classes modificado. A primeira JVM liga o *framework* AspectWerkz na segunda JVM no instante em que a classe principal e suas dependências são carregadas. O *HotSwap Nativo,* ao invés de disparar uma segunda JVM, utiliza uma extensão nativa escrita em C, que executa na JVM da aplicação e realiza a modificação do carregador de classes do sistema. A vantagem está em não necessitar uma segunda JVM.

O *HotSwap* apresenta alguns problemas, como não ser adaptado a todas as máquinas virtuais. Além disso, necessita o uso da *flag* de depuração da JVMDI, que pode reduzir desempenho. Esse problema é resolvido em Java 5 através do uso da API JVMTI (JVMTI, 2005), onde os agentes definidos pela JVMTI permitem a redefinição dos *bytecodes* das aplicações em tempo de execução, sem a necessidade de recarregar as classes.

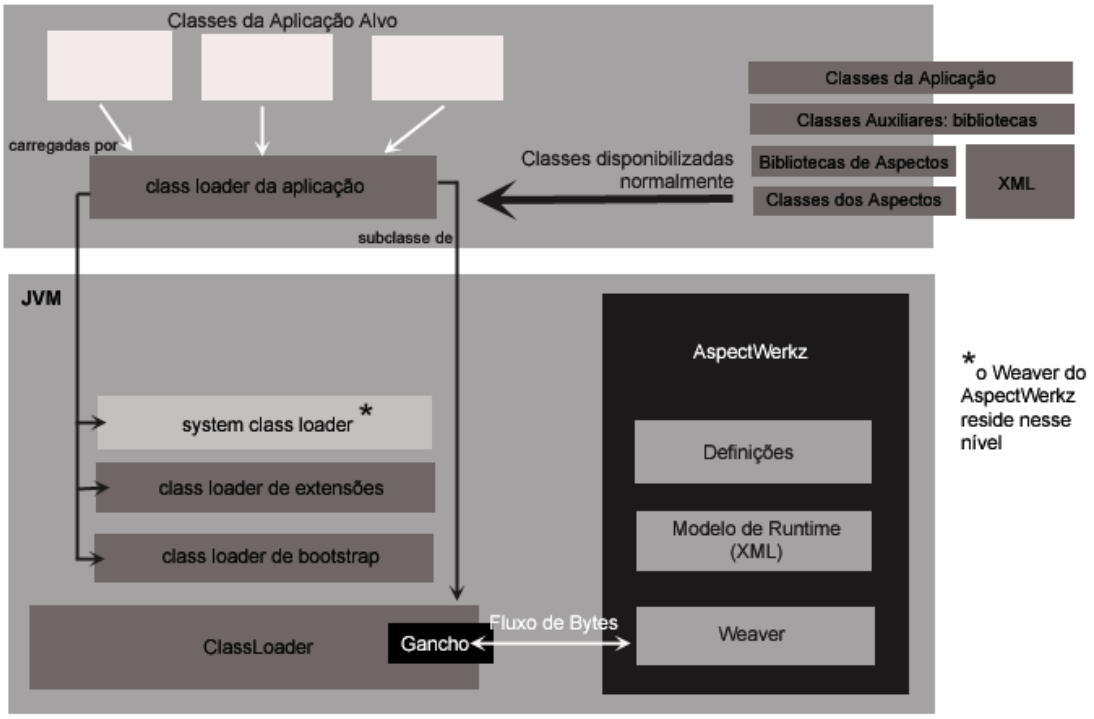

Figura 3.7 - *Online Weaving*

A figura 3.7 ilustra o processo de *online weaving*. No momento de disparo da aplicação as classes do sistema, classes auxiliares, bibliotecas e também os aspectos são carregados no carregar de classes da aplicação. No momento em que é realizada a ligação com o carregador de classes do sistema (alterado pelo *framework* AspectWerkz), são identificados os *join points* na aplicação alvo (definidos nos arquivos XML do AspectWerkz) e então, o carregador de classes faz a ligação com o contêiner de aspectos, que fica responsável por inserir e identificar todos os pontos de interceptação de métodos e onde devem ser substituídos *bytecodes*.

3.5.2.2 Modelos de Definição

AspectWerkz suporta duas formas de definição dos aspectos: através de XML ou através de Anotações. Ambos os formatos são apenas visões diferentes de um mesmo modelo que está executando no nível da JVM, isto é, ambos podem co-existir e serem usados em conjunto. As definições em XML podem ser usadas para substituir a definição por anotações ou como complemento.

Anotações

As anotações são um estilo de programação declarativa onde o desenvolvedor diz o que deve ser feito e as ferramentas utilizadas criam código automático para realizar as

tarefas. As anotações não afetam diretamente a semântica dos programas, mas sim, como os programas tratam ferramentas e bibliotecas, as quais podem afetar a semântica do programa em execução (ANNOTATIONS, 2005). O uso de anotações não é recomendado caso a aplicação necessite ser portável entre vários ambientes (servidores de aplicação, bases de dados, etc.) (BURKE, 2005).

No caso de AspectWerkz, mesmo que todas as definições sejam feitas utilizando anotações (e nada de XML) é necessário que um pequeno descritor XML seja escrito. Isso é necessário para que o sistema, em tempo de execução, saiba quais aspectos devem ser carregados.

XML

As definições em XML podem ser usadas sozinhas ou para refinar e sobrescrever definições feitas com anotações, assim como resolver problemas como definições que faltam, por exemplo, *pointcuts* que são referenciados, mas não foram definidos (BONÉR; VASSEUR, 2004).

Para esse trabalho são utilizadas apenas definições de aspectos em arquivos XML, visto que é a recomendação para que não se perca a portabilidade da ferramenta.

3.6 Trabalhos Desenvolvidos com Programação Orientada a Aspectos

Essa seção tem como objetivo apresentar alguns trabalhos que vêm sendo realizados com programação orientada a aspectos. É importante ressaltar que são apresentados trabalhos que usam efetivamente POA como ferramenta e não estudos da linguagem e estrutura.

A seção é dividida em duas subseções. A primeira apresenta uma visão genérica, sobre várias áreas em que orientação a aspectos está sendo usada. A segunda subseção apresenta alguns trabalhos em que a orientação a aspectos está sendo utilizada para instrumentação de código.

3.6.1 Visão genérica

A ferramenta OIF - *Object Infraestructure Framework* - pretende simplificar a criação de aplicações distribuídas pela introdução de comportamento nos caminhos de comunicação entre os componentes. O objetivo principal é o redirecionamento de requisições de serviços aos servidores específicos, ampliando confiabilidade, segurança, facilidade de gerenciamento e qualidade de serviços. A ferramenta utiliza POA para interceptar a comunicação (FILMAN; LEE, 2001).

Também já foi realizado um trabalho que ilustra como a POA pode ser utilizada para endereçar requisitos não funcionais de sistemas distribuídos de tempo real com características de dependabilidade, separando-os dos requisitos funcionais. São apresentados três interesses, implementados na forma de aspectos: distribuição, tempo-real e tolerância a falhas (GAL; SPINCZYK; PREIKSHAT, 2002).

O DLBS - *Dynamic Load Balacing Service* – é uma ferramenta com o objetivo de solucionar os problemas de escalabilidade em balanceamento de carga para

aplicações baseadas em *middleware* para sistemas distribuídos. A POA foi utilizada para a criação de um aspecto responsável por dois tipos de integração: balanceamento de carga do lado cliente e balanceamento de carga para o serviço de nomes (PUTTRYEZ; BERNARD, 2002).

Também já foram realizados estudos onde é descrito o uso de POA para melhorar o desempenho de servidores tolerantes a falhas construídos com suporte de *middleware*. A abordagem é ilustrada pela descrição de como os aspectos de sincronização são mesclados nas aplicações através da modificação da plataforma FT-CORBA (SZENTIVANYI; TEHRANI, 2004).

Uma avaliação do uso de POA para a separação de interesses transversais de sistemas de tempo real foi realizado com o objetivo de determinar quando o uso de POA para encapsular esses interesses traria benefícios para facilitar a compreensão e manutenção dos sistemas (TSANG; CLARKE; BANIASSAD, 2004).

Foi proposta uma abordagem para desenvolver aplicações adaptativas para *middlewares* que necessitem de QoS (*Quality of Service*). Essa abordagem separa os comportamentos de QoS e de adaptação dos comportamentos funcionais e distribuídos (DUZAN, 2004).

Uma abordagem baseada em POA para a realização de teste funcional de software é usada na construção da ferramenta AFTT – *Aspect-based Functional Testing Tool.* A ferramenta oferece suporte a alguns critérios funcionais e utiliza recursos do AspectJ para permitir a análise de cobertura de teste funcional (ROCHA *et al*, 2004).

Um trabalho onde POA é utilizada para inserir segurança em aplicações é apresentado em (WIN; VANHAUTE; DECKER, 2001). Através do exemplo de controle de acesso em uma aplicação, é investigado como a orientação a aspectos pode tratar com a separação de interesses de segurança dos interesses funcionais da aplicação.

É importante salientar que essa lista não é completa e visa apenas ilustrar algumas áreas em que a Programação Orientada a Aspectos está sendo utilizada.

3.6.2 Programação Orientada a Aspectos na Instrumentação de Código

Vários trabalhos vêm sendo desenvolvidos com POA para instrumentação de código, porém, nenhum específico para injeção de falhas.

Como citado anteriormente, a instrumentação consiste em inserir código auxiliar no programa que está sendo executado, permitindo observar o comportamento do programa e obter informações ou medidas referentes à sua execução. Essas informações permitem conhecer, por exemplo, o estado do sistema e os métodos que foram executados durante e execução da aplicação.

Algumas ferramentas como JIE - *Java Instrumentation Engine* - permitem a instrumentação diretamente sobre o código fonte, por meio da introdução de comandos (ou anotações) de instrumentação em locais específicos. A instrumentação do código fonte possui um nível maior de abstração, pois é feita no nível da linguagem de programação. No entanto, além de exigir a disponibilidade do código, esse tipo de abordagem leva à manutenção de dois códigos fontes: um instrumentado e outro não (TROMER apud ROCHA *et al*, 2004). Ferramentas como Javassist (CHIBA, 2004) e

BCEL (DAHM, 2002) permitem manipulação do bytecode Java e introdução de código instrumentado por meio da inserção de instruções de baixo nível. A instrumentação do *bytecode*, por sua vez, apesar de eliminar os problemas citados anteriormente, muitas vezes exige o conhecimento da estrutura do *bytecode* e, portanto não é muito intuitiva.

Alguns autores têm usado a POA para instrumentar programas, com diversas finalidades. Em Deters (DETERS; CYTRON, 2001), POA vem sendo utilizada para instrumentar códigos de aplicações com o objetivo de realizar análise dinâmica de código, isto é, obter informações de tempo de execução dos programas, tais como estado do programas, memória, etc.

A ferramenta Dyjit - Dynamic Java Instrumentation Toolkit – foi desenvolvida para ser um framework que tem por objetivo instrumentação dinâmica, isto é, inserção e remoção de trechos de códigos em tempo de execução (PEARSON, 2003).

O JSpy é um pacote para instrumentação de *bytecodes* Java e análise de dados de execução. Esse pacote pode ser visto como ambiente de POA, visto que é guiado por regras, ou aspectos, que especificam como o programa deve ser transformado para alcançar a funcionalidade adicional. Entretanto, o objetivo principal desses aspectos é extrair informação de um programa em execução (GOLDENBERG; HAVELUND, 2005).

Apesar da instrumentação de código estar sendo utilizada em diversas áreas, ainda são desconhecidas abordagens específicas para instrumentação de programas visando especificamente a injeção de falhas para a validação de sistemas distribuídos com características de dependabilidade.

4 USO DE ORIENTAÇÃO A ASPECTOS PARA INJEÇÃO DE FALHAS

Esse capítulo apresenta como a Programação Orientada a Aspectos pode ser utilizada na construção de ferramentas de injeção de falhas de comunicação, utilizando a técnica de instrumentação dinâmica. São apresentados os recursos oferecidos pelo paradigma de orientação a aspectos e as vantagens de se utilizar o mesmo.

4.1 Motivação para o uso de Orientação a Aspectos

A injeção de falhas possui um comportamento que abrange os diversos módulos da aplicação alvo, afetando métodos que são executados em diversas classes em diversos pontos da aplicação. Desta forma, a injeção de falhas pode ser encapsulada sob a forma de aspectos.

A POA é uma alternativa para impedir a alteração do código fonte das aplicações a serem testadas, evitando assim intrusividade espacial. É oferecido um mecanismo para a injeção de falhas de alto nível em sistemas orientados a objetos, possibilitando a instrumentação de métodos, classes e variáveis de um sistema, em tempo de compilação, carga ou execução, sem a necessidade de alterar a estrutura do sistema, tornando indiferente a disponibilidade do código fonte para o teste. Dessa forma, é necessário apenas o mínimo de informação sobre a aplicação, como classes, métodos e nomes de atributos.

O uso de POA permite que os instrumentos sejam encapsulados em um aspecto e introduzidos em nível de *bytecodes*. É o aspecto que identifica as chamadas aos métodos que devem ser modificados e instrumenta todo o código ou parte dele em tempo de carga e execução. O encapsulamento da lógica de injeção de falhas em um único módulo que afeta as diversas classes do sistema, separando o código funcional do código de injeção, facilita a inserção e remoção dos aspectos na aplicação alvo e auxilia na criação rápida de diversos cenários de falhas. Além disso, caso o sistema deva ser executado sem o injetor de falhas (por já ter sido validado e colocado em produção, por exemplo), basta que se remova o aspecto responsável por essa funcionalidade, conservando intacto o programa original.

As linguagens orientadas a aspectos oferecem recursos que podem ser aplicados em diversas atividades de teste. Um exemplo é a instrumentação de programas utilizando linguagens como o AspectJ (KICKZALES *et al*, 2001) ou AspectWerkz

(BONNÉR, 2004), que implementam mecanismos que facilitam o acesso aos objetos, atributos, parâmetros e resultados de métodos e permite que determinadas verificações, muitas vezes difíceis de ser executadas usando apenas recursos da orientação a objetos, sejam simplificadas (ROCHA *et al*, 2004). Essas características tornam a POA uma técnica útil para a injeção de falhas, pois a interceptação da execução de métodos e a captura de seus contextos (incluindo parâmetros e resultados) são essenciais para a validação de sistemas distribuídos com características de dependabilidade.

4.2 Recursos de Orientação a Aspectos para injeção de falhas

A Programação Orientada a Aspectos complementa a Programação Orientada a Objetos por facilitar um outro tipo de modularidade que expande a implementação de um interesse, que está espalhado por diversos pontos da aplicação, dentro de uma simples unidade que encapsula os comportamentos que afetam diversas classes, na forma de um módulo reusável. Essa unidade é chamada de aspecto e pode-se dizer que ela é um módulo ortogonal aos outros módulos do sistema.

A POA oferece recursos que possibilitam a criação de uma abordagem para injeção de falhas bastante modular e reusável, visto que, essas características são muito importantes em ferramentas de injeção de falhas. A tabela 5.1 apresenta um comparativo entre alguns conceitos de injeção de falhas com a POA.

Tabela 4.1 - Comparativo Injeção de Falhas x POA

Injeção de Falhas	**POA**
Métodos que devem ser instrumentados	Join points / Pointcuts
Lógica de injeção de falhas (instrumentação de métodos)	Advices
Injetor de Falhas	Aspecto

A tabela 4.1 mostra que os principais elementos de um injetor de falhas podem ser diretamente mapeados para quatro (de cinco) dos principais conceitos de POA. Os métodos da aplicação que devem ser instrumentados para sofrer a injeção de falhas se encaixam nos conceitos de *join point* e *pointcut.* Os *join points* são os métodos em si, e os *pointcuts* sinalizam quais os métodos devem ser instrumentados. Ao serem atingidos esses métodos, eles tem seus comportamento alterado pela lógica de injeção de falhas que é implementada nos *advices* do aspecto, ou seja, os *advices* são definidos em termos de *pointcuts* e implementam o comportamento que deve ser executado quando os *pointcuts* são alcançados. O código do *advice* é executado cada vez que um *join point* é capturado por algum *pointcut*. Por fim, o aspecto encapsula os elementos anteriores, dando forma ao injetor de falhas.

Como citado anteriormente, uma ferramenta de injeção de falhas por software geralmente é um trecho de código que usa todos os ganchos do processador para criar um comportamento errôneo de maneira controlada (CARREIRA; LEITE; WEBER, 1999), o que fortalece as relações estabelecidas na tabela 4.1.

A abordagem básica deste trabalho é a implementação de aspectos que definem *pointcuts* em termos das primitivas de comunicação das aplicações e protocolos,

procedimento necessário para a injeção de falhas em aplicações distribuídas de rede. A POA é uma alternativa viável e eficiente, pois torna possível a construção de aspectos que interceptam, em um único ponto, os métodos que são chamados de qualquer lugar da aplicação. Além disso, as classes da aplicação não precisam ser alteradas de forma alguma, nem ao mesmo para a inserção de cláusulas de instanciação ou inserção de bibliotecas adicionais, conservando intacto o programa original. A união das classes da aplicação com os aspectos é realizada pelo *weaver* da ferramenta de programação orientada a aspectos que está sendo utilizada.

4.3 Vantagens da Orientação a Aspectos para Injeção de Falhas

Além de encapsular toda a lógica de injeção de falhas em um único módulo que afeta as diversas classes do sistema, outra vantagem do uso de POA é que sua funcionalidade é plugável, isto é, caso o sistema deva ser executado sem o injetor de falhas, basta que se remova o aspecto responsável por essa funcionalidade.

A funcionalidade plugável de POA resulta em outro benefício importante na construção de injetores de falhas. Sistemas de missão crítica, que exigem alta disponibilidade, devem ser fortemente validados antes de serem colocados em produção, pois podem até mesmo envolver risco a pessoas (por exemplo, sistemas de controle aéreo). Porém, além desses, existem sistemas distribuídos sem um alto grau de risco, mas que nem por isso podem deixar de ser validados. Muitas vezes por falta de tempo no desenvolvimento, ou na falta de ferramentas flexíveis que possam ser facilmente usadas em várias aplicações, a fase de validação dos mecanismos de tolerância a falhas fica relegada a um segundo momento, até mesmo já em fase de manutenção. Um injetor de falhas, altamente reusável e facilmente (des)plugável é de valor inquestionável, pois amplia o uso da injeção de falhas como mecanismo de validação, fazendo que sistemas sejam mais facilmente validados e ofereçam maior segurança de uso.

É importante ressaltar que uma estratégia baseada em POA simplifica o desenvolvimento da ferramenta de injeção de falhas, possibilitando a instrumentação de código em linguagem de programação de alto nível, ao contrário do uso de JVMTI em FIONA (SILVA, 2004-b) que exige que parte do injetor seja implementada na linguagem C. Esse ponto tende a facilitar a expansão da ferramenta desenvolvida para outros tipos de falhas.

Além disso, a modularidade e plugabilidade de POA auxiliam na redução da intrusividade espacial. A instrumentação pela alteração direta do código fonte da aplicação alvo é considerada intrusão espacial, podendo inclusive ser fonte de inserção de erros não desejados, alterando a semântica da aplicação que está sendo validada. Vários estudos vêm sendo desenvolvidos para evitar a intrusividade gerada pela inserção de código de injeção de falhas nas aplicações a serem validadas. Como citado na seção 2.5, sobre ferramentas de injeção de falhas, técnicas como reflexão computacional e utilização de interfaces de programação nativa vêm sendo utilizadas no desenvolvimento de soluções não intrusivas.

Para melhor exemplificar o problema da intrusividade e a solução obtida com POA, serão utilizados dois diagramas. A figura 4.1 apresenta um diagrama com três classes. Uma classe que oferece a funcionalidade de envio e recepção de mensagens e

duas classes que utilizam essa funcionalidade. Uma abordagem intrusiva para injeção de falhas poderia seguir dois caminhos:

- Alteração das classes A e B para inserção de código instrumentado quando as chamadas de envio e recepção da classe `ProtocoloComunicacao` fosse chamadas;

- Alteração direta da classe `ProtocoloComunicacao` nos métodos de envio e recepção.

Para o primeiro caso, o código de injeção estaria espalhado e cruzando várias classes. Essa intrusão dificultaria a inserção e remoção do código relativo a injeção de falhas, além da necessidade de recompilação dessas classes.

No caso da instrumentação da classe `ProtocoloComunicacao`, o código de injeção estaria em apenas uma classe, porém, continuaria intrusivo. Os mesmos problemas de plugabilidade e recompilação seriam enfrentados. Assim, mostra-se que essa forma comum de injeção de falhas, independente do caminho utilizado é ineficaz.

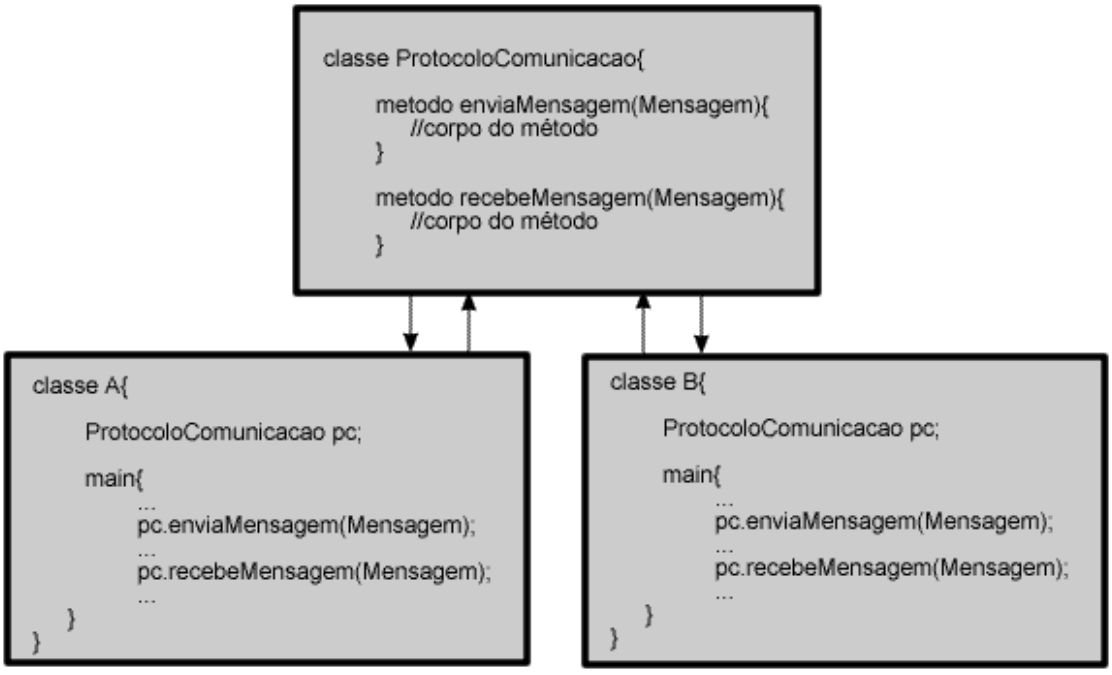

Figura 4.1 - Diagrama sem Injeção de Falhas

A figura 4.2 mostra como ficaria o diagrama com a utilização de POA. É criado um aspecto que serve como uma espécie de interceptador. As classes A e B não são alteradas, nem a classe `ProtocoloComunicacao`. O aspecto é colocado de forma transparente (pela ferramenta de POA, que se encarrega de integrar o aspecto e as classes, em tempo de compilação, execução ou carga) e encapsula a lógica de injeção de falhas e as condições em que a injeção deve ocorrer. Quando os métodos que podem disparar alguma falha são atingidos (os *join points* definidos em *pointcuuts*) o *advice* ligado a esse *join point* é executado. Caso a condição para injeção seja satisfeita o aspecto executa o código instrumentado, caso contrário o fluxo é devolvido a classe que implementa os métodos originais.

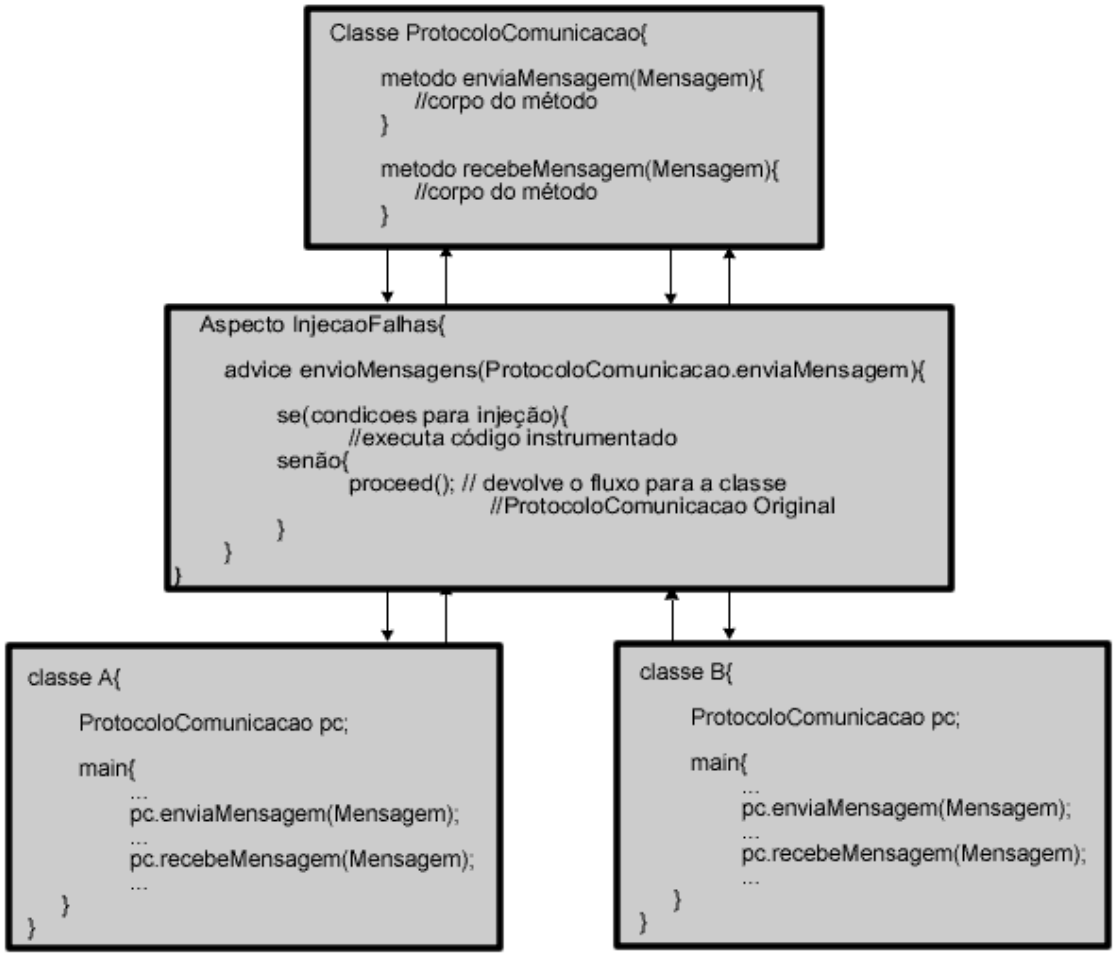

Figura 4.2 - Abordagem POA

Para demonstrar a validade da abordagem proposta, foi implementada a ferramenta FICTA. Utilizou-se a linguagem Java e para a implementação dos aspectos foi utilizado o *framework* de POA AspectWerkz. Cada tipo de falha considerada no modelo de falhas é implementado como um *advice* no aspecto de injeção de falha e para realizar a injeção é realizada a interceptação e instrumentação de primitivas de comunicação do protocolo UDP. As seções a seguir apresentam a ferramenta, o modelo de falhas abrangido, a implementação, a demonstração e os testes realizados com a ferramenta.

4.4 Limitações da Orientação a Aspectos para Injeção de Falhas

Existem duas principais desvantagens em relação ao uso de POA no desenvolvimento de uma estratégia de injeção de falhas. A primeira está relacionada a necessidade de um contêiner de aspectos (ou no caso de *weaving offline*, uma segunda fase de compilação) e a segunda está relacionada ao desempenho do injetor de falhas.

A necessidade de uma ferramenta auxiliar para a utilização da orientação a aspectos pode ser vista como uma desvantagem. No caso no trabalho em questão, é utilizado *weaving* em tempo de execução, o qual necessita que o contêiner de POA

oferecido pela ferramenta realize modificações do carregador de classes de sistema da máquina virtual Java, para que os aspectos possam ser utilizados.

Outra limitação imposta pelo uso de *weaving* em tempo de execução relaciona-se a intrusividade temporal que o injetor insere na aplicação alvo. Como é realizada a alteração de *bytecodes* em tempo de execução, a cada chamada de método instrumentado pelos aspectos, existe um atraso inserido pela troca de contexto realizada. O carregador de classes do sistema identifica que os aspectos devem ser executados e transfere o controle para o contêiner de POA que está ligado a ele, que então verifica o que deve ser executado. Caso o injetor tivesse como objetivo a validação de aplicações com restrições temporais (sistemas de tempo real), talvez a estratégia tivesse que ser revista.

O capítulo 6, sobre Testes e Experimentos, apresenta algumas medidas relativas a intrusividade temporal nos testes realizados com FICTA.

4.5 Modelo Geral de Injetor de Falhas

A estratégia para injeção de falhas utilizada nesse trabalho é baseada no modelo genérico de ambiente de injeção de falhas proposto por Hsueh (HSUEH, 1997) apresentado na figura 4.3.

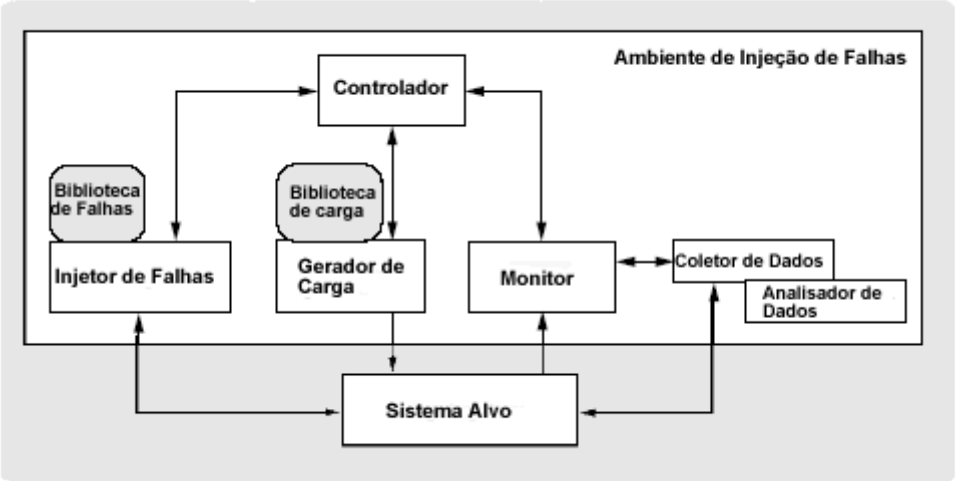

Figura 4.3 - Ambiente de Injeção de Falhas proposto por Hsueh

A figura 4.3 mostra os componentes de um ambiente de injeção de falhas e suas relações. Um ambiente genérico pode ter seus componentes em hardware ou software ou em uma mistura de hardware e software. Para o caso menos genérico de injeção de falhas por software, a seta com origem no injetor de falhas e destino no sistema alvo representa a instrumentação do código deste sistema alvo que deve sofrer a injeção de falhas. No caso específico de injeção de falhas de comunicação, essa seta representa as primitivas que a aplicação alvo utiliza para troca de mensagens (por exemplo, métodos *send* e *receive*). Em aplicações distribuídas baseada em pilhas de protocolos, podem existir vários níveis de trocas de mensagens, podendo a injeção de falhas agir em determinados níveis, ou algum nível específico. O nível da injeção de falhas é uma escolha conjunta entre controlador e o injetor de falhas, obedecendo ao modelo de falhas do sistema alvo e a carga de falhas especificada de acordo com a Biblioteca de

Falhas pelo engenheiro de teste que vai conduzir dado experimento sobre o sistema alvo. A Programação Orientada a Aspectos proporciona a instrumentação em qualquer nível da aplicação de forma simples, permitindo que as primitivas a serem instrumentadas sejam pré-definidas ou sejam escolhidas através de algum programa ou arquivo que permita a parametrização do Injetor de Falhas.

O gerador de carga para o sistema a ser testado se relaciona com o controlador do ambiente de injeção de falhas e com o sistema alvo. O sistema alvo deve estar executando tarefas com perfil de carga tão próximo quanto possível do ambiente de operação real. O gerador de carga pode, por exemplo, simular transações ou solicitações de serviço ao sistema alvo.

A carga de falhas deve ser criada a partir desta definição em função do modelo de falhas determinado para o sistema alvo. Por exemplo, em injetores de falhas de comunicação utiliza-se um modelo de falhas baseado em troca de mensagens ou baseado em protocolos de mais baixo nível, como TCP ou UDP. A carga de falhas define quantas mensagens são afetadas, quando as mensagens são afetadas e de que forma. Dependendo do sistema alvo e das características do injetor, a carga de falhas pode ser definida a partir de um arquivo texto. Esse arquivo pode ser gerado manualmente, através de alguma interface gráfica ou através de uma linguagem de mais alto nível. A ferramenta ORCHESTRA (DAW, 1996) utiliza scripts para a definição de carga de trabalho, e a ferramenta FIONA (SILVA, 2004) utiliza uma interface gráfica com o usuário. Uma alternativa é o uso de arquivos XML para a definição dos parâmetros da carga de falha, visto que é um formato que facilitaria uma distribuição para vários injetores diferentes operando em um mesmo experimento de teste na validação de sistemas heterogêneos distribuídos.

O mecanismo de monitoramento de dados (coleta e análise) também pode utilizar os recursos da Programação Orientada a Aspectos, complementando todo o ambiente de injeção de falhas e permitindo a extração automática de métricas de dependabilidade (como cobertura de falhas, queda de desempenho sob falhas, tempo médio de recuperação entre outros), entretanto, esses mecanismos fogem ao escopo deste trabalho no trabalho.

A figura 4.4 apresenta o ambiente de injeção de falhas utilizado na estratégia orientada a aspectos proposta pelo trabalho. O ambiente implementa a parte essencial do ambiente proposto por Hsueh necessária para a condução de experimentos de validação de protótipos, excluindo os componentes de monitoramento.

O gerador de carga não foi necessário, pois os sistemas alvos que são abordados neste trabalho são as próprias aplicações que devem ser validadas através de injeção de falhas e suas cargas de trabalho são impostas por serviços que essas aplicações prestam a um usuário durante os experimentos.

O controlador é componente responsável por identificar os momentos em que as falhas devem ser injetadas. A carga de falhas é um arquivo que define quantidades e tipos de falhas a serem injetadas. Deve ser observado que o ambiente não exclui o monitoramento que pode ser desenvolvido e acrescentado de forma independente ao injetor de falhas, pois não tem com esse qualquer interação.

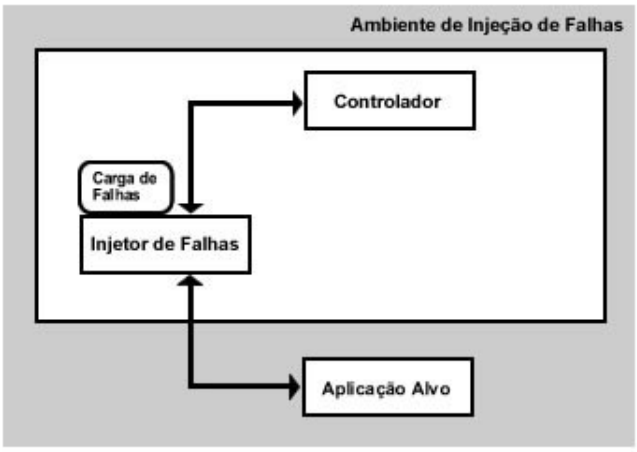

Figura 4.4 - Ambiente de Injeção de Falhas utilizado na estratégia proposta pelo trabalho.

O capítulo 5 apresenta o ambiente de injeção de falhas proposto a partir do modelo de Hsueh, mapeado para a utilização de Programação Orientada a Aspectos.

5 FICTA: FAULT INJECTION COMMUNICATION TOOL BASED ON ASPECTS

FICTA (*Fault Injection Communication Tool based on Aspects*) é uma ferramenta que tem como principal objetivo a injeção de falhas de comunicação em aplicações Java distribuídas, utilizando para isso a programação orientada a aspectos. O resultado esperado na aplicação da ferramenta é que a aplicação distribuída que está sendo validada, que tenha sido construída usando técnicas de tolerância a falhas para detecção e recuperação de erros, responda corretamente, mesmo na presença de falhas.

A ferramenta é implementada na linguagem Java e como suporte para a Programação Orientada a Aspectos é utilizado o framework AspectWerkz.

Esse capítulo apresenta os sistemas alvo que a ferramenta pretende atingir, a arquitetura da ferramenta, modelo de falhas abrangido, como é realizada a ativação das falhas e também detalhes sobre a implementação.

5.1 Introdução

Para a abordagem apresentada nesse trabalho, interessam as falhas de comunicação que podem ocorrer em sistemas distribuídos. Estas afetam mensagens que são transmitidas através de um canal de comunicação, que podem ser omitidas, duplicadas ou entregues com atraso (JACQUES, 2004-a). Normalmente, a injeção de falhas em sistemas distribuídos é realizada através da instrumentação das primitivas de comunicação (envio e recepção de mensagens) oferecidas pelo sistema. Apesar de estarem implementadas, muitas vezes, em uma única classe, suas chamadas se encontram espalhadas em diversos pontos da aplicação. Pode ser necessário, além de alterar a classe que implementa as primitivas de comunicação, inserir ganchos em cada chamada de método.

A POA é uma alternativa viável, pois torna possível a construção de aspectos que centralizam a interceptação de métodos que são chamados de qualquer lugar da aplicação. A partir dessa interceptação é decidido se os métodos que devem ser executados são os da classe original ou os métodos instrumentados, através de regras baseadas no modelo de falhas do injetor. Além disso, as classes da aplicação não precisam ser alteradas de forma alguma, nem mesmo para a inserção de cláusulas de instanciação ou inserção de bibliotecas adicionais, pois toda a lógica de injeção de

falhas se encontra no aspecto e as definições de quais métodos devem ser interceptados estão num arquivo XML.

Basicamente, FICTA implementa aspectos que definem *pointcuts* em termos das primitivas de comunicação das aplicações e protocolos, e implementam *advices* que substituem os códigos originais por códigos instrumentados com a lógica de injeção de falhas.

5.2 Sistemas Alvo

FICTA tem como sistemas alvo, aplicações de rede escritas em Java sobre o protocolo UDP. O protocolo UDP (*User Datagram Protocol*) é um protocolo que envia pacotes de dados independentes, chamados datagramas, de um computador a outro, porém sem garantias sobre a chegada desses pacotes (SUN, 2005).

O protocolo UDP é não confiável e não orientado a conexão, no entanto é comumente usado como base para a implementação de protocolos que implementam a confiabilidade necessária através de camadas superiores adicionais. Um exemplo é o *middleware* de comunicação de comunicação de grupo JGroups (BAN, 1998) que usa UDP por padrão na base de sua pilha de protocolos.

Outro exemplo é o *framework* de comunicação Appia. Appia é um protocolo de kernel que oferece suporte a aplicações que necessitam de vários canais de comunicação coordenados, oferecendo informação consistente sobre falhas de nodos remotos (MIRANDA; PINTO; RODRIGUES, 2001).

5.3 Arquitetura e Funcionamento

A arquitetura de FICTA é composta basicamente de um contêiner para POA que suporte *weaving* de classes e aspectos em tempo de carga e execução, um aspecto chamado de *FaultInjector,* que encapsula os códigos instrumentos, e um arquivo XML que define quais os métodos devem ser instrumentados, isto é, devem sofrer a injeção de falhas. Para FICTA o contêiner utilizado é oferecido pelo *framework* AspectWerkz.

Como pode ser observado na figura 5.1, que ilustra a arquitetura e o fluxo de controle de FICTA, o contêiner POA modifica o *Class Loader* de sistema da JVM. Essa modificação é necessária para que seja possível a identificação dos métodos que devem ser instrumentados. É importante salientar que essa modificação não é permanente e só acontece quando a aplicação alvo for executada para testes.

No momento de carga das classes da aplicação e do injetor, o *Class Loader* de Sistema, modificado pelo contêiner POA, realiza a identificação das classes que terão seus métodos alterados pelo injetor de falhas. Essas classes e métodos são definidos em um arquivo XML usado pelo contêiner. No momento que a aplicação alvo cria uma instância da classe `java.net.DatagramSocket` o contêiner verifica a existência da classe no arquivo XML e insere ganchos nos métodos a serem instrumentados (`send` e `receive`). Esses ganchos são os *join points* do injetor de falhas orientado a aspectos. A partir desse momento, `java.net.DatagramSocket` está preparada para sofrer a injeção de falhas.

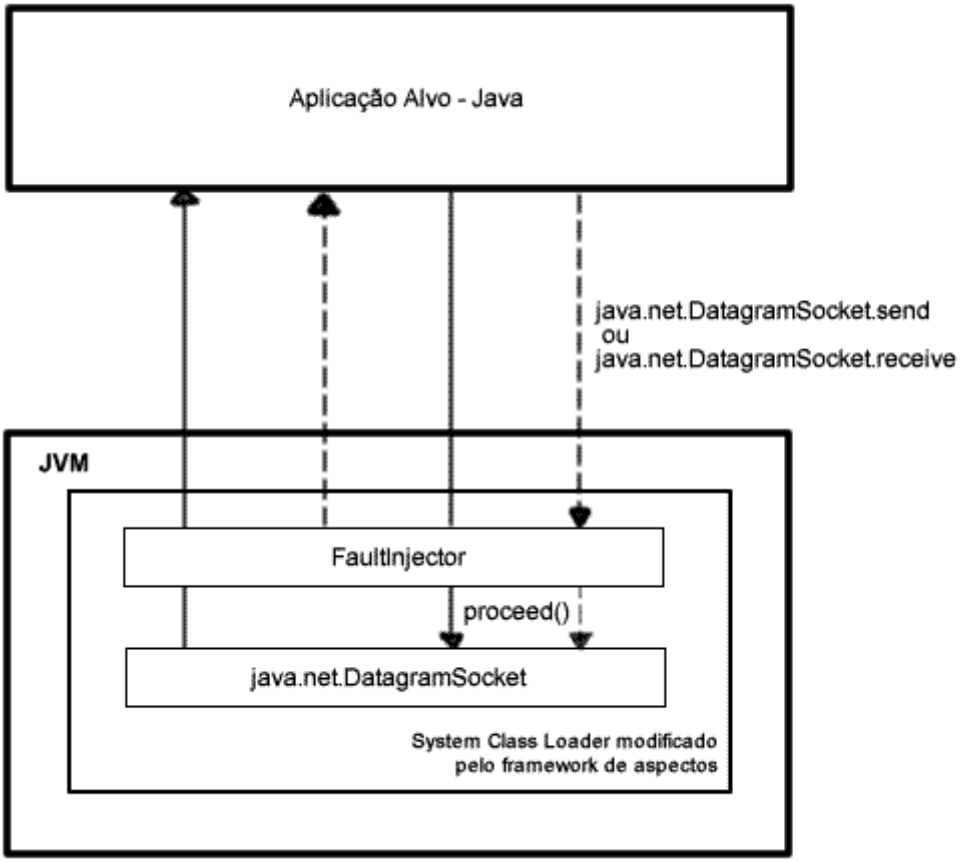

Figura 5.1 - Arquitetura e fluxo de controle da ferramenta FICTA

A aplicação alvo executa normalmente até a chamada de um `send` ou `receive` de `java.net.DatagramSocket`. As setas pontilhadas, na figura 5.1, mostram o fluxo executado quando o aspecto *FaultInjector* identifica a chamada a um método que será instrumentado. O contêiner POA identifica o *join point* (através do arquivo de definição em XML), recolhe informações sobre o mesmo e determina quais os *advices* que estão ligados a ele. Nesse momento, o injetor de falhas dispara um método privado chamado `trigger()`,responsável por verificar se a falha deve ser disparada naquele momento (esse método será detalhado na seção 6.5, sobre ativação de falhas). Caso o método retorne o valor booleano verdadeiro, a falha é injetada, ou seja, é executado o *advice* do injetor de falhas, responsável pela instrumentação do método que foi chamado pela aplicação alvo. Caso o valor retornado pelo método `trigger()` seja falso, então o fluxo é demonstrado pelas setas continuas da figura 5.1. O injetor de falhas identifica que não é o momento de injeção e chama o método `proceed()`, que devolve o fluxo de execução para a classe `java.net.DatagramSocket`, que então, é executada normalmente.

5.4 Modelo de Falhas

O modelo de falhas considerados em FICTA é baseado no modelo de falhas para sistemas distribuídos definido por Cristian (CRISTIAN, 1991) e definidos no capítulo 2.

Do conjunto descrito por Cristian, esse trabalho considera as falhas de colapso e omissão, por serem consideradas as mais comuns. São consideradas as falhas de omissão no envio de mensagens. O nodo da rede que está executando o injetor de falhas omite o envio de mensagens para todos os outros nodos envolvidos no processo. No atual estado da ferramenta, não é possível especificar um endereço de destino ao qual a mensagem deve ser omitida, porém, está prevista essa extensão através da utilização dos atributos de *runtime* oferecidos pelo AspectWerkz.

As falhas de colapso consideradas são as falhas de colapso de parada, onde o nodo que executa o injetor de falhas, ao sofrer o colapso, não reinicia seus serviços. Esse tipo de colapso é simulado quando o processo que está executando o injetor é morto de forma repentina.

As falhas de temporização e resposta não são consideradas, pois estão fortemente relacionadas com a semântica das aplicações alvo. As falhas de temporização estão associadas à semântica de sincronismo (sistemas síncronos ou assíncronos) e as falhas de resposta estão associadas ao conteúdo das mensagens trocadas pela rede. Como trabalho futuro, é possível estudar a possibilidade do uso dos métodos do *framework* AspectWerkz, que buscam atributos de tempo de execução, para que seja possível identificar atributos de tempo e de conteúdo das aplicações alvo.

5.5 Cenário de Falhas

O cenário de falhas possui definições sobre o momento de injeção e quais os tipos de falhas a serem injetadas. Em FICTA, todos os parâmetros de configuração que compõem o cenário de falhas ou *faultload* são especificados em um arquivo texto. Esse arquivo é carregado quando as classes e aspectos estão sendo carregados e as configurações são armazenadas em variáveis membros do aspecto. A figura 5.2 exemplifica o formato desse arquivo.

```
#parameters
perc=50
type=OMISSION
expType=DETERMINISTIC
```

Figura 5.2 - Exemplo de arquivo de configuração

O parâmetro `perc` identifica o percentual de mensagens que devem ser omitidas, ou após quantas mensagens o processo deve sofrer um colapso. O parâmetro `type` identifica se a falha é de omissão ou colapso (`OMISSION` ou `CRASH`, respectivamente). Por último o parâmetro `expType`, onde é definido se o experimento é determinístico ou aleatório (`DETERMINISTIC` ou `RANDOM`, respectivamente). Atenção especial deve ser dada para a escolha de `RANDOM`. Quando `expType` é configurado para `RANDOM`, será escolhido um valor aleatório para os parâmetros anteriores caso algum deles, ou ambos, não tenha sido preenchido. Ou seja, `RANDOM` não sobrescreve os outros parâmetros, apenas complementa.

Através da alteração dos parâmetros do arquivo de configuração, é possível ativar diferentes regras no injetor. Um mesmo cenário pode ser utilizado para validar várias aplicações.

5.6 Implementação da Ferramenta

Na implementação de FICTA, cada uma das falhas consideradas no modelo foi mapeada como um *advice* no aspecto chamado `FaultInjector`. Portanto, são encontrados dois *advices* principais e métodos auxiliares. Os *advices* trabalham com as primitivas de comunicação, interceptando chamadas de métodos de envio e recepção de mensagens, construtores, objetos e variáveis de forma que seja possível a alteração dos mesmos para a emulação de falhas.

No geral, para realizar os experimentos, cada aspecto possui no seu XML *pointcuts* que identificam as chamadas de métodos (*joint points*) de envio e recepção de mensagens. Ao serem atingidos esses *joint points*, existem *advices* responsáveis pela lógica adicional a ser inserida durante a execução do sistema alvo. Para as diferentes falhas, existem pequenas diferenças nos *pointcuts.* As grandes diferenças se encontram no código presente nos *advices*, pois são eles que instrumentam efetivamente os métodos.

A extensão de FICTA pode ser realizada de duas formas, ambas muito simples. A primeira é alterando a própria classe `FaultInjector` para a inserção de métodos adicionais relativos aos novos tipos de falhas que devem ser injetados. A segunda forma é estendendo a classe `FaultInjector`, para que sejam herdados os métodos originais e sejam definidos os novos métodos. A segunda forma evita a recompilação da classe `FaultInjector`. Ambos os métodos de extensão necessitam que o arquivo XML seja alterado, para inserir os novos *pointcuts* relativos aos novos tipos de falhas, identificando quais métodos serão afetados.

FICTA implementa injeção de falhas de comunicação através da instrumentação da classe `java.net.DatagramSocket` da API padrão da linguagem Java. É essa classe que permite o envio e recebimento de pacotes através do protocolo UDP. A instrumentação de uma classe base da linguagem garante a reutilização para toda e qualquer aplicação, protocolo ou *middleware* escrito em Java e baseada no protocolo UDP. Portanto, qualquer aplicação Java construída sobre essa classe pode ser validada utilizando a ferramenta FICTA. Caso a ferramenta tivesse sido construída para instrumentar uma classe específica de alguma aplicação alvo, não seria genérica o suficiente para garantir reutilização. No momento que se instrumenta uma classe da aplicação, restringe-se a ferramenta apenas a aquela aplicação. Nesse contexto, por aplicação entende-se desde sistemas desenvolvidos diretamente com a API, protocolos, *middlewares* e *toolkits* que se utilizam dessa classe para a construção de outros sistemas.

A classe Java `java.net.DatagramPacket` representa um datagrama UDP. Datagramas são utilizados para a realização de entregas de pacotes em sistemas sem conexão e normalmente incluem o endereço destino e informação de porta. A classe `java.net.DatagramSocket` é um *socket* utilizado para o envio de datagramas em uma rede pelo protocolo UDP. Um datagrama é enviado por um `java.net.DatagramSocket` pela chamada do método `send(DatagramPacket dp)`. O método `receive(DatagramPacket dp)` é utilizado para realizar a recepção de um datagrama.

A classe `java.net.MulticastSocket` também pode ser utilizada para envio e recebimento de datagramas para um grupo *multicast*. Essa classe é subclasse de

`java.net.DatagramSocket` e adiciona as funcionalidades para *multicasting*. Com FICTA também é possível validar protocolos que utilizam *multicast* UDP, já que a classe `java.net.MulticastSocket` é uma classe filha de `java.net.DatagramSocket` e não reimplementa os métodos `send` e `receive`. Ou seja, a injeção de falhas acontece transparentemente, através do mesmo mecanismo.

5.6.1 Decisões de Projeto

Como citado no capítulo 3, o *framework* AspectWerkz foi escolhido para a implementação de FICTA devido a possibilidade de *weaving* em tempo de execução. A possibilidade de integrar o código de injeção de falhas nos *bytecodes* sem alterar o código fonte da aplicação fortaleceu essa decisão. Além disso, o *weaving* em tempo de execução não necessita de outras fases de compilação para a inserção dos aspectos na aplicação alvo.

A utilização do *framework* AspectWerkz permite que não seja necessária a alteração direta de nenhum código fonte, nem das aplicações de carga, nem da classe alvo (`java.net.DatagramSocket`). Isso é de grande importância, pois existem casos que os códigos fontes não estão disponíveis. Além disso, evita-se a intrusão espacial do código fonte, que pode causar alteração de semântica introduzindo erros não desejados. A intrusividade acontece apenas nos *bytecodes* da aplicação, ou seja, apenas no momento da execução. A lógica de injeção de falhas fica modularizada em um aspecto e a integração dos aspectos com as classes é realizada pelo framework, de forma transparente, que também é responsável pela modificação do carregador de classes de sistema que dá suporte a integração em tempo de carga e tempo de execução.

Optou-se pelas definições dos aspectos da ferramenta de injeção de falhas através de arquivos XML ao invés de anotações. Ao usar as anotações, a alteração dos *pointcuts* teria que ser feita no próprio aspecto, necessitando de recompilação. Com o uso do arquivo XML, o aspecto não necessita de alterações e nem recompilação, tornando a ferramenta mais flexível e reusável.

5.6.2 Ativação de Falhas

Uma abordagem comumente usada para a ativação de falhas de comunicação é disparar a falha durante o envio ou recepção de uma mensagem (DAWSON; JAHANIAN; MITTON, 1996).

FICTA utiliza como ativador para o experimento de injeção de falhas a quantidade de mensagens enviadas e recebidas por um membro do grupo. Essa quantidade de mensagens pode ser pré-definida, isto é, definida pelo usuário que está realizado o experimento, ou aleatória, onde o próprio injetor gera um valor para a quantidade de mensagens. A quantidade de mensagens é considerada tanto para falhas de omissão quanto de colapso. Esses parâmetros são definidos no arquivo de configuração apresentado na seção 5.5 sobre cenário de falhas.

Os parâmetros contidos no arquivo de configuração são carregados no momento que o aspecto é carregado e armazenados em variáveis membro da classe *FaultInjector.* Eles serão utilizados pelo método `trigger()` do injetor. Esse método é responsável por verificar se a falha deve ser inserida. Quando um *join point* é alcançado (envio ou recepção de mensagens) é passado como parâmetro para `trigger()` o número de

mensagens que já foram enviadas ou recebidas até o momento. Com esse valor, e o valor dos parâmetros do arquivo de configuração é verificado se é momento da falha ser injetada. O valor retornado é um booleano (verdadeiro ou falso). Caso verdadeiro, o *advice* insere o código instrumento, substituindo o corpo do método `send` ou do método `receive` (de acordo com o que foi configurado no XML).

Para as falhas de omissão, é definida uma porcentagem de mensagens que deverão ser omitidas de acordo com o número de mensagens enviadas, por exemplo, 30% das mensagens enviadas devem ser omitidas. Para as falhas de colapso, a quantidade de mensagens enviada sinaliza o momento em que o colapso deve ser simulado, por exemplo: "o nodo deve ter uma falha de colapso injetada após ter enviado 50 mensagens".

5.6.3 Falhas de Colapso de Parada

Para as falhas de colapso de parada, foi implementado um *before advice*, chamado `crashHalt`. Um *before advice* executa logo após os argumentos do método terem sido avaliados, porém, antes do corpo do método executar. Caso a resposta do método `trigger()` for verdadeira, o método de colapso é executado.

Para simular o colapso de parada, através do método `crashHalt` o *advice* mata o processo da JVM em que o mesmo está rodando. Para matar a JVM, executa-se uma chamada Java ao método `Runtime.getRuntime().halt(n)`, da API padrão Java, com *n* diferente de zero. O método *halt*, com qualquer parâmetro diferente de zero, termina forçadamente a execução da Máquina Virtual Java que está executando, emulando então colapso do membro. O retorno desse método nunca é normal. A opção pelo uso desse método foi pela sua portabilidade.

```
public void crashHalt(JoinPoint joinPoint) throws Throwable {
      count++;

      //verificações de runtime

      if(trigger(count)){

            Runtime.getRuntime().halt(10);

      }
}
```

Figura 5.3 - Trecho do código instrumento para colapso de parada

```
<aspectwerkz>
 <system id="id" >
   <package name="ficta">
    <aspect class="FaultInjector">
            <pointcut name="crashPoint"
                  expression="
                  call(* java.net.DatagramSocket.send(..)) "/>
          <advice name="crashHalt" type="around"
                                bind-to="crashPoint"/>
      </aspect>
   </package>
 </system>
</aspectwerkz>
```

Figura 5.4 - Trecho do XML para injeção de colapso parada

As figuras 5.3 e 5.4 apresentam, respectivamente, um trecho do código que implementa o *advice* de colapso de parada e o trecho do XML que define o aspecto e qual método deve ser instrumentado. Esse método é definido dentro do atributo *pointcut* do XML.

5.6.4 Falhas de Omissão

Para a implementação das falhas de omissão, foi utilizado um *around advice*. Um a*round advice* é executado quando um *join point* é alcançado e tem o controle explícito da execução do *join point*, ou seja, a utilização de um *around advice* permite alterar completamente a lógica de um método, por exemplo.

Quando o método de envio de mensagens é chamado pela aplicação alvo, os parâmetros definidos no arquivo de configuração do injetor são testadas para verificar se as falhas devem ser ativadas no momento através do método `trigger()`. Caso o método retorne verdadeiro, o *advice* instrumenta o método de envio de forma que o pacote não seja enviado. Caso o método retorne falso, o fluxo de execução é retornado para o método original, através do método `proceed()`.

O *advice* `omission` apresentado na figura 5.5, apresenta a lógica que é inserida no método `send()`. A Figura 5.6 apresenta o arquivo XML que especifica o aspecto, dizendo que o *advice* `omission` deve substituir o método `send()` do protocolo UDP.

```
public Object omission(JoinPoint joinPoint) throws Throwable {
      count++;

      if(trigger(count)){
            return null;
      }else{
            return joinPoint.proceed();
      }
}
```

Figura 5.5 - Trecho de código instrumentado para omissão.

```
<aspectwerkz>
  <system id="id" >
   <package name="ficta">
    <aspect class="FaultInjector">
            <pointcut name="omissionPoint"
                  expression="
                        call(* java.net.DatagramSocket.send(..)) "/>
          <advice name="crash" type="around"
                                bind-to="omissionPoint"/>
      </aspect>
   </package>

  </system>
</aspectwerkz>
```

Figura 5.6 - Arquivo de Configuração do aspecto Omissão

É importante salientar que estão sendo apresentados trechos básicos dos *advices*. O método `trigger()` realiza o cálculo do momento de injeção das falhas e também avalia informações de *runtime*. Essas informações podem ser utilizadas para a construção de um futuro aspecto com a função de monitorar e armazenar dados relativos aos momentos de injeção de falhas.

5.6.5 Extensão do Modelo de falhas

Como citado na seção 5.4, falhas de temporização e de resposta não são consideradas no escopo desse trabalho. No entanto, existe a possibilidade de estender a ferramenta para que esses tipos de falhas sejam abrangidos, através do uso de métodos de *runtime* oferecidos pelo framework AspectWerkz. Esses métodos retornam determinados atributos dos métodos e classes da aplicação alvo que está sendo executada. A tabela 5.1 lista alguns desses métodos.

Tabela 5.1 - Atributos de *Runtime*

Mrtti.getName();	Retorna o nome do método que será instrumentado.
JoinPoint.getCaller().	Retorna classe que chamou o método que será instrumentado.
JoinPoint.getCallee().	Retorna a classe que foi chamada.
(DatagramPacket) mrtti.getParameterValues().getAddress()	No caso de um DatagramPacket retorna o endereço de destino do pacote.
(DatagramPacket) mrtti.getParameterValues().getLength();	Retorna o tamanho do pacote.
(DatagramPacket) mrtti.getParameterValues().getData();	Retorna os dados contidos no pacote.
(DatagramPacket) mrtti.getParameterValues().getPort();	Retorna a porta para o qual o pacote foi enviado.

É possível buscar dados relativos ao pacote UDP que está sendo enviado pelo `java.net.DatagramSocket`, como por exemplo endereço de destino dos pacotes. A utilização de `(DatagramPacket)mrtti.getParameterValues().getAddress()` e `(DatagramPacket)mrtti.getParameterValues().getPort()` pode auxiliar na extensão do método de omissão de mensagens, para que sejam omitidos pacotes enviados para um determinado destino e determinada porta.

A utilização de `(DatagramPacket)mrtti.getParameterValues().getData())` pode auxiliar na extensão da ferramenta para falhas de resposta, pois permite buscar o conteúdo do pacote e alterá-lo, simulando uma saída incorreta.

Além disso, é possível estender FICTA para a injeção de falhas de duplicação de mensagens e inserção de mensagens adulteradas ou falsas. Esse modelo de falhas está previsto em Schneider (SCHNEIDER, 1993) e são consideradas subconjuntos de falhas de resposta.

6 TESTES E EXPERIMENTOS

Este capítulo apresenta testes que mostram a viabilidade da abordagem utilizada para a construção de FICTA. É importante ressaltar que o trabalho não visa uma validação completa de algum sistema específico, mas sim, o uso de algumas aplicações alvo como auxílio na prova de conceitos da abordagem apresentada.

Para demonstração das funcionalidades da ferramenta foram escolhidas aplicações distribuídas desenvolvidas sobre um sistema de comunicação de grupo, que é responsável por oferecer confiabilidade as aplicações. O critério utilizado para a escolha das aplicações levou em conta a abrangência da aplicação para demonstrar a ferramenta de injeção de falhas, além de um comportamento sob falhas fosse de fácil visualização.

O capítulo apresenta rapidamente os conceitos de comunicação de grupo, o *middleware* de comunicação de grupo utilizado pelas aplicações sob teste (JGroups) e finalmente os testes realizados com FICTA.

6.1 Alvos de Teste: Sistemas de Comunicação de Grupo

Sistemas de comunicação de grupo - SCG - (CHOCKLER; KEIDAR; VITENBERG, 2001) são *middlewares* para comunicação multiponto confiável. Os membros de um grupo podem não ser apenas processos, mas também objetos ou ainda qualquer entidade que possa enviar e receber mensagens para/de um grupo (BAN, 1998). Estes sistemas oferecem propriedades como acordo, validade, integridade e terminação. Outra possibilidade é o ordenamento no recebimento de mensagens, característica necessária em sistemas que necessitam que atualizações de valores sejam realizadas de forma consistente. Além disso, o serviço de *membership* é normalmente oferecido. Este serviço é responsável por gerenciar os membros de cada grupo a cada instante, que pode mudar sempre que um membro se une ou abandona um grupo (voluntariamente ou em caso de falha). Para representar a cada momento os membros de um grupo é usado o conceito de visão. Os membros de um grupo são comunicados da adesão ou do abandono de um membro através da instalação de uma nova visão.

Tais sistemas tornaram-se blocos básicos na construção de aplicações distribuídas com requisitos de confiabilidade e disponibilidade, pois suas características e os serviços oferecidos retiram do desenvolvedor a responsabilidade de escrever o código necessário para tolerar falha no sistema. Como as responsabilidades de detectar e

remover erros são delegados ao *middleware*, torna-se essencial a validação do mesmo, para que seja assegurado que sua especificação seja satisfeita. Desprezar os testes dos mecanismos de detecção e correção de erros pode comprometer irremediavelmente a qualidade do sistema na sua fase operacional.

Um exemplo de SCG é o JGroups (BAN, 1998), usado como um bloco de construção de aplicações Java distribuídas de alta disponibilidade. O JGroups utiliza UDP por padrão na base de sua pilha de protocolos. Os protocolos que executam no topo da pilha são responsáveis pela operação segura. Uma ferramenta de injeção de falhas baseada em UDP é essencial para a validação experimental dessas camadas superiores.

6.2 Testes com JGroups

Os testes utilizados para demonstrar o funcionamento de FICTA são simples, porém deve ser lembrado que o objetivo principal é mostrar como a abordagem baseada em POA e a ferramenta pode ser utilizada, e não os mecanismos de tolerância a falhas do protocolo alvo. A escolha de testes simples pode demonstrar como eles devem se comportar na presença de falhas. É importante a escolha de um vetor de testes com resultados conhecidos para que possam avaliar se os resultados obtidos na utilização do protótipo são consistentes com falhas reais.

Os casos de teste utilizados fazem parte das aplicações disponibilizadas como demonstrações no pacote do JGroups.

6.2.1 Ambiente de testes

Para que seja possível realizar os experimentos com FICTA, é necessário passar pela etapa de configuração do ambiente.

A primeira etapa é a instalação do ambiente Java (SUN, 2005), configuração de PATH e variáveis de ambiente, de acordo com o recomendado pela SUN Microsystems. Caso os experimentos estejam sendo realizados em um ambiente já configurado para o uso da linguagem Java, esse passo pode ser ignorado.

A segunda etapa é a instalação do *framework* AspectWerkz (ASPECTWERKZ, 2005). O *download* pode ser feito em http://aspectwerkz.codehaus.org/index-merge.html e no próprio pacote existem informações de configuração. Basicamente é realizado procedimentos semelhantes a configuração do ambiente Java (variáveis de ambiente e PATH).

Caso se queira repetir os experimentos apresentados nesse trabalho, é preciso também realizar o *download* do *toolkit* JGroups, em http://www.jgroups.org e configurar de acordo com as especificações.

A ferramenta FICTA encontra-se disponível para *download* em http://www.inf.ufrgs.br/~kohl/ficta. Juntamente com a ferramenta, encontra-se um arquivo texto com instruções de instalação e um roteiro para repetir os experimentos apresentados nesse trabalho.

6.2.2 Aplicação java.org.jgroups.Draw2Channel

A aplicação alvo consiste de um quadro de anotações distribuído, onde cada vista do quadro reside em um diferente nó do sistema. Cada usuário pode fazer anotações na sua vista, que são distribuídas para todos os demais nós ponto a ponto. Se a aplicação não for configurada para o uso de nenhum recurso de detecção e retransmissão de mensagens, uma mensagem omitida na origem representa um ponto em claro em todas as demais vistas, enquanto uma mensagem omitida no destino representa um ponto em claro em apenas em uma vista. Um nó em colapso provoca a exclusão de participante no sistema, ou seja, provoca uma alteração na visão de grupo.

O quadro de anotações distribuído que foi utilizado para esses testes utiliza dois canais de comunicação: um para controle de mudanças de visão, chamado de canal de controle, e outro para as informações relativas as anotações realizadas em cada vista, chamado canal de dados. O canal de controle utiliza, por padrão, uma pilha de protocolos com características de detecção e retransmissão de mensagens, para que seja possível manter um controle do número de membros da visão, porém pode ser configurado sem os protocolos responsáveis por essa confiabilidade. Já o canal de dados pode ser configurado para que possua mais ou menos confiabilidade. Para demonstrar a ferramenta FICTA, utilizamos para o canal de dados, apenas o protocolo UDP, para que fosse possível visualizar a omissão dos pacotes, no caso dos testes de omissão.

É apresentado como a aplicação se comporta em três situações: omissão no envio de mensagens, omissão na recepção de mensagens e colapso de uma vista. Uma quarta situação é apresentada para ilustrar, no caso especifico do quadro de anotações, o que a omissão de uma mensagem de limpeza da tela de anotações pode causar de inconsistência nas diferentes vistas.

6.2.2.1 Primeira situação: Omissão de envio de mensagens

Para o primeiro experimento, definiu-se na pilha do canal de comunicação da aplicação alvo, apenas o protocolo UDP, ou seja, nenhum mecanismo de tolerância a falhas. FICTA foi configurada para uma taxa de omissão de mensagens de 50%. A figura 6.1 ilustra a ferramenta agindo como esperado. A janela da esquerda corresponde ao processo onde as falhas estão sendo injetadas e onde o desenho está sendo construído. A omissão de mensagens é realizada no método `send` de `java.net.DatagramSocket`, impossibilitando o envio da mensagem para a rede, resultando na omissão da mensagem. As janelas do meio e da direita possuem um mesmo desenho incompleto, uma vez que as mensagens são omitidas na origem e não são recebidas.

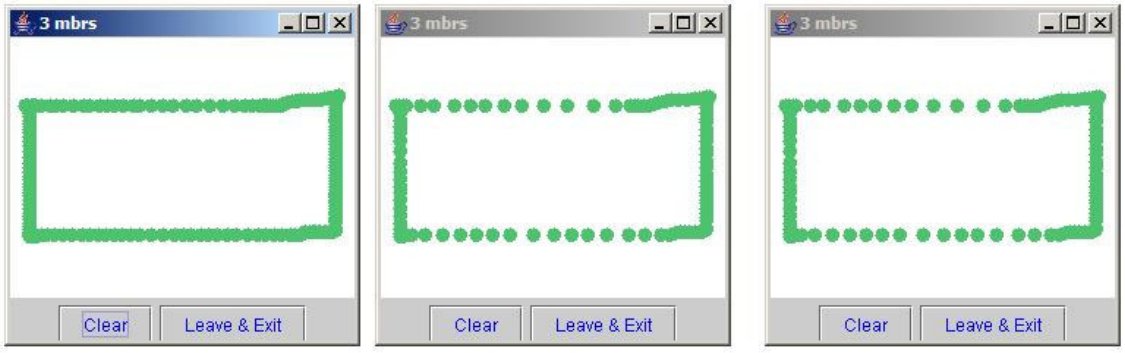

Figura 6.1 - Omissão de envio de mensagens.

6.2.2.2 Segunda Situação: Omissão da mensagem de limpeza de tela (envio ou recepção)

A terceira situação ilustra a inconsistência causada quando a mensagem omitida pela aplicação é uma mensagem de limpeza da tela de anotações. Para esse teste, FICTA também foi configurada para uma taxa de omissão de mensagens de 50%. A figura 6.2 mostra o estado inicial das vistas. Como no primeiro caso, a janela da esquerda é onde as falhas são injetadas, e as seguintes não recebem as mensagens, pois a omissão é realizada na origem.

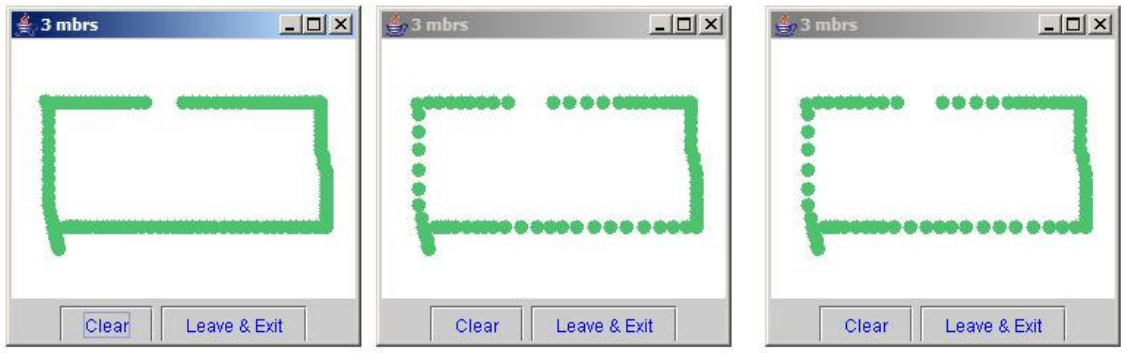

Figura 6.2 - Estado inicial de omissão de mensagem de limpeza de tela

Após um determinado período, o usuário que está utilizando a vista que está sob efeito do injetor de falhas clica no botão de limpar. Esse comando deveria ser enviado as outras vistas também, no entanto, a mensagem é omitida pelo injetor de falhas. A figura 6.3 apresenta o estado das vistas após a omissão da mensagem.

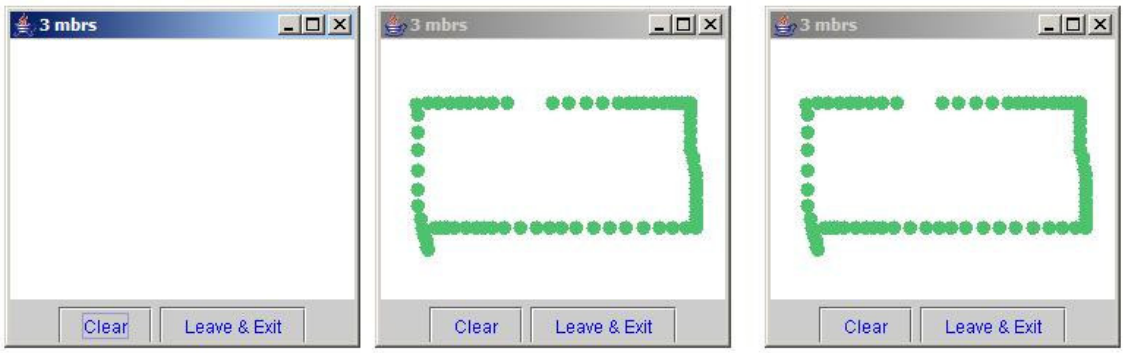

Figura 6.3 - Omissão da mensagem de limpeza da tela

Após a omissão da mensagem de limpeza, a vista que originou a mensagem está limpa, no entanto as demais continuam com o conteúdo enviado anteriormente. O estado das vistas se torna inconsistente.

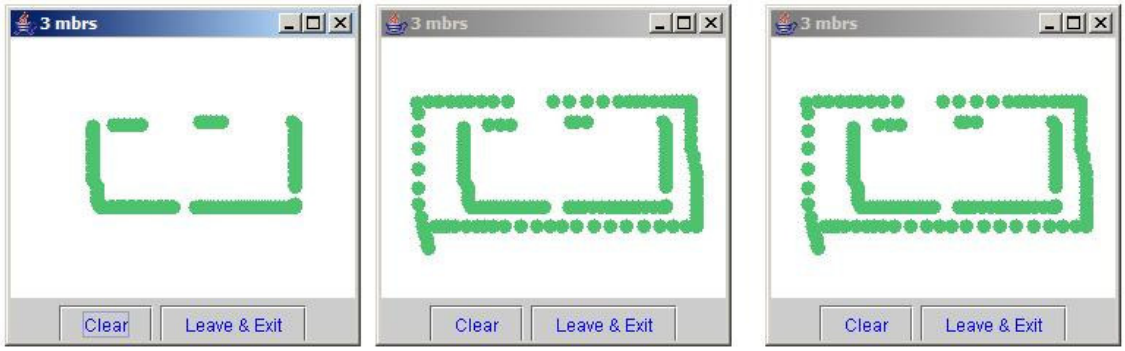

Figura 6.4 - Vistas após a omissão de limpeza da tela

A Figura 6.4 mostra o conteúdo das vistas, após a vista que omitiu a mensagem de limpeza enviar novas anotações para as outras janelas. Nesse caso, toda e qualquer anotação realizada irá se misturar a informação enviada anteriormente a limpeza de tela.

Os mesmos testes foram realizados com os mecanismos de retransmissão e detecção habilitados. Quando um pacote UDP é perdido, é detectado pelos protocolos responsáveis e o pacote é retransmitido. Os resultados desse teste mostram que mesmo com falhas sendo injetadas, os desenhos em todas as janelas são idênticos. Esse comportamento é esperado, pois as falhas estão sendo mascarados pelos mecanismos de tolerância a falhas.

O quadro de anotações distribuído, utilizando apenas UDP no seu canal de transmissão de dados, assemelha-se a *softwares* de vídeo-conferência, por exemplo. No caso de vídeo-conferências, o custo com retransmissão de pacotes é alto e pode gerar atrasos desnecessários na transmissão. Nesses casos, não retransmitir algum pacotes de *streaming* de vídeo, por exemplo, causa uma perda de informação mínima se comparada com o custo de retransmissão de pacotes perdidos.

6.2.3 Aplicação java.org.jgroups.ViewDemo

Essa aplicação não possui uma interface gráfica, porém mostra no console as mudanças de visões. O comportamento esperado da aplicação é que todos os membros sejam devidamente avisados da entrada ou saída de novos membros mantendo as visões sempre corretas. Nessa aplicação, sempre o último membro que entra na visão se torna o coordenador do grupo.

Essa aplicação não apresentou o comportamento exposto em sua especificação ao serem inseridas falhas de colapso pelo injetor de falhas. Os resultados obtidos com esse caso de testes fortalecem os principais motivos que levam a utilização de uma ferramenta de injeção de falhas para a validação de um *software* que oferece mecanismos de tolerância a falhas.

A figura 6.5 mostra o estado inicial dos três processos que foram inicializados. O primeiro processo recebe três mensagens de alteração de visão, o segundo duas

mensagens e o terceiro apenas uma, durante a inicialização do grupo. Ao final da inicialização dos três processos, todos possuem a mesma visão, contendo os endereços dos três processos participantes.

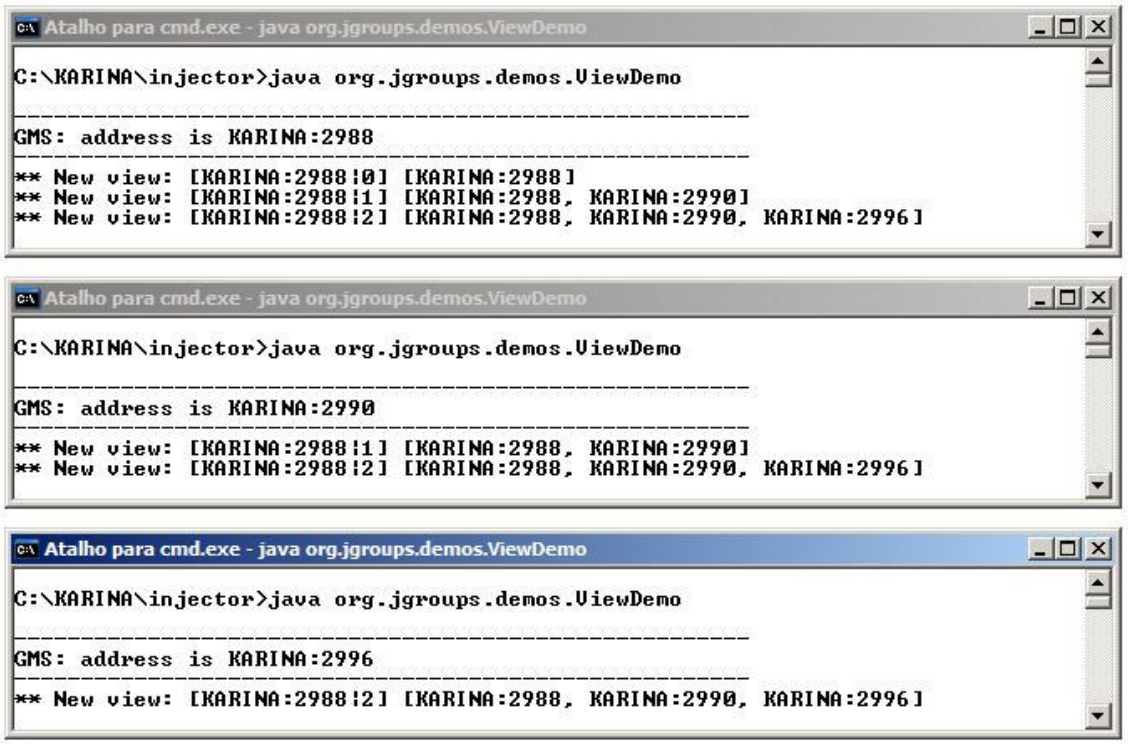

Figura 6.5 - Três processos com visões consistentes.

A figura 6.6 apresenta a alteração de visão nos processos, quando o segundo processo é morto, sem o uso da ferramenta de injeção de falhas, apenas pelo cancelamento da execução do processo. O resultado é a alteração correta de visões, mantendo a consistência, como previsto na especificação.

A figura 6.7 apresenta o resultado obtido ao utilizar FICTA para inserir falha de colapso no último processo que entra no grupo. Como citado no capítulo que descreve FICTA, a falha de colapso é simulada utilizando o método `halt` da classe `Runtime`, da API padrão da linguagem Java. O método `halt` utilizado com qualquer parâmetro inteiro diferente de zero termina forçadamente a execução da JVM que está executando, emulando então colapso do membro. O retorno desse método nunca é normal. Através disso, é possível simular a saída de um membro da visão de forma inesperada e forçada.

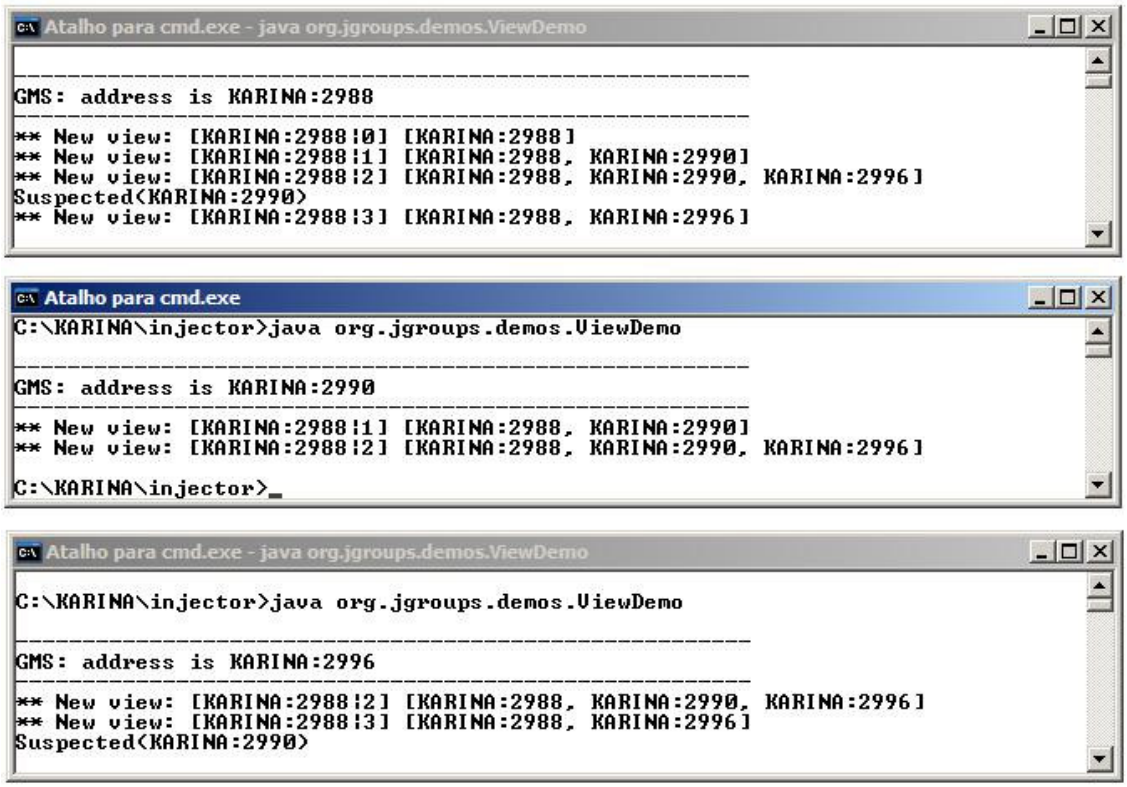

Figura 6.6 - Mudança de visão com consistência mantida.

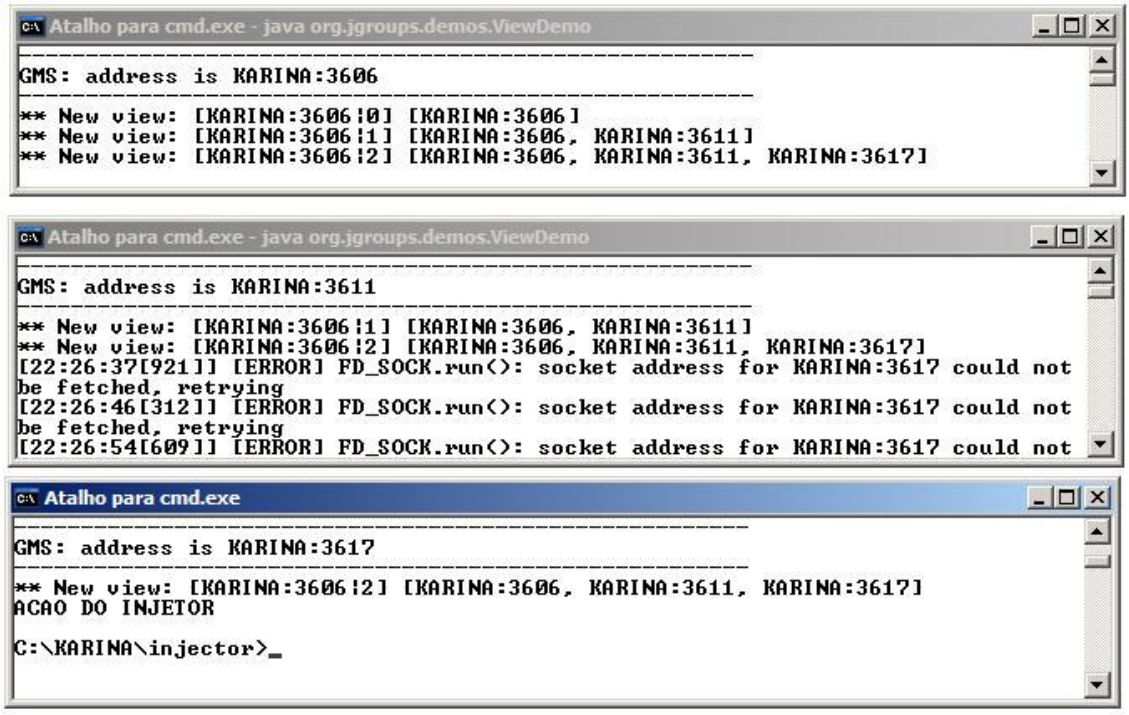

Figura 6.7 - Falha de colapso em um membro da visão. Visões inconsistentes.

Assim como apresentado anteriormente, o primeiro processo recebe três alterações de visão (a sua entrada, e as entradas dos outros dois processos). O segundo processo recebe duas alterações de visão e o terceiro processo recebe apenas uma alteração de visão (correspondente a sua entrada no grupo), durante a inicialização do

grupo. Assim que o último membro entra no grupo, o injetor de falhas, executando no mesmo processo simula a falha de colapso. Nesse caso, o esperado de acordo com a especificação da aplicação e dos protocolos utilizados pela mesma, é que os outros processos do grupo, recebam uma mensagem de alteração de visão, porém, isso não acontece. Como pode ser observado na figura 6.7, quando o terceiro processo (endereço `KARINA:3617`) sofre colapso, o processo de endereço `KARINA: 3611` não se recupera, ficando sem receber nenhuma mensagem de alteração de visão. Além disso, o processo tenta enviar mensagens para o processo que falhou e fica apenas sinalizando o erro de que o `socket` não pode ser encontrado. O processo de endereço `KARINA:3606` fica num estado mais inconsistente ainda, pois nem mensagens de erro, nem de alteração de visão são recebidas.

6.3 Avaliação de Resultados

FICTA tem como estratégia de desenvolvimento, evitar a intrusividade espacial no código fonte das aplicações alvo. Essa característica é importante, pois permite avaliar um amplo intervalo de aplicações, inclusive àquelas que não disponibilizam seus códigos fontes. Para aplicações que disponibilizam seus códigos, não existe a necessidade de varrer todas as classes a procura dos métodos que devem ser alterados e necessitando recompilação de códigos. As alterações são feitas em tempo de execução e afetam apenas os *bytecodes* da aplicação que está executando, através de um carregador de classes de sistema modificado. Esses *bytecodes* são alterados apenas quando a aplicação está rodando juntamente com o injetor, ou seja, no momento da validação.

A seção 6.3.1 mostra alguns resultados de *benchmarks* realizados pela equipe de AspectWerkz, relativos ao aumento de tempo de execução inserido pela utilização de POA. No caso de FICTA, esse aumento de tempo de execução caracteriza intrusividade temporal na aplicação alvo. Essa intrusividade seria crítica caso o trabalho tivesse sistemas de tempo real como aplicações alvo de validação. Como essas aplicações fogem ao escopo das aplicações consideradas neste trabalho, os resultados apresentados dos *benchmarks* servem de ilustração e também como estímulo para a pesquisa de como POA poderia ser usada em injeção de falhas para sistemas com restrições temporais.

A alteração do carregador de classes do sistema e modificação de *bytecodes* em tempo de execução geram problemas relativos a intrusividade temporal. O primeiro problema é relacionado com a inicialização do contêiner de aspectos, que carrega as bibliotecas necessárias para a execução dos mesmos e também realiza a modificação do carregador de classes de sistema, trazendo atraso na inicialização da aplicação que está rodando com o injetor de falhas.

A modificação de *bytecodes* em tempo de execução também é fonte geradora de atrasos na execução, pois redireciona os métodos que sofrem a injeção de falhas para que o contêiner de aspectos possa instrumentá-los.

A tabela 6.1 compara resultados de um *benchmark* realizado pela equipe do AspectWerkz e que mede em nano segundos o tempo de execução de métodos que são alterados por *advices*. Também é realizada uma comparação com AspectJ (VASSEUR, 2005).

A métrica utilizada mostra os resultados em "nano segundos por invocação de método que sofreu modificação por *advice*". De acordo com a equipe do AspectWerkz (VASSEUR,2005) uma invocação de método normal, ou seja, sem *advices*, leva cerca de 5 ns por iteração no *software* que foi utilizado para a realização do *benchmark*. Os resultados obtidos foram a média de dois milhões de iterações e representam os atrasos inseridos pelos dois tipos de *advices* utilizados por FICTA, os *before advices* e os *around advices*.

Tabela 6.1 - Atraso Inserido pelo AspectWerkz

	AspectWerkz (ns/invocação)	**AspecJ (ns/invocação)**
before	15	15
before, com acesso a informaçãos de *runtime*	50	50
around	60	10
around, com acesso a informaçãos de *runtime*	70	50

É importante frisar que o aumento no tempo de execução inserido pelos *frameworks* deve-se a necessidade de acesso a informações contextuais da execução da aplicação, como valores de parâmetros e o método alvo que está sendo modificado.

É possível observar nos valores da tabela 6.1 que as diferenças entre AspectWerkz e AspectJ se mostram mais visíveis nos *advices* do tipo *around* (com ou sem acesso a informações contextuais). É importante salientar que para esses testes, a versão de AspectJ utilizada não permitia *weaving* em tempo de execução, isto é, os aspectos tinham que ser compilados juntamente com a aplicação que dá suporte ao *benchmark*. É uma explicação para os tempos reduzidos de AspectJ em *around advices*.

7 CONCLUSÕES

Este capítulo tem como objetivo finalizar o trabalho com um resumo do que foi apresentado, resultados alcançados, dificuldades encontradas e possibilidades de trabalhos futuros.

7.1 Resultados

Este trabalho teve como motivação básica a falta de confiabilidade natural aos sistemas baseados em rede e a necessidade de validação dos mecanismos de tolerância a falhas de aplicações desenvolvidas para esses ambientes.

O objetivo principal era o desenvolvimento de uma estratégia para o desenvolvimento de injetores de falhas baseados em software, onde se alcançasse maior modularidade, reusabilidade e flexibilidade. Várias estratégias e injetores vêm sendo desenvolvidos, porém, para plataformas específicas, tornando seu uso restrito. Uma abordagem mais flexível amplia a utilização da injeção de falhas como técnica de validação de aplicações com características de dependabilidade. Assim, um maior número de aplicações pode ser validado, tornando seu uso mais seguro.

O primeiro resultado desse trabalho foi a apresentação de uma nova estratégia baseada em Programação Orientada a Aspectos - POA - para a construção de ferramentas de injeção de falhas com o objetivo de validar de sistemas distribuídos que possuam mecanismos de tolerância a falhas.

A POA possui características como flexibilidade, modularidade e facilidade de manutenção e extensão. Quanto à flexibilidade, aspectos facilitam a introdução de funcionalidades de forma não pervasiva, permitindo que a arquitetura desenvolva-se sem intrusividade sobre o código. Quanto a modularidade, aspectos permitem o projeto de componentes modulares de forma a evitar as desvantagens de espalhamento e emaranhamento de código. Finalmente, quanto à facilidade de manutenção e extensão, aspectos tornam mais fácil de manter e posteriormente estender suas características, pois a lógica está centralizada ao invés de espalhada pelo código.

Dentre as características citadas acima, a que deu origem ao estudo do uso de POA para injeção de falhas foi a de flexibilidade, pois foi identificada a possibilidade de realizar a instrumentação dinâmica do código da aplicação a ser validada. Isto é, a

inserção do código de injeção de falhas poderia ser feita em tempo de execução sem a necessidade de alteração direta do código fonte, evitando a intrusividade do injetor de falhas no código do sistema alvo.

O principal objetivo de POA é manter código relacionado em um único ponto. Um aspecto é uma unidade modular de implementação cruzada e encapsula os comportamentos que afetam diversas classes em módulos reusáveis. A injeção de falhas possui um comportamento que abrange os diversos módulos da aplicação alvo, afetando métodos que são executados em diversas classes em diversos pontos da aplicação. Desta forma, a injeção de falhas pode ser encapsulada sob a forma de aspectos.

Além dos benefícios que as características citadas trazem, foi realizado um mapeamento entre as principais construções da injeção de falhas com as principais construções de POA. Esse mapeamento ilustra a relação muito forte entre os conceitos de ambos, o que reforça a validade da abordagem.

Para a validação prática da abordagem apresentada, foi desenvolvida a ferramenta de injeção de falhas FICTA (*Fault Injection Communication Tool based on Aspects)* que tem como objetivo a validação de aplicações Java distribuídas construídas sobre o protocolo UDP e que implementam mecanismos de tolerância a falhas em protocolos de camadas superiores.

FICTA foi desenvolvida em Java, utilizando o *framework* AspectWerkz como apoio para a construção dos aspectos. A integração das classes do sistema alvo com os aspectos é realizada pelo *weaver* do AspectWerkz, que insere os ganchos para que os aspectos possam alterar os métodos corretos. Além disso, o *framework* modifica o carregador de classes de sistema da JVM para possibilitar que os aspectos alterem os *bytecodes* do sistema alvo para que os aspectos possam inserir o código instrumentado com a injeção de falhas, em tempo de execução. A configuração da ferramenta é realizada por um arquivo texto simples, onde os parâmetros são buscados no momento de carga das classes e aspectos.

O modelo de falhas que a arquitetura de FICTA leva em consideração compreende as falhas de colapso e omissão, definidas por Cristian (CRISTIAN, 1991), as mais comumente encontradas em ambientes de rede. FICTA implementa injeção de falhas de comunicação através da instrumentação da classe `java.net.DatagramSocket` da API padrão da linguagem Java. A instrumentação de uma classe base da linguagem garante a reusabilidade para toda e qualquer aplicação, protocolo ou *middleware*, escrito em Java e baseada no protocolo UDP. Portanto, qualquer aplicação Java construída sobre essa classe pode ser validada utilizando a ferramenta FICTA. Além disso, FICTA também permite a validação de aplicações *multicast* que utilizem a classe `java.net.MulticastSocket`. Essa classe estende `DatagramSocket`, sem reimplementar as primitivas de envio e recepção de mensagens (base da implementação da ferramenta), dessa forma a injeção ocorre de forma transparente, sem a necessidade de modificações na ferramenta.

O último capítulo do trabalho apresenta alguns testes realizados com FICTA, utilizando o *toolkit* para comunicação de grupo, JGroups. O objetivo não era validar totalmente o sistema, mas sim mostrar a ferramenta em ação. Foram escolhidos casos de testes simples, mas que possibilitassem uma fácil visualização da ação da ferramenta.

Durante os testes, foi descoberta uma falha no protocolo de membership do JGroups. Ao simular o colapso de um dos membros, todos os outros deveriam receber uma nova visão, que exclui o membro em falha, o que não aconteceu durante os testes. Ao "matar" qualquer um dos processos de forma *gracefull* o mecanismo de *membership* identificou a falha e enviou uma mensagem de nova visão aos membros restantes. No entanto, quando o injetor de falhas matou o último processo que entrou no grupo (usando o método `halt` da classe `Runtime`), no momento imediatamente posterior a sua entrada e a mensagem de nova visão ter sido enviada, o mecanismo de *membership* não identificou o membro em falha e os outros membros não receberam uma nova visão.

Ao longo do desenvolvimento do trabalho e através dos resultados obtidos nos testes da ferramenta, foram identificados três cenários distintos para o uso de FICTA na validação de aplicações de rede suportadas por UDP. No primeiro cenário, FICTA é uma ferramenta de teste para o desenvolvedor da aplicação de rede de alta dependabilidade. Sem alterar seu código e sem a preocupação de posterior exclusão de código extra de teste, o desenvolvedor testa sua aplicação para diferentes cenários de falhas, possibilitando o refinamento da mesma caso a cobertura de falhas não seja satisfatória. Para esse cenário, FICTA está pronta para uso imediato. Em um segundo cenário o integrador de sistemas ou o responsável pela aquisição de software precisa avaliar se uma determinada aplicação de rede ou *middleware* para sistemas distribuídos, semelhante ao JGroups por exemplo, apresentada a cobertura de falhas desejada, ou seja, cumpre com sua especificação. O fornecedor do software não tem qualquer interesse em abrir seu código na fase de avaliação. Neste cenário, FICTA apresenta a vantagem de não necessitar do código fonte para instrumentar a aplicação e pode também ser imediatamente aplicada no estado em que se encontra. Em um terceiro cenário deseja-se medir a confiabilidade, disponibilidade, latência de erro, queda de desempenho sob falhas ou qualquer outra métrica de dependabilidade. Neste último cenário, FICTA pode compor um ambiente mais completo onde monitores, coletores e analisadores de dados estariam disponíveis para cálculo e análise das métricas.

FICTA foi desenvolvida no âmbito do grupo de Tolerância a Falhas da UFRGS, em paralelo com as ferramentas FIONA e Jaca, utilizando estratégias de implementação diferentes, porém, com todas visando evitar a intrusividade espacial nas aplicações alvo. FICTA foi desenvolvida em um nível intermediário entre FIONA e Jaca. FIONA utiliza a API JVMTI de Java 5, a qual necessita de programação utilizando interface nativa (linguagem C), enquanto Jaca utiliza reflexão computacional, tratando diretamente no meta nível. O uso de POA permitiu que FICTA fosse desenvolvida no mesmo nível que a programação Java tradicional.

Por fim, o trabalho apresentado, validado pelo desenvolvimento da ferramenta FICTA, mostrou que o uso de POA para a criação de injetores de falhas é uma abordagem factível e eficiente, resultando em uma solução modular, reusável e sem intrusividade espacial nas aplicações alvo. Não foi encontrado nas pesquisas, nenhum trabalho que utilizasse POA especificamente para injeção de falhas, no entanto a orientação a aspectos tem sido largamente usada para instrumentação de código e testes funcionais, devendo se consolidar como mecanismo fundamental. Além disso, o trabalho faz uso de novas tecnologias para resolver um problema clássico, tornando a abordagem apresentada original.

7.2 Dificuldades Encontradas

A maior dificuldade encontrada foi definir qual seria a ferramenta que daria suporte a Programação Orientada a Aspectos para o desenvolvimento de FICTA. Por ser um paradigma relativamente novo, que vem ganhando força e crescendo, durante o desenvolvimento de FICTA, muitas mudanças em ferramentas estudadas ocorreram, novas implementações, etc.

Um primeiro protótipo de FICTA foi desenvolvido utilizando AspectJ. No entanto, no período de desenvolvimento, AspectJ não oferecia suporte para *weaving online*. Dessa forma, os códigos fontes das classes do sistema alvo deveriam estar disponíveis para serem compilados juntamente com os aspectos. Essa característica fugia aos objetivos traçados para o desenvolvimento de FICTA, inclusive inviabilizando a instrumentação da classe `java.net.DatagramSocket`, tornando a ferramenta menos portável.

A partir desse problema, optou-se pelo uso do *framework* AspectWerkz, o qual permite que classes e aspectos sofram *weaving* em tempo de carga e execução, a partir da modificação do carregador de classes do sistema. Além disso, a utilização do modelo de definição em XML oferecido pelo AspectWerkz, flexibiliza a definição dos *pointcuts*, tornando mais fácil realizar alterações caso algum novo método deva ser instrumentado pelos mesmos *advices* presentes no aspecto de injeção de falhas, sem necessidade de recompilações.

Nos momentos finais de desenvolvimento da ferramenta e da escrita deste texto, os projetos AspectJ e AspectWerkz concordaram em trabalhar juntos para produzir uma única plataforma de Programação Orientada a Aspectos complementando suas forças e experiências. Dessa união surgiu o AspectJ 5, que oferece os mecanismos de *weaving* que seriam necessários no início desse trabalho. Como o lançamento de AspectJ 5 é muito recente, não ouve possibilidade de portar a ferramenta para o mesmo.

7.3 Trabalhos Futuros

Existem alguns possíveis trabalhos a serem realizados para tornar FICTA uma ferramenta mais completa. Tais como:

- A criação de uma interface com o usuário que permita a configuração do arquivo de configurações, facilitando a criação e modificação dos cenários de falhas;

- A utilização de FICTA com aplicações auxiliares: monitores, coletores e analisadores de dados. O objetivo seria a criação um ambiente completo de injeção de falhas, permitindo além da realização dos experimentos de injeção de falhas, a análise dos resultados dos mesmos;

- Expansão do modelo de falhas para falhas de temporização, falhas de resposta e falhas de duplicação de mensagens;

- Portar a ferramenta para AspectJ 5.

REFERÊNCIAS

ARLAT, J. et al. Fault Injection for dependability Validation: a methodology and some applications. **IEEE Transactions on Software Engineering**, New York, v. SE-16, n.2, pp. 166-182, Feb. 1990.

ANNOTATIONS. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html#annotations>. Acesso em: 15 maio 2005.

ASPECTJ+ASPECTWERKZ. Disponível em <http://aspectwerkz.codehaus.org/index-merge.html>. Acesso em: 15 maio 2005.

ASPECTWERKZ. Disponível em: <http://aspectwerkz.codehaus.org>. Acesso em: 15 maio 2005.

BAN, B**. JavaGroups – Group Communication Patterns in Java**. [S.l.]: Department of Computer Science, Cornell University, July 1998.

BARCELOS, P.P.A.; DREBES, R.J. ; JACQUES-SILVA G.; WEBER, T.S**.** A toolkit to test the intrusion of fault injection methods. In: LATIN AMERICAN TEST WORKSHOP: IEEE LATW, 5., 2004, Cartagena. **Proceedings…** Cartagena: [s.n.], 2004. p. 152-157.

BARCELOS, P.P.A.; WEBER, T. **INFIMO:** Um Toolkit para Experimentos de Intrusão de Injetores de Falhas. 2001. 161 f**.** Tese (Doutorado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

BARCELOS, P.P.A.; LEITE, F.O.; WEBER, T.S**.** Building a Fault Injector to Validate Fault Tolerant Communication Protocols. In: INTERNATIONAL CONFERENCE ON PARALLEL COMPUTING SYSTEMS, IPCS, 1999, Ensenada. **Proceedings…** Enseada: [s.n.], 1999.

BONÉR, J. **AspectWerkz – dynamic AOP for Java**. Documento eletrônico. Disponível em: <http://http://codehaus.org/~jboner/papers/aosd2994_aspectwerkz.pdf>. Acesso em: 15 março 2005.

BONÉR, J.; VASSEUR, A. AspectWerkz for Dynamic Aspect-Oriented Programming. In: INTERNATIONAL CONFERENCE ON ASPECT ORIENTED SOFTWARE

DEVELOPMENT: AOSD'04, 3., 2004, Lancaster. **Proceedings…** Lancaster: [s.n.], 2004.

BURKE, B. **When to use Annotations.** Documento Eletrônico. Disponível em: <http://jboss.org/jbossBlog/blog/bburke/?permalink=When_to_use_annotations.txt>. Acesso em: 15 maio 2005.

CARREIRA, J.; SILVA, J.G. Why do some (weird) People Inject Faults? **Software Engineering Notes**, New York, v.23, n. 1, p. 42-43, 1998.

CARREIRA, J.; MADEIRA, H. Assessing the Effects of Communication Faults on Parallel Applications. In: INTERNATIONAL COMPUTER PERFORMANCE AND DEPENDABILITY SYMPOSIUM, 1995. **Proceedings…** Erlangen: Springer-Verlag, IEEE, 1995.

CHIBA, S. **Javassist:** Java bytecode manipulation made simple. 2004. Disponível em: <http://www.jboss.org/developers/projects/javassist.html>. Acesso em: 25 abr. 2005.

CHOCKLER, G.; KEIDAR, I.; VITENBERG, R**.** Group Communication Specifications: a comprehensive study. **ACM Computing Surveys**, New York, v. 33, n. 4, Dec. 2001.

CRISTIAN, F. Understanding Fault-Tolerant Distributes Systems. **Communication of the ACM**, New York, v.34, n.2, Feb. 1991.

CUNHA, J.C.; RELA, M.Z.; SILVA, J.G. **RT:Xception:** Fault Injection for Real Time. Coimbra: Centro de Informática e de Sistemas da Universidade de Coimbra, 2000 (TR-2000/002).

DAHM, M. **Byte Code Engineering Library**. 2002. Disponível em: <http://bcel.sourceforge.net/> Acesso em: 25 abr. 2005.

DAWSON, S.; JAHANIAN, F.; MITTON, T. ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementation. In: INTERNATIONAL COMPUTER PERFORMANCE AND DEPENDABILITY SYMPOSIUM, IPDS, 1996, cidade **Proceedings..**. [S.l.: s.n.], 1996.

DETERS, M.; CYTRON, R. K**.** Introduction of Program Instrumentation using Aspects. In: WORKSHOP ON ADVANCED SEPARATION OF CONCERNS IN OBJECT-ORIENTED SYSTEMS, OOPSLA, 2001, Tampa. **Proceedings…** Tampa: ACM, 2001.

DREBES, R.J.; LEITE, F.O.; SILVA, G.J.; MOBUS, F.; WEBER, T.S. ComFIRM: a Communication Fault Injector for Protocol Testing and Validation**.** In: IEEE LATIN AMERICAN TEST WORKSHOP, LATW, 2005, Salvador**. Proceedings...** [S.l.:s.n.], 2005**.**

DUZAN, G. et al. Building Adaptive Distributed Applications with Middleware and Aspects. In: INTERNATIONAL CONFERENCE ON ASPECT ORIENTED SOFTWARE DEVELOPMENT, AOSD, 2004, Lancaster. **Proceedings…** Lancaster: [s.n.], 2004.

ELRAD, T.; AKSIT, M.; KICKZALES, G.; LIEBERHERR, K.; OSSHER, K. Discussing Aspects of AOP. **Communication of the ACM**, New York, v. 44, n. 10, Oct. 2001.

FICTA. Disponível em: < http://www.inf.ufrgs.br/~kohl/ficta >. Acesso em: 25 junho 2005.

FILMAN, R.; LEE, D. Redirecting by Injector. In: IEEE INTL. CONF. ON DISTRIBUTED COMPUTING SYSTEMS, 21., 2001. **Proceedings…** [S.l.:s.n.], 2001. p.141-146.

GAL, A.; SPINCZYK, O.; PREIKSHAT, W. S. On Aspect Orientation in Distributed Real Time Dependable Systems. In: INTERNATIONAL WORKSHOP ON OBJECT ORIENTED REAL TIME DEPENDABLE SYSTEMS, 17., 2002. **Proceedings…** [S.l.:s.n.], 2002.

GOLDENBERG, A.; HAVELUND, K**. Instrumentation of Java Bytecode for Runtime Analysis.** Documento Eletrônico. Disponível em: <http://ase.arc.nasa.gov/people/havelund/Publications/jspy-final.pdf> . Acesso em: 15 maio 2005.

HSUEH, M.; TSAI, T.; IYER, R. Fault Injection Techniques and Tools. **IEEE Computer**, Los Alamitos, v. 30, n. 4, Apr. 1997.

IYER, R.K. Experimental Evaluation. In: INTERNATIONAL FAULT-TOLERANT COMPUTING SYMPOSIUM, 25., 1995, Pasadena. **Proceedings...** Pasadena, USA: [s.n.], 1995.

JALOTE, P. **Fault Tolerance in Distributed Systems**. Englewood Cliffs: Prentice-Hall, New Jersey, 1994.

JAVA. **Java Technology**. Disponível em: <http://java.sun.com>. Acesso em: 24 abr. 2005.

JVMDI. **Java Virtual Machine Debugger Interface**. Disponível em: <http://java.sun.com/j2se/1.3/docs/guide/jpda/architecture.html#jvmdi>. Acesso em: 23 abr. 2005.

JVMTI. **Java Virtual Machine Tool Interface**.Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html>. Acesso em: 23 abr. 2005.

KANAWATTI, G.A. et al**.** FERRARI: A flexible software based fault and error injection system**. IEEE Transactions on Computers**, New York, v. 44, n. 2, p 248-260, Feb. 1995.

KAO, W.; IYER, R.K. DEFINE: a distributed fault injection and monitoring environment. In: IEEE WORKSHOP ON FAULT TOLERANT PARALLEL AND DISTRIBUTED SYSTEMS, 1994, cidade. **Proceedings**…[S.l: s.n.], 1994.

KAO, W. et al. FINE: A Fault Injection and Monitoring Environment for Tracing the Unix System Behaviors under Faults**. IEEE Transactions on Software Engineering**, New York, v.SE-19, n.11, Nov. 1993.

KARLSSON, J. et al. Using Heavy-ion radiation to validate fault handling mechanisms. **IEEE Micro,** New York, v.14, n. 1, p 8-23, Feb. 1994.

KICKZALES, G. Aspect Oriented Programming. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1997. **Proceedings…** [S.l.: s.n.], 1997.

KICKZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. An Overview of AspectJ. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, ECOOP, 15., 2001. **Proceedings…**Berlin: Springer-Verlag, 2001. p. 327-353. (Lecture Notes in Computer Science, 2072).

LADDAD, R. **I Want my AOP!, Part 1 - Separate software concerns with aspect-oriented programming.** Disponível em: <http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect_p.html. 2002> Acesso em: 15 fev. 2005.

LAPRIE, J.C. Dependable Computing and Fault Tolerance: Concepts and Terminology. In: IEEE INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING SYSTEMS, FTCS, 1985, Ann Arbor. **Proceedings…** Ann Arbor, USA: [s.n.], 1985.

LEE,K.W. **An Introduction to Aspect-Oriented Programming**. Reading Assignment. The Hong Kong University of Science and Technology. [S.l.; s.n.]. 2003.

LEITE, F.O. **ComFIRM:** Injeção de Falhas de Comunicação através da alteração de recursos do sistema operacional. 2000. 117 f. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul. Porto Alegre.

MARTINS, E.; RUBIRA, C.M.F.; LEME, N.G.M**.** Jaca: A Reflective Fault Injection Tool Based on Patterns. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 2002, Washington. **Proceedings…**Washington, EUA: IEEE Computer Society, 2002.

MIRANDA, H.; PINTO, A.; RODRIGUES, L**.** Appia, a flexible protocol kernel supporting multiple coordinated channels. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 21., 2001, Phoenix. **Proceedings...** Arizona, EUA: IEEE Computer Society, 2001.

OSSHER, H.; TARR, P**.** Using Multidimensional Separation of Concerns to (Re)shape Evolving Software**. Communications of the ACM**, New York, v. 44, n. 10, p 43-50, Oct. 2001.

PEARSON, C**. A Framework for the Aspect-Oriented Dynamic Instrumentation of Java Programs**. London: Department of Computing - Imperial College London, 2003.

PUTTRYEZ, E.; BERNARD, G**.** Using Aspect Oriented Programming to build a portable load balancing service. In: INTERNATIONAL CONFERENCE ON

DISTRIBUTED COMPUTING SYSTEMS WORKSHOPS, 22., 2002, Vienna. **Proceedings…** Vienna, Áustria: IEEE Computer Society. 2002.

ROCHA, A.D.; SIMÃO, A.S.S.; MALDONADO, J.C.; MASIERO, P.C. Teste Funcional: Uma abordagem Auxiliada por Aspectos. In: WORKSHOP BRASILEIRO DE DESENVOLVIMENTO DE SOFTWARE ORIENTADO A ASPECTOS, WASP, 2004, Brasília. **Proceedings…** Brasília, DF: [s.n.], 2004.

ROSA, A.A.; MARTINS, E. Using reflexive programming to inject faults into object oriented systems. In: IFIP INTERNATIONAL WORKSHOP ON DEPENDABLE COMPUTING AND ITS APPLICATIONS, DCIA, 1998, Johannesburg. **Proceedings…** Johannesburg, South Africa: [s.n.], 1998.

ROYCHOUDHURY, S.; GRAY, J.; WU, H.; ZHANG, J.; LIN, Y. A Comparative Analysis of Meta-programming and Aspect-Orientation**.** In: ANNUAL ACM SOUTH EAST CONFERENCE, 41., 2003, Savannah. **Proceedings…** Georgia, EUA: [s.n.], 2003.

SCHNEIDER, F.B. What good are Models and What Models are Good? In: MULLENDER, S. **Distributed Systems**. [S.l.]: Addison-Wesley, 1993.

SILVA, G. J.; MORAES, R.; WEBER, T.; MARTINS, E. Validando sistemas distribuídos desenvolvidos em Java utilizando injeção de falhas de comunicação por software**.** Em: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, WTF, 5., Gramado. **Anais…** Gramado, RS: [s.n.], 2004. p. 53-64.

SILVA, G. J.; DREBES, R.J.; GERCHMANN, J.; WEBER, T.S. FIONA: A Fault Injector for Dependability Evaluation of java-Based Network Applications. In: IEEE INTERNATIONAL SYMPOSIUM ON NETWORK COMPUTING AND APPLICATIONS, 3., 2004, Boston. **Proceedings…**. Massachusetts, EUA: IEEE Computer Society, 2004.

SILVEIRA, K.K.; WEBER, T.S. Um injetor de falhas de comunicação construído usando Programação Orientada a Aspectos. Em: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, WTF, 6., 2005 . **Anais…** Fortaleza: [s.n.], 2005.

SILVEIRA, K.K.; WEBER, T.S. Uma arquitetura de injetor de falhas orientada a aspectos para a validação de sistemas de comunicação de grupo. In: ESCOLA REGIONAL DE REDES DE COMPUTADORES, ERRC, 2., 2004. **Anais...** Canoas: [s.n.], 2004.

SUN MICROSYSTEMS. **The Java Tutorial:** All About Datagrams. Disponível em: < http://java.sun.com/docs/books/tutorial/networking/datagrams/>. Sun MicroSystems. Acesso em: 15 maio 2005.

SZENTIVANYI, D.; TEHRANI, S.N. Aspects for Improvement of Performance in Fault Tolerant Software. In: IEEE PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING, 10., 2004. Papeete. **Proceedings…** Tahiti: French Polynesia: IEEE Computer Society, 2004.

TATSUBORI, M.; CHIBA, S.; KILLIJIAN, M.O.; ITANO, K. OpenJava: A Class-based Macro System for Java. In: **Reflection in Software Engineering**. [S.l.]: Springer Verlag, 2000. (Lecture Notes in Computer Science, 1826).

TROMER, E. **Java Instrumentation Engine (JIE)**. 1999. Disponível em: <http://www.forum2.org/eran/jie/>. Acesso em: 25 abr. 2005.

TSANG, S. L.; CLARKE, S.; BANIASSAD, E. An Evaluation of Aspect Oriented Programming for Java based Real Time Systems Development. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT ORIENTED REAL TIME DISTRIBUTED COMPUTING, 17., 2004. Vienna. **Proceedings…** Vienna, Áustria: IEEE Computer Society, 2004.

VASSEUR, A. **AOP Benchmark**. Documento Eletrônico. Disponível em: <http://docs.codehaus.org/display/AW/AOP+Benchmark>. Última atualização dezembro de 2004. Acesso em: 15 maio 2005.

WIN, B.D.; VANHAUTE, B.; DECKER, B.D. Security Through Aspect-Oriented Programming. In: ANNUAL WORKING CONFERENCE ON NETWORK SECURITY , 1., 2001. Louven. **Proceedings…** Louven, Belgium: Kluwer. 2001.

XML. Extensible Markup Language. Disponível em: < http://www.w3.org/XML/ >. Acesso em: 20 junho 2005.