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

CRISTINA CIPRANDI MENEGOTTO

Injeção de Falhas de Comunicação em Aplicações Multiprotocolo

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

Profª. Drª. Taisy Silva Weber
Orientadora

Porto Alegre, novembro de 2009

Livros Grátis

http://www.livrosgratis.com.br

Milhares de livros grátis para download.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Menegotto, Cristina Ciprandi

Injeção de Falhas de Comunicação em Aplicações Multiprotocolo / Cristina Ciprandi Menegotto. – Porto Alegre: PPGC da UFRGS, 2009.

92 p.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2009. Orientadora: Taisy Silva Weber.

1. Injeção de falhas de comunicação. 2. Aplicações Java multiprotocolo. I. Weber, Taisy Silva. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Pró-Reitor de Coordenação Acadêmica: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PPGC: Prof. Álvaro Freitas Moreira
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“The strongest of all warriors are these two — Time and Patience.”

— LEO TOLSTOY

AGRADECIMENTOS

Agradeço, especialmente, à minha orientadora, Taisy Silva Weber, pelo apoio e incentivo dedicados desde que ingressei no Grupo de Tolerância a Falhas.

Agradeço a Juliano Vacaro, ex-colega do Grupo de Tolerância a Falhas, pelo auxílio quando comecei a trabalhar na área de injeção de falhas, que foi um incentivo para o ingresso no mestrado e continuidade de pesquisa na área.

Agradeço ao CNPq pela concessão da bolsa de mestrado.

Agradeço aos meus pais pelo suporte e auxílio financeiro. Ao meu irmão, Marcos Ciprandi Menegotto, e às amigas Sonia Lugo, Michele dos Santos e Mariana Pooli pelo carinho e confiança. E, por fim, agradeço àqueles que compreenderam e respeitaram o tempo e esforços dedicados à realização deste trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 8

LISTA DE FIGURAS 10

LISTA DE TABELAS 11

RESUMO 12

ABSTRACT 13

1 INTRODUÇÃO 14
1.1 Objetivos 15
1.2 Resultados Alcançados 15
1.3 Organização do Texto 16

2 INJEÇÃO DE FALHAS 18
2.1 Conceitos Básicos 18
2.1.1 Utilidades da Injeção de Falhas 19
2.1.2 Ambiente de Injeção de Falhas 20
2.1.3 Falhas de Comunicação 20
2.1.4 Intrusividade 22
2.2 Ferramentas e Frameworks de Injeção de Falhas 22
2.2.1 FIONA 23
2.2.2 FIRMAMENT 24
2.2.3 FIERCE 24
2.2.4 FIRMI 25
2.2.5 DOCTOR 25
2.2.6 NFTAPE 26
2.2.7 ORCHESTRA 26
2.2.8 Loki 27
2.2.9 FAIL-FCI 28
2.2.10 FIT 28
2.3 Injeção de Falhas em Aplicações Multiprotocolo 29
2.3.1 Aplicações Multiprotocolo 29
2.3.2 Dificuldades na Injeção de Falhas em Aplicações Java Multiprotocolo . . 30
2.3.3 Requisitos Identificados para a Solução 31
2.3.4 Potencial de Injetores para o Teste de Aplicações Java Multiprotocolo . . 32
2.4 Comparação de Comform a Trabalhos Relacionados 34
2.5 Conclusões do Capítulo 35

3 UDP, TCP E RMI EM JAVA . . . 36
3.1 UDP . . . 37
3.1.1 UDP no pacote java.net . . . 37
3.1.2 UDP no pacote java.nio . . . 38
3.2 TCP . . . 38
3.2.1 TCP no pacote java.net . . . 39
3.2.2 TPC no pacote java.nio . . . 40
3.3 RMI . . . 41
3.4 Conclusões do Capítulo . . . 43

4 QUESTÕES DE PROJETO . . . 44
4.1 Modelo Genérico . . . 44
4.1.1 Modelagem de Falhas . . . 44
4.1.2 Cuidados para Emulação de Colapso em Aplicações TCP . . . 45
4.1.3 Seleção e Manipulação de Mensagens . . . 46
4.2 Abordagens para Injeção de Falhas em Aplicações Java . . . 46
4.2.1 Suporte à Instrumentação em Ambientes Java . . . 47
4.2.2 Javassist: Biblioteca de Manipulação de Bytecode . . . 48
4.2.3 Programação Orientada a Aspectos . . . 49
4.2.4 Uso de Firewalls para Emulação de Falhas . . . 50
4.3 Conclusões do Capítulo . . . 50

5 INJETOR DE FALHAS COMFORM . . . 52
5.1 Arquitetura de Comform . . . 52
5.2 Classes Instrumentadas . . . 55
5.2.1 Instrumentação de UDP e TCP . . . 55
5.2.2 Instrumentação de RMI . . . 59
5.3 Filtro de Mensagens . . . 61
5.4 Monitoramento e Coleta de Dados . . . 63
5.5 Falhas e Cargas de Falhas . . . 63
5.5.1 Modelagem de Falhas . . . 63
5.5.2 Construção de Módulos de Carga de Falhas . . . 66
5.5.3 Carregamento de Módulos de Carga de Falhas . . . 67
5.6 Portabilidade . . . 68
5.7 Expansão de Comform para Novos Protocolos . . . 68
5.8 Protótipo de Comform . . . 69
5.9 Conclusões do Capítulo . . . 69

6 EXPERIMENTOS DE INJEÇÃO DE FALHAS . . . 71
6.1 Experimento com Aplicação RMI . . . 71
6.2 Experimento com Zorilla . . . 73
6.3 Experimento com JGroups . . . 77
6.4 Conclusões do Capítulo . . . 79

7 CONSIDERAÇÕES FINAIS . . . 80
7.1 Conclusões . . . 80
7.2 Trabalhos Futuros . . . 81

REFERÊNCIAS . . . 82

APÊNDICE A HISTÓRICO DE DESENVOLVIMENTO 89

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
BCEL	Byte Code Engineering Library
ComFIRM	COMmunication Fault Injection through Operating System Resources Modification
Comform	COMmunication Fault injector ORiented to Multi-protocol Java applications
COTS	Commercial Off the Shelf
DOCTOR	IntegrateD SOftware Fault InjeCtiOn EnviRonment
FAIL	FAult Injection Language
FCI	FAIL Cluster Implementation
FICTA	Fault Injection Communication Tool Based on Aspects
FIERCE	Fault Injector for Evaluation of Remote Communication Environments
FIONA	Fault Injector Oriented to Network Applications
FIRMAMENT	Fault Injection Relocatable Module for Advanced Manipulation and Evaluation of Network Transports
FIRMI	Fault Injector for RMI
FIRMVM	FIRMAMENT Virtual Machine
FIT	Fault Injection Technology
FTP	File Transfer Protocol
Grid-FIT	Grid - Fault Injection Technology
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JAR	Java Archive
Javassist	Java Programming Assistant
JDK	Java Development ToolKit
JVM	Java Virtual Machine
JVMTI	Java Virtual Machine Tool Interface

LWFI	LightWeight Fault Injector
NFTAPE	Network Fault Tolerance and Performance Evaluator
PFI	Protocol Fault Injection
POA	Programação Orientada a Aspectos
RFC	Request for Comments
RMI	Remote Method Invocation
RNG	Random Number Generator
SCTP	Stream Control Transmission Protocol
SDK	Standard Development Kit
SO	Sistema Operacional
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
U2TP	UML 2.0 Testing Profile
U2TP-FI	UML 2.0 Testing Profile with Fault Injection
UDP	User Datagram Protocol
UFRGS	Universidade Federal do Rio Grande do Sul
UML	Unified Modelling Language
WS-FIT	Web Service - Fault Injection Technology

LISTA DE FIGURAS

Figura 2.1: Componentes de ambiente de injeção de falhas (HSUEH; TSAI; IYER, 1997). 21
Figura 2.2: Inadequação de injeção de falha de colapso em um nodo Zorilla usando FIONA. 32

Figura 3.1: Exemplo de aplicação distribuída baseada em RMI (SUN MICROSYSTEMS, 2006a). 42

Figura 4.1: Emulação inconsistente de falhas de colapso no nível da JVM para aplicações Java baseadas em TCP. 46

Figura 5.1: Arquitetura simplificada da ferramenta. 53
Figura 5.2: Adequação da injeção de falha de colapso em um nodo Zorilla usando Comform. 54
Figura 5.3: Instrumentação de métodos `bind()`. 56
Figura 5.4: Instrumentação de métodos relacionados ao envio e recebimento de mensagens. 57
Figura 5.5: Instrumentação de métodos relacionados ao estabelecimento de conexão. 57
Figura 5.6: Modelagem de mensagens dos protocolos de interesse. 61
Figura 5.7: Diagrama simplificado de modelagem de falhas. 63

Figura 6.1: Um trecho de código do cliente. 72
Figura 6.2: *Faultload* para emulação de colapso. 72
Figura 6.3: Exceção gerada no cliente. 73
Figura 6.4: Planejamento de injeção de falha de colapso em um nodo Zorilla usando Comform. 74
Figura 6.5: Inicialização do nodo `jaguar`. 74
Figura 6.6: *Faultload* para emulação de colapso de nodo em `dkw`. 75
Figura 6.7: Exceção gerada no nodo `mercedes`. 75
Figura 6.8: Comportamento do nodo `grantorino` após inicialização. 76
Figura 6.9: *Faultload* para emulação de colapsos de *link*. 78
Figura 6.10: Injeção de colapsos de *link*. 78

LISTA DE TABELAS

Tabela 2.1: Comparando Comform a trabalhos relacionados. 34

Tabela 5.1: Métodos instrumentados em `DatagramSocket`, `ServerSocket`, `Socket`, `SocketInputStream` e `SocketOutputStream`. . . 58
Tabela 5.2: Métodos de interesse em `DatagramChannel`, `SocketChannel` e `ServerSocketChannel`. 59
Tabela 5.3: Métodos de interesse em `RemoteRef` e `ServerRef`. 60
Tabela 5.4: Formato de regras de *firewall* aplicado quando Comform é executado em um nodo e falhas de colapso de nodo e de colapso de *link* devem ser percebidas em todos os outros nodos da aplicação alvo. 62
Tabela 5.5: Formato de regras de *firewall* aplicado quando Comform é executado em um nodo e emula falhas de colapso de *link* somente no *link* entre esse e outro(s) nodo(s) determinados da aplicação alvo. 62
Tabela 5.6: Valores para o atributo tipo (`type`) de mensagens UDP, TCP e RMI. 67

Tabela 6.1: Configurações dos computadores disponíveis para experimentos. . . . 71

RESUMO

Aplicações de rede com altos requisitos de dependabilidade devem ser testadas cuidadosamente em condições de falhas de comunicação para aumentar a confiança no seu comportamento apropriado na ocorrência de falhas. Injeção de falhas de comunicação é a técnica mais adequada para o teste dos mecanismos de tolerância a falhas destas aplicações em presença de falhas de comunicação. Ela é útil tanto para auxiliar na remoção de falhas como na previsão de falhas.

Dentre as aplicações de rede, algumas são baseadas em mais de um protocolo, como UDP, TCP e RMI. Elas são denominadas multiprotocolo no contexto desse trabalho, que foca naquelas escritas em Java e baseadas em protocolos que estão acima do nível de rede na arquitetura TCP/IP. Um injetor de falhas de comunicação adequado, que trate todos os protocolos utilizados, é necessário para o seu teste. Caso a emulação de uma falha que afete a troca de mensagens não leve em consideração todos os protocolos simultaneamente utilizados, o comportamento emulado durante um experimento poderá ser diferente daquele observado na ocorrência de uma falha real, de modo que podem ser obtidos resultados inconsistentes sobre o comportamento da aplicação alvo em presença da falha.

Muitos injetores de falhas de comunicação encontrados na literatura não são capazes de testar aplicações Java multiprotocolo. Outros possuem potencial para o teste dessas aplicações, mas impõem grandes dificuldades aos engenheiros de testes. Por exemplo, contrariamente ao enfoque deste trabalho, algumas ferramentas são voltadas à injeção de falhas de comunicação para o teste de protocolos de comunicação, e não de aplicações. Tal orientação ao teste de protocolos costuma levar a grandes dificuldades no *teste de caixa branca* de aplicações. Entre outros exemplos de dificuldades proporcionadas por ferramentas encontradas na literatura estão a incapacidade de testar diretamente aplicações Java e a limitação quanto aos tipos de falhas que permitem emular. A análise de tais ferramentas motiva o desenvolvimento de uma solução voltada especificamente ao teste de aplicações multiprotocolo desenvolvidas em Java.

Este trabalho apresenta uma solução para injeção de falhas de comunicação em aplicações Java multiprotocolo. A solução opera no nível da JVM, interceptando mensagens de protocolos, e, em alguns casos, opera também no nível do sistema operacional, usando regras de *firewall* para emulação de alguns tipos de falhas que não podem ser emulados somente no nível da JVM. A abordagem é útil para testes de caixa branca e preta e possui características importantes como a preservação do código fonte da aplicação alvo. A viabilidade da solução proposta é mostrada por meio do desenvolvimento de Comform, um protótipo para injeção de falhas de comunicação em aplicações Java multiprotocolo que atualmente pode ser aplicado para testar aplicações Java baseadas em qualquer combinação dos protocolos UDP, TCP e RMI (incluindo as baseadas em um único protocolo).

Palavras-chave: Injeção de falhas de comunicação, aplicações Java multiprotocolo.

Communication Fault Injection in Multi-Protocol Applications

ABSTRACT

Networked applications with high dependability requirements must be carefully tested under communication faults to enhance confidence in their proper behavior. Communication fault injection is the most suitable technique for testing the fault tolerance mechanisms of these applications under communication faults. The technique is useful both for helping in fault removal and for fault forecasting.

Some networked applications are based on more than one protocol, such as UDP, TCP and RMI. They are called multi-protocol in the context of this work, that targets on those written in Java and based on protocols above Internet layer in TCP/IP architecture. A suitable communication fault injector, that properly handles all used protocols, is required for their test. If the emulation of a fault that affects message exchanging does not take into account all simultaneously used protocols, the behavior emulated in an experiment can be different from that observed under real fault occurrence. Therefore, inconsistent results about the target application's behavior under faults can be obtained from the experiments.

Many communication fault injectors found in literature are not capable of testing multi-protocol Java applications. Others can potentially test these applications, but they impose several drawbacks for test engineers. For instance, contrarily to the focus of this work, some tools aim to communication protocol testing and not to application testing. This orientation to protocol testing usually leads to great difficulties in white box testing of applications. Other examples of difficulties related to using tools found in literature include inability to directly test Java applications and limitations in respect to types of emulated faults. The analysis of the fault injectors found in literature motivates the development of a solution specifically aimed at testing multi-protocol applications written in Java.

This work presents a solution for communication fault injection in multi-protocol Java applications. The solution works at JVM level, intercepting protocol messages, and, in some cases, it also operates at the operating system level, using firewall rules for the emulation of faults that could not be emulated at JVM level. The approach is useful for both white box and black box testing and has advantages such as preserving the target application's source code. The viability of the proposed solution is shown by the development of a prototype tool called Comform, that is aimed at multi-protocol Java application testing. Comform can be applied to test Java applications based on any combination of UDP, TCP, and RMI (including those based on a single protocol).

Keywords: Communication fault injection, multi-protocol Java applications.

1 INTRODUÇÃO

O desenvolvimento de aplicações de rede com altos requisitos de dependabilidade requer que a possibilidade de ocorrência de falhas de comunicação durante a execução dessas aplicações seja levada em consideração. Colapsos de nodos, colapsos de *links*, omissões de envio e recepção e outros tipos de falhas de comunicação ocorrem comumente em ambientes de rede. Visando à prevenção de defeitos, mecanismos de tolerância a falhas devem ser corretamente projetados e implementados. *Injeção de falhas* (HSUEH; TSAI; IYER, 1997) é a técnica mais adequada para testar se os mecanismos de tolerância a falhas de aplicações de rede comportam-se como esperado na ocorrência de falhas de comunicação.

Durante o desenvolvimento de *software*, o principal foco de atividades de teste é revelar falhas, de modo que elas possam ser removidas (PEZZÉ; YOUNG, 2008). Injeção de falhas é útil para *remoção de falhas* (AVIZIENIS et al., 2004), durante o desenvolvimento e durante ações de manutenção preventiva, provendo retorno aos desenvolvedores de *software* sobre evidências de incorreções em suas aplicações. Por outro lado, injeção de falhas também é importante para *previsão de falhas* (AVIZIENIS et al., 2004), ou seja, para a avaliação da dependabilidade de um sistema. Exemplos incluem *benchmarks* de dependabilidade (KANOUN; SPAINHOWER, 2008) e a avaliação de componentes, como componentes comerciais de prateleira – COTS – (BARBOSA et al., 2007), cujo desenvolvimento pode estar encerrado e/ou cujo código fonte nem sempre está disponível.

Considerando aplicações de rede, algumas delas são baseadas em mais de um protocolo de comunicação da arquitetura TCP/IP. O teste de tais aplicações – aqui denominadas *multiprotocolo* – em condições de falhas de comunicação, usando injeção de falhas, é o escopo deste trabalho. Optou-se por tratar aplicações Java baseadas em protocolos de comunicação que estão acima do nível de rede na pilha TCP/IP. Este tipo de aplicação é comum, pois diferentes protocolos são adequados a diferentes propósitos. Alguns exemplos interessantes de aplicações Java multiprotocolo encontradas na literatura incluem Zorilla (DROST; NIEUWPOORT; BAL, 2006) e Anubis (MURRAY, 2005). Algumas aplicações de demonstração do *middleware* para comunicação de grupo JGroups (BAN, 2002) também são multiprotocolo.

Assim como qualquer outra aplicação de rede com altos requisitos de dependabilidade, as aplicações Java multiprotocolo que possuem tais requisitos devem operar conforme sua especificação em presença de falhas. Visando a atingir esse objetivo, elas devem ser testadas adequadamente em condições de falhas de comunicação. Uma ferramenta de injeção de falhas apropriada deve, primordialmente, ter a capacidade de lidar adequadamente com todos os protocolos usados simultaneamente pela aplicação alvo. Outros requisitos levantados para tal ferramenta incluem a adequação para testes de caixa branca e de caixa preta e a preservação do código fonte da aplicação alvo. Entretanto,

os injetores de falhas encontrados na literatura ou não são capazes de testar esse tipo de aplicação ou impõem aos engenheiros de teste dificuldades bastante relevantes.

Alguns injetores de falhas de comunicação são voltados ao teste de aplicações Java baseadas em um único protocolo específico como, por exemplo, UDP (FARCHI; KRASNY; NIR, 2004) (JACQUES-SILVA et al., 2006) (SILVEIRA; WEBER, 2006), TCP (GERCHMAN; WEBER, 2006) ou RMI (VACARO; WEBER, 2006). No teste dos mecanismos de tolerância a falhas de uma aplicação multiprotocolo, caso a emulação de uma falha que afete a troca de mensagens não leve em consideração todos os protocolos simultaneamente utilizados, o comportamento emulado durante um experimento poderá ser diferente daquele observado na ocorrência de uma falha real. Consequentemente, podem ser obtidos resultados inconsistentes sobre o comportamento da aplicação alvo em presença da falha quando não é considerada a possibilidade desta falha afetar diferentes protocolos em uso simultâneo.

Algumas ferramentas são capazes de injetar falhas de comunicação sem limitação a um protocolo específico. Porém, a pesquisa realizada neste trabalho mostra que elas impõem outros tipos de dificuldades a engenheiros de teste. Por exemplo, contrariamente ao enfoque deste trabalho, ferramentas como FIRMAMENT (DREBES, 2005) são voltadas à injeção de falhas de comunicação para o teste de protocolos de comunicação, e não de aplicações. Tal orientação ao teste de protocolos costuma levar a grandes dificuldades no *teste de caixa branca* de aplicações. Testes de caixa branca, também chamados de estruturais, levam em consideração o código fonte da aplicação alvo (PEZZÉ; YOUNG, 2008). Entre outros exemplos de dificuldades proporcionadas por ferramentas da literatura estão a incapacidade de testar diretamente aplicações Java e a limitação quanto aos tipos de falhas que permitem emular.

1.1 Objetivos

O foco desta dissertação é o estudo do problema de injeção de falhas de comunicação em aplicações multiprotocolo. Conforme apresentado brevemente nesta introdução, não há ferramentas voltadas especificamente a esse fim. Assim, o maior objetivo do trabalho é a elaboração de uma solução para injeção de falhas de comunicação em *aplicações Java multiprotocolo*, considerando a possibilidade do uso simultâneo desses protocolos.

1.2 Resultados Alcançados

Entre os resultados alcançados por este trabalho, estão:

- Definição de requisitos para uma ferramenta de injeção de falhas de comunicação voltada ao teste e avaliação de aplicações Java multiprotocolo.

- Análise do potencial de ferramentas e *frameworks* existentes para a injeção de falhas em aplicações Java multiprotocolo e conclusão sobre a necessidade de uma nova ferramenta.

- Especificação de uma solução para o problema de injeção de falhas em aplicações Java multiprotocolo visando suprir a necessidade identificada. A solução é adequada tanto para a realização de testes de caixa branca como de caixa preta, podendo auxiliar tanto na remoção de falhas, como na previsão de falhas e na obtenção de métricas de dependabilidade. Por ser útil ao teste de caixa branca de aplica-

ções, ela possui benefícios em relação a ferramentas voltadas ao teste de protocolos, como FIRMAMENT (DREBES, 2005), que não são adequadas a esse propósito.

- Desenvolvimento de Comform (COMmunication Fault injector ORiented to Multi-protocol Java applications), um protótipo para injeção de falhas em aplicações Java multiprotocolo, buscando cumprir com os requisitos identificados para a solução. O protótipo pode ser aplicado para injetar falhas em aplicações Java baseadas em qualquer combinação dos protocolos UDP, TCP e RMI (incluindo aquelas baseadas em um único protocolo). Podem ser injetadas falhas, inclusive, em aplicações Java que fazem uso do pacote `java.nio` (SUN MICROSYSTEMS, 2008) para comunicação usando TCP e UDP. A instrumentação de classes de comunicação deste pacote ainda não havia sido tratada por trabalhos anteriores. Comform provê boa representatividade de falhas (GIL et al., 2002), ou seja, os erros induzidos são similares àqueles provocados por falhas reais. Também possui boa portabilidade e não requer a modificação do código fonte das aplicações alvo. Um benefício adicional proporcionado por Comform é oferecer ao usuário a capacidade de testar várias aplicações Java com o aprendizado de uma única ferramenta.

- Condução de experimentos de injeção de falhas com o protótipo desenvolvido, mostrando a viabilidade do modelo e da arquitetura propostos para o teste de aplicações Java multiprotocolo.

1.3 Organização do Texto

Esta dissertação está organizada da seguinte forma:

- O Capítulo 2, *Injeção de Falhas*, apresenta conceitos básicos de injeção de falhas (Seção 2.1) e ferramentas e *frameworks* de injeção de falhas capazes de injetar falhas de comunicação (Seção 2.2). A seguir, trata da injeção de falhas em aplicações multiprotocolo (Seção 2.3). Na mesma Seção, mostra exemplos de aplicações Java multiprotocolo, apresenta o problema de injeção de falhas em aplicações multiprotocolo e identifica os requisitos importantes para uma ferramenta de injeção de falhas voltada a aplicações Java multiprotocolo. Além disso, é feita uma análise do potencial das ferramentas e *frameworks* de injeção de falhas apresentados anteriormente para o teste de aplicações Java multiprotocolo, mostrando sua inadequação para esse problema. Neste Capítulo, também é feita uma comparação de Comform, o protótipo para injeção de falhas em aplicações Java multiprotocolo desenvolvido no contexto desse trabalho, a trabalhos relacionados.

- O Capítulo 3, *UDP, TCP e RMI em Java*, introduz os protocolos de comunicação UDP (Seção 3.1), TCP (Seção 3.2) e RMI (Seção 3.3) no contexto de sua arquitetura e implementação na linguagem Java. O leitor que possui conhecimento sobre essas APIs e o funcionamento dos protocolos em questão pode evitar a leitura desse Capítulo sem maiores prejuízos à compreensão do restante do texto.

- O Capítulo 4, *Questões de Projeto*, apresenta um modelo de solução para injeção de falhas de comunicação em aplicações multiprotocolo (Seção 4.1) e discute possibilidades de abordagens a serem empregadas no projeto de um injetor de falhas para aplicações Java (Seção 4.2).

- O Capítulo 5, *Injetor de Falhas Comform*, apresenta a proposta de arquitetura do injetor de falhas Comform (Seção 5.1). São detalhadas informações sobre as classes instrumentadas (Seção 5.2) e sobre componentes importantes da arquitetura como o Filtro de Mensagens (Seção 5.3), o Monitor (Seção 5.4) e modelagem de Falhas e Cargas de Falhas (Seção 5.5). Também é analisada a portabilidade de Comform (Seção 5.6) e apresentada uma discussão sobre a expansão de Comform para novos protocolos (Seção 5.7). Por fim, são discutidos aspectos práticos sobre o uso do protótipo (Seção 5.8).

- O Capítulo 6, *Experimentos de Injeção de Falhas*, apresenta experimentos de injeção de falhas conduzidos com Comform, mostrando a viabilidade do modelo e da arquitetura propostos.

- Por fim, o Capítulo 7, *Considerações Finais*, apresenta a conclusão deste trabalho e indicações de possibilidades de trabalhos futuros.

Ao final dos Capítulos 2, 3, 4, 5 e 6 são apresentadas conclusões parciais que sintetizam o conteúdo apresentado no Capítulo e indicam suas principais contribuições.

2 INJEÇÃO DE FALHAS

Este Capítulo apresenta conceitos básicos de injeção de falhas (Seção 2.1) e ferramentas e *frameworks* de injeção de falhas capazes de injetar falhas de comunicação (Seção 2.2). A Seção 2.3 trata da injeção de falhas em aplicações multiprotocolo. Ela mostra exemplos de aplicações Java multiprotocolo, apresenta o problema de injeção de falhas em aplicações multiprotocolo e identifica os requisitos importantes para uma ferramenta de injeção de falhas voltada a aplicações Java multiprotocolo. Além disso, é feita uma análise do potencial das ferramentas e *frameworks* de injeção de falhas apresentados anteriormente para o teste de aplicações Java multiprotocolo, mostrando sua inadequação para esse problema. Por fim, na Seção 2.4, é feita uma comparação de Comform, o protótipo para injeção de falhas em aplicações Java multiprotocolo desenvolvido no contexto desse trabalho, a trabalhos relacionados.

2.1 Conceitos Básicos

Além de serem propensas a falhas de desenvolvimento, assim como qualquer aplicação, aplicações de rede também são suscetíveis a falhas de comunicação, como colapsos de nodos, colapsos de *links* e omissões de envio e recebimento de mensagens. Visando à prevenção de defeitos na ocorrência de falhas de comunicação, mecanismos de tolerância a falhas podem ser empregados. A dependabilidade de tais sistemas deve ser avaliada para assegurar que estes mecanismos foram corretamente projetados e implementados e que o sistema irá prover o nível desejado de serviço confiável (CLARK; PRADHAN, 1995).

Dependabilidade foi definida originalmente como a capacidade de um sistema computacional fornecer serviço no qual se pode confiar de modo justificável; de forma alternativa, pode ser definida como a capacidade de evitar que defeitos de serviço sejam mais frequentes e mais graves do que é considerado aceitável (AVIZIENIS et al., 2004). Dependabilidade é um conceito integrador que engloba os seguintes atributos: confiabilidade, disponibilidade, *safety*, integridade e mantenabilidade (*maintainability*). Os dois primeiros atributos constituem as medidas primárias de dependabilidade (CLARK; PRADHAN, 1995) e correspondem, respectivamente, à probabilidade de um sistema permanecer funcional e sem defeitos em um determinado intervalo de tempo e à probabilidade de um sistema estar operacional e sem defeitos em um dado instante de tempo.

Injeção de falhas (CLARK; PRADHAN, 1995) (HSUEH; TSAI; IYER, 1997) pode ser definida como a inserção artificial de falhas em um sistema, de forma controlada, para observar o seu comportamento na presença de falhas e avaliar sua dependabilidade. Segundo Hsueh, Tsai e Iyer (HSUEH; TSAI; IYER, 1997), as técnicas de injeção de falhas podem ser classificadas em *baseadas em simulação* e *baseadas em protótipo*. As técnicas baseadas em simulação são úteis nas fases iniciais de projeto de um sistema.

Por outro lado, as técnicas baseadas em protótipo são usadas quando um protótipo do sistema alvo já está disponível. Considerando as técnicas de injeção de falhas baseadas em protótipo, falhas podem ser injetadas tanto em sistemas de *hardware* como de *software* usando métodos de injeção de falhas por *hardware* ou por *software*.

Os métodos de injeção de falhas implementados por *hardware* utilizam *hardware* adicional para inserir falhas no *hardware* do sistema alvo (HSUEH; TSAI; IYER, 1997). Segundo Hsueh, Tsai e Iyer (1997), os métodos de injeção de falhas por *hardware* são classificados em injeção com contato e injeção sem contato. No primeiro método, o injetor tem contato físico direto com o sistema alvo, produzindo mudanças de tensão ou corrente externamente ao *chip* alvo. No segundo método, uma fonte externa produz algum fenômeno físico, como interferência eletromagnética.

A injeção de falhas por *software*, ou seja, implementada em *software*, emula falhas em componentes de *hardware* ou *software*, acelerando manifestação de erros devidos a tais falhas. Os métodos de injeção de falhas por *software* apresentam algumas vantagens em relação à injeção de falhas por *hardware* (HSUEH; TSAI; IYER, 1997). Dentre elas, destacam-se dispensar *hardware* de custo alto e poderem ser usados para atingir aplicações e sistemas operacionais, o que é difícil fazer através de métodos de injeção de falhas por *hardware*. Apesar da injeção de falhas por *software* ser flexível, ela apresenta algumas deficiências (HSUEH; TSAI; IYER, 1997). Ela não pode inserir falhas em localizações inacessíveis ao *software*, pode perturbar a carga de trabalho em execução no sistema alvo e é inadequada para emular falhas de baixa latência.

A *injeção de falhas de software por software* ou *teste baseado em falhas* (*fault-based testing*) (PEZZÉ; YOUNG, 2008) tem como objetivo avaliar casos de teste, selecionando os mais capazes de distinguir comportamentos válidos de inválidos (SUGETA; MALDONADO; WONG, 2004). A técnica principal é o *teste de mutantes* (MASIERO et al., 2006): são geradas versões alteradas do programa ou da especificação original, usando operadores que os modificam de forma a emular erros comuns que ocorrem durante seu desenvolvimento. Esse trabalho não trata de injeção de falhas de *software*, mas de *injeção de falhas de comunicação* emuladas por *software* em aplicações Java multiprotocolo.

O injetor de falhas por *software* pode ser implementado como uma aplicação dedicada, que é integrada ao ambiente de execução do sistema alvo. Também é possível injetar falhas inserindo código de teste diretamente no código fonte da aplicação alvo, porém essa abordagem leva a uma grande intrusividade espacial e é pouco recomendada, pois *bugs* não intencionais podem ser inseridos na aplicação alvo ou mesmo pode haver modificação do fluxo de execução da aplicação.

Um injetor de falhas implementado por *software* pode realizar emulação de falhas de comunicação por meio da interceptação do envio e recebimento de pacotes ou mensagens. Os pacotes ou mensagens de interesse devem poder ser *selecionados* e *manipulados*.

2.1.1 Utilidades da Injeção de Falhas

Durante o desenvolvimento de *software*, o foco principal das atividades de teste é revelar falhas, de modo que elas possam ser removidas (PEZZÉ; YOUNG, 2008). Injeção de falhas pode ser utilizada, durante o processo de desenvolvimento, visando à *remoção de falhas* (AVIZIENIS et al., 2004) complementarmente a outras técnicas de teste.

Além de ser muito útil para auxiliar na remoção de falhas, existem outros usos importantes para a técnica de injeção de falhas, como a *previsão de falhas* (AVIZIENIS et al., 2004) e obtenção de métricas de dependabilidade. Avaliar a dependabilidade de um sistema empiricamente, esperando pela ocorrência natural de defeitos após ele ter entrado

em produção, é uma abordagem impraticável para a maioria dos sistemas tolerantes a falhas, pois o tempo necessário para obter um número de defeitos estatisticamente significativo pode ser inaceitável (CLARK; PRADHAN, 1995). Injeção de falhas é uma alternativa bastante eficiente em relação a essa abordagem de avaliação de dependabilidade. Exemplos de métricas que podem ser obtidas incluem cobertura de mecanismos de detecção e recuperação de erros e queda de desempenho provocada por falhas (HSUEH; TSAI; IYER, 1997).

Recentemente, alguns trabalhos têm sido feitos na área de *benchmarks* de dependabilidade para diferentes classes de sistemas (KANOUN; SPAINHOWER, 2008). O principal objetivo dos *benchmarks* de dependabilidade é geralmente a comparação de diferentes implementações de sistemas que possuem a mesma finalidade visando determinar qual o melhor. Injeção de falhas é uma técnica utilizada como base para *benchmarks* de dependabilidade (HOARAU; TIXEUIL; VAUCHELLES, 2007).

Injeção de falhas pode ser útil tanto para *testes de caixa preta* como para *testes de caixa branca*. Testes de caixa preta (PEZZÉ; YOUNG, 2008), também conhecidos como testes funcionais, não levam em consideração o conhecimento do código fonte da aplicação alvo. Por outro lado, testes de caixa branca (PEZZÉ; YOUNG, 2008), também chamados de estruturais, levam em consideração o código fonte da aplicação alvo. Conforme será mostrado neste Capítulo, nem todas as ferramentas de injeção de falhas são adequadas para a realização de ambos os tipos de teste em um determinado sistema alvo.

2.1.2 Ambiente de Injeção de Falhas

A fim de resumir o funcionamento da técnica de injeção de falhas, a Figura 2.1 apresenta os componentes básicos de um ambiente de injeção de falhas segundo Hsueh, Tsai e Iyer (1997). O injetor de falhas injeta falhas no sistema alvo enquanto o sistema alvo executa comandos do gerador de carga de trabalho (aplicações, *benchmarks* e cargas de trabalho sintéticas). Conforme mencionado anteriormente, o injetor de falhas pode ser implementado em *hardware* ou *software*. O monitor verifica a execução dos comandos e inicia a coleta de dados sempre que necessário. O coletor de dados realiza coleta de dados em tempo de execução do experimento. O analisador de dados, que pode operar posteriormente à execução, realiza processamento e análise de dados. O controlador é um programa responsável por controlar o experimento, que pode ser executado no sistema alvo ou em um computador separado. Uma biblioteca de falhas pode prover valores, por exemplo, para diferentes tipos de falhas, localizações de falhas e durações de falhas. A biblioteca de falhas, o gerador de carga de trabalho, o monitor e outros componentes podem ser implementados como componentes independentes para maior flexibilidade e portabilidade.

Na prática, este ambiente é customizado em cada projeto de injetor de falhas, podendo haver diversas variações em relação ao apresentado na Figura.

Para realizar um experimento de injeção de falhas, é preciso definir a carga de falhas a ser injetada na aplicação alvo e a carga de trabalho. A carga de trabalho (*workload*) consiste de uma lista de requisições de serviços para a aplicação alvo e é essencial que ela seja representativa da utilização do sistema em condições reais. A carga de falhas é a descrição das falhas a serem injetadas.

2.1.3 Falhas de Comunicação

Falhas de comunicação, como colapsos de nodos, colapsos de *links*, omissões de envio e recepção e particionamentos de rede, entre outras, ocorrem comumente em ambientes

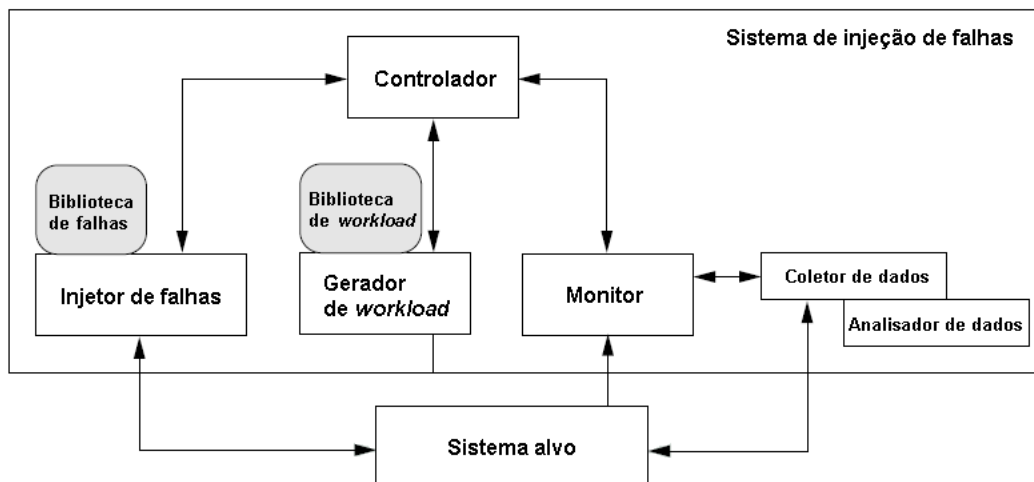


Figura 2.1: Componentes de ambiente de injeção de falhas (HSUEH; TSAI; IYER, 1997).

de rede.

No projeto de uma ferramenta de injeção de falhas, é preciso levar em consideração os modelos de falhas de suas aplicações alvo, para definir os tipos de falhas a serem implementados pela ferramenta. Um modelo de falhas especifica como uma falha pode se manifestar em um sistema, proporcionando compreensão sobre seus efeitos e consequências. Alguns exemplos de modelos de falhas para sistemas distribuídos encontrados na literatura incluem os abordados por Cristian (CRISTIAN, 1991), Schneider (SCHNEIDER, 1993) e Birman (BIRMAN, 1996).

Birman identifica modelos de falhas possíveis nestes sistemas, que são descritos a seguir. *Falhas de colapso* constituem um modelo de falhas em que um processo ou computador opera corretamente ou simplesmente pára sua execução e entra em colapso sem tomar ações incorretas no caso de falhas. A única maneira de detectar o colapso é por *time-out*. No caso do colapso de um computador, a denominação *colapso de nodo* é utilizada durante o restante do texto. *Falhas de parada segura (fail-stop)* constituem um modelo similar ao anterior, porém a falha é precisamente detectável pelos outros componentes do sistema. *Omissão de envio de mensagens* caracteriza-se por falhas para enviar uma mensagem que, de acordo com a lógica dos sistemas distribuídos, deveria ter sido enviada. Ela é frequentemente causada por falta de espaço para armazenamento nos *buffers* do sistema operacional ou interface de rede, a qual faz com que uma mensagem seja descartada depois que uma aplicação a enviou, mas antes que ela deixe a máquina de origem. *Omissão de recepção de mensagens* é semelhante à omissão de envio, mas manifesta-se quando uma mensagem é perdida nas proximidades do processo ao qual é destinada, frequentemente devido à falta de memória para armazená-la ou devido a evidências de corrupção de dados terem sido descobertas. *Falhas de rede* manifestam-se quando a rede perde mensagens enviadas entre pares de processos. *Falhas de particionamento de rede* manifestam-se quando uma rede se divide em uma ou mais subredes desconectadas levando nodos de diferentes subredes a perderem a capacidade de se comunicar. *Falhas de temporização* ocorrem quando um requisito temporal do sistema é violado. Por fim, *falhas bizantinas* incluem uma grande variedade de comportamentos falhos, como corrupção de dados ou até um comportamento malicioso da aplicação.

Cristian (CRISTIAN, 1991), ao contrário de Birman, considera a possibilidade de rei-

nicialização de um nodo após o sofrimento do colapso e distingue diferentes tipos de comportamento de colapso em relação a esta possibilidade: *colapso com amnésia* (*amnesia-crash*), *colapso com amnésia parcial* (*partial-amnesia-crash*), *colapso com pausa* (*pause-crash*) e *colapso com parada* (*halting-crash*). O *colapso com amnésia* ocorre quando um servidor reinicia em um estado inicial que não depende das entradas observadas antes do colapso. O *colapso com amnésia parcial* ocorre quando, ao reiniciar, alguma parte do estado é a mesma de antes do colapso, enquanto o restante volta a um estado inicial predefinido. O *colapso com pausa* ocorre quando um servidor reinicia no mesmo estado que tinha antes do colapso. Por fim, o *colapso com parada* ocorre quando um servidor não volta do colapso. Este último pode ser mapeado para a definição de colapso de Birman.

Para a construção de ferramentas de injeção de falhas, não há interesse na causa específica de cada falha e sim na capacidade da ferramenta emular as possíveis manifestações de cada falha (JACQUES-SILVA, 2005). A emulação de falhas de comunicação por *software* costuma ser feita afetando o sistema de troca de mensagens. Mensagens de interesse são *selecionadas* e *manipuladas* no seu envio e/ou recepção, de modo a emular o comportamento desejado.

2.1.4 Intrusividade

No projeto de injetores de falhas, é importante procurar minimizar dois tipos de intrusividade: a *temporal* e a *espacial*.

A intrusividade temporal ocorre devido ao aumento no tempo de execução da aplicação alvo em virtude da execução das atividades do injetor de falhas. Ela torna-se um problema crítico no teste de aplicações de tempo real. Injetores de falhas implementados por *software* elevam a intrusividade temporal e só é possível minimizar essa intrusividade, mas não anulá-la.

A intrusividade espacial é relacionada à modificação do código fonte da aplicação alvo. Ela é indesejável, já que pode levar a inserção de *bugs* não intencionais no código da aplicação e mesmo alterar o fluxo de execução da aplicação. Adicionalmente, o código fonte da aplicação alvo nem sempre está disponível. Ao contrário da intrusividade temporal, é possível evitar a intrusividade espacial.

2.2 Ferramentas e Frameworks de Injeção de Falhas

Esta Seção apresenta uma revisão bibliográfica sobre ferramentas e *frameworks* de injeção de falhas capazes de injetar *falhas de comunicação*, mostrando suas principais características. Diversas ferramentas de injeção de falhas de comunicação são encontradas na literatura, mas não foram encontrados trabalhos voltados especificamente ao teste de aplicações multiprotocolo. Algumas até possuem potencial para o teste dessas aplicações, mas impõem grandes dificuldades aos engenheiros de testes.

Primeiramente, são apresentados injetores de falhas recentemente desenvolvidos pelo Grupo de Tolerância a Falhas da Universidade Federal do Rio Grande do Sul. São eles: FIONA (JACQUES-SILVA et al., 2006), FIRMAMENT (DREBES, 2005), FIERCE (GERCHMAN; WEBER, 2006) e FIRMI (VACARO; WEBER, 2006).

A seguir, são apresentadas as ferramentas e/ou *frameworks* DOCTOR (HAN; SHIN; ROSENBERG, 1995), NFTAPE (STOTT et al., 2000), ORCHESTRA (DAWSON; JAHANIAN; MITTON, 1996), Loki (CHANDRA et al., 2004), FAIL-FCI (HOARAU; TIXEUIL; VAUCHELLES, 2007) e FIT (TOWNEND et al., 2008), que são bastante referenciados na literatura ou constituem trabalhos recentes. São apresentadas características

das ferramentas como, por exemplo, os sistemas alvo, tipos de falhas que são capazes de emular e a abordagem utilizada para injeção de falhas.

2.2.1 FIONA

FIONA (*Fault Injector Oriented to Network Applications*) (JACQUES-SILVA et al., 2006) é um injetor de falhas de comunicação para a validação de sistemas distribuídos construídos em Java e baseados no protocolo UDP. A ferramenta faz instrumentação da classe de comunicação de UDP responsável pelo envio e recebimento de mensagens. Tal instrumentação é feita usando os recursos da interface de monitoramento e depuração JVMTI (*Java Virtual Machine Tool Interface*) (SUN MICROSYSTEMS, 2006b).

Em Java, a classe `java.net.DatagramSocket` representa um *socket* para o envio e recebimento de datagramas UDP. O injetor de falhas FIONA pode ser usado para o teste de qualquer aplicação que use subclasses de `java.net.DatagramSocket`, desde que os métodos `send()` e `receive()` da superclasse sejam invocados. A ferramenta usa tais métodos como pontos de interceptação para a injeção de falhas.

O modelo de falhas adotado por FIONA inclui falhas de particionamento de rede, que são comuns em sistemas distribuídos de larga escala (JACQUES-SILVA et al., 2006). Este tipo de falha afeta mais de um nodo por vez e sua emulação é dificultada quando feita por meio de um injetor de falhas que opere em um nodo único ou quando não há uma ação coordenada entre injetores usados de maneira simultânea. As demais ferramentas desenvolvidas pelo grupo de Tolerância a Falhas da UFRGS não são capazes de emular este tipo de falha. Além deste tipo de falha, a ferramenta suporta emulação de falhas de omissão de mensagens, colapso de nodos, colapso de *links*, falhas de temporização e duplicação de pacotes.

FIONA possui uma arquitetura local e uma arquitetura distribuída (JACQUES-SILVA et al., 2006). A arquitetura local é composta de 3 blocos: o agente JVMTI, codificado na linguagem C, as classes de injeção de falhas e o protocolo instrumentado, ambos codificados em Java. A arquitetura distribuída é usada para a condução de experimentos (como, por exemplo, aqueles que incluem falhas de particionamento) onde é necessário que o injetor seja instanciado em mais de um nodo. A proposta da arquitetura distribuída de FIONA tem o objetivo de cumprir os principais objetivos da ferramenta: (1) escalabilidade, (2) possibilidade de configuração e análise de *log* do experimento de forma centralizada e (3) emulação de um modelo de falhas consistente com sistemas de larga escala.

A arquitetura distribuída de FIONA é baseada em arquiteturas de monitoramento de larga escala visando alcançar escalabilidade (JACQUES-SILVA et al., 2006). Duas estratégias escaláveis para monitoramento de sistemas distribuídos de larga-escala, GRM (BALATON et al., 2001) e Ganglia (MASSIE; CHUNB; CULLER, 2003), foram combinadas na arquitetura distribuída de FIONA, adaptando uma arquitetura de monitoramento para uma arquitetura de injeção distribuída de falhas.

GRM estabelece uma hierarquia entre os monitores, classificando-os em monitor principal, monitor de *site* e monitor local. Monitores locais são executados em cada nodo do sistema e se comunicam com os monitores de *site*. Cada monitor de *site* recolhe e passa todas as informações ao monitor principal, que coordena os monitores de *site* e a coleta global de dados. Ganglia usa UDP *multicast* para comunicação intra-cluster e TCP para comunicação inter-cluster. FIONA combina a arquitetura hierárquica proposta por GRM e a estratégia de protocolos usada por Ganglia, fazendo comunicação entre injetores locais e injetores de *site* baseada em UDP *multicast* e comunicação entre injetores de *site* e injetor principal via TCP.

2.2.2 FIRMAMENT

FIRMAMENT (*Fault Injection Relocatable Module for Advanced Manipulation and Evaluation of Network Transports*) (DREBES, 2005) é uma ferramenta de injeção de falhas implementada como um módulo de núcleo do sistema operacional Linux. Trata-se da evolução de uma ferramenta anterior chamada ComFIRM (*Communication Fault Injection through Operating System Resource Modification*) (LEITE, 2000), que tinha a desvantagem de ser codificada diretamente no núcleo (*kernel*) do sistema operacional. FIRMAMENT tem acesso total aos fluxos de entrada e saída de pacotes e é capaz de injetar falhas nos protocolos IPv4 e IPv6, sendo o teste de protocolos baseados nestes dois últimos protocolos o foco principal da ferramenta.

FIRMAMENT utiliza Netfilter (RUSSEL; WELTE, 2002) como gancho na pilha de protocolos do núcleo Linux. Netfilter é um *framework* de núcleo para manipulação de pacotes que define ganchos na pilha de protocolos onde funções de *callback* podem ser registradas. Quando os pacotes atravessam os ganchos, são passados às funções de *callback*, que possuem acesso completo a seu conteúdo. As funções interpretam, para cada pacote enviado e/ou recebido, micro-programas, chamados *faultlets*, que tem capacidade de alterar o conteúdo dos pacotes e retornar à pilha de protocolos a ação final a ser realizada sobre o pacote. Tal ação pode ser o atraso, descarte ou processamento normal do pacote. A execução de *faultlets* é feita pela máquina virtual FIRMVM, que associa efetivamente o pacote ao *faultlet* e às variáveis de estado.

Faultlets são especificados usando uma linguagem de *bytecode* desenvolvida para descrever um experimento através de códigos de operação. Esta linguagem contém um conjunto de 31 instruções que proporcionam alto poder de expressão dos cenários de falhas. Elas permitem a inspeção e seleção de mensagens de forma determinística ou estatística e fornecem diversas ações a serem realizadas sobre os pacotes e variáveis internas do injetor, fazendo-o imitar o comportamento de falhas reais, como descarte e duplicação de mensagens, atraso e modificação de conteúdo.

2.2.3 FIERCE

FIERCE (*Fault Injection Environment for Remote Communication Evaluation*) (GERCHMAN; WEBER, 2006) é um injetor de falhas de comunicação voltado ao teste de aplicações Java construídas sobre o protocolo TCP. FIONA serviu como inspiração para o desenvolvimento desta ferramenta.

A ferramenta emula erros que o protocolo TCP/IP não mascara. O modelo de falhas adotado em seu desenvolvimento é o de Neves e Fuchs (1997), que usa apenas os códigos de erro retornados pela biblioteca de *sockets* do sistema operacional e o conhecimento prévio dos estados que os módulos TCP podem assumir para o desenvolvimento de módulos de detecção de falhas de comunicação para aplicações distribuídas tolerantes a falhas. Quatro tipos de falhas que resultam no término de um processo com diferentes comportamentos na interface de *sockets* são considerados: término de processo, colapso, reinicialização e colapso com reinicialização.

A ferramenta faz instrumentação das classes de comunicação responsáveis pelo estabelecimento de uma conexão TCP e pelo envio e recebimento de mensagens usando a interface de depuração e monitoramento JVMTI (*Java Virtual Machine Tool Interface*) (SUN MICROSYSTEMS, 2006b) para emulação das falhas descritas em um cenário de falhas. FIERCE é composto de um agente JVMTI, classes de comunicação instrumentadas e classes auxiliares para injeção de falhas. Sua ativação é completamente transparente

para a aplicação alvo.

2.2.4 FIRMI

FIRMI (*Fault Injector for RMI*) (VACARO; WEBER, 2006) é uma ferramenta de injeção de falhas para o teste de aplicações Java construídas sobre o protocolo RMI.

O modelo de falhas da ferramenta inclui falhas de colapso de nodo, colapso de *link*, temporização e bizantinas. FIRMI não considerou na sua especificação a ação coordenada entre os injetores usados de maneira simultânea em um experimento, não sendo possível, assim, a emulação de falhas de particionamento de rede.

A ferramenta implementa uma abordagem mista para a injeção de falhas de colapso de nodo e falhas de colapso de *link*, que combina a interceptação de requisições RMI no nível da JVM e a interação com o sistema operacional através de arquiteturas de *firewall* (VACARO, 2007). RMI é um protocolo que opera sobre o TCP e, conforme análise feita por Vacaro (VACARO, 2007), a emulação das falhas de RMI apenas no nível da Máquina Virtual Java não é precisa, pois o gerenciamento do protocolo TCP é feito pelo núcleo (*kernel*) do sistema operacional. Além disso, Vacaro (VACARO, 2007) também concluiu que emular as falhas de RMI apenas no nível do núcleo do sistema operacional é inviável, pois o fluxo de RMI neste nível de abstração se encontra em forma serializada (exigindo a análise da *stream* de dados a fim de extrair apenas as informações úteis ao injetor) e devido à dificuldade de identificação de um processo específico para uso do injetor no nível do núcleo.

Com a interceptação de requisições RMI no nível da JVM, é possível interagir diretamente com os módulos de RMI, obtendo informações ou alterando seu comportamento através da configuração de parâmetros. Assim, é possível monitorar as informações presentes no fluxo de requisições RMI antes que as mesmas sejam serializadas pelo protocolo. As informações coletadas são então usadas para criar as regras de *firewall*. Segundo Vacaro (VACARO, 2007), o uso de um *firewall* para emular falhas de colapso permite reproduzir com precisão o comportamento do protocolo TCP, pois características como confirmação de entrega de dados, *timeout* para retransmissão de pacotes e gerenciamento dos *buffers* de comunicação são efetivamente realizadas, já que pacotes são descartados pelo próprio sistema operacional e não pelos níveis superiores da aplicação. O bloqueio do tráfego não se restringe apenas aos endereços e portas alocados por RMI, mas também é possível bloquear portas adicionais como a porta TCP/UDP 7 (echo), tráfego ICMP ou até protocolos do nível de enlace como ARP e, deste modo, o colapso de nodos e canais de comunicação é emulado para aqueles mecanismos de tolerância a falhas não construídos integralmente em RMI (VACARO, 2007).

2.2.5 DOCTOR

DOCTOR (*Integrated Software Fault Injection Environment*) (HAN; SHIN; ROSENBERG, 1995) (HAN; SHIN, 1999) é um ambiente integrado de injeção de falhas por *software* para a validação e avaliação de sistemas distribuídos de tempo real capaz de (1) gerar cargas de trabalho sintéticas, (2) injetar vários tipos de falhas com diferentes opções e (3) coletar dados relativos a desempenho e dependabilidade.

O modelo de falhas de DOCTOR inclui três classes de falhas: falhas de memória, falhas de CPU e falhas de comunicação. Cada falha injetada pode ser permanente, transiente ou intermitente. O sistema provê uma interface gráfica que permite ao usuário a especificação do momento de injeção de falhas. Um plano de injeção de falhas pode ser formulado probabilisticamente ou baseado no histórico de eventos passados. Falhas em

links de comunicação podem ser emuladas usando uma camada especial de injeção de falhas que é embarcada na pilha de protocolos do sistema operacional em tempo de compilação. DOCTOR permite a emulação de vários tipos predefinidos de falhas de comunicação incluindo perda, duplicação, corrupção e atraso de mensagens. Ele ainda permite que usuários especifiquem comportamentos falhos fornecendo funções que serão chamadas pela camada especial de injeção de falhas. Não é possível fazer injeção de falhas de colapso usando DOCTOR, apesar deste tipo de falha ser muito comum em sistemas distribuídos.

A primeira versão de DOCTOR era restrita ao teste do sistema distribuído de tempo real HARTS (SHIN, 1991). Em um artigo posterior (HAN; SHIN, 1999), DOCTOR é usado para avaliação experimental de dois esquemas de detecção de defeitos baseados em comportamento para serviços de comunicação confiáveis de tempo real (detecção de vizinhos e detecção fim-a-fim).

2.2.6 NFTAPE

NFTAPE (STOTT et al., 2000) (STOTT et al., 2002) é um *framework* para avaliação de dependabilidade em sistemas distribuídos por meio de injeção de falhas. Ele foi criado visando superar duas limitações principais encontradas em outras ferramentas de injeção de falhas: (1) nenhuma delas foi considerada suficiente para injetar todos os modelos de falhas necessários e (2) há dificuldade em portá-las a novos sistemas. O objetivo do *framework* é a composição de experimentos de injeção de falhas automatizados a partir de injetores "leves" de falhas (*LightWeight Fault Injector* – LWFI), mecanismos para disparar a injeção (gatilhos, do inglês *triggers*), monitores e outros componentes disponíveis.

Injetores leves no NFTAPE são pequenos programas, muito mais simples do que um injetor de falhas tradicional, responsáveis por injetar falhas. Há diversos tipos de LWFI incluindo baseados em *driver*, baseados em depurador, específicos a um determinado alvo, baseados em simulação e baseados em *hardware*. Ao contrário de injetores tradicionais, eles geralmente não incluem mecanismos para disparar a injeção. O LWFI usa uma função da API do NFTAPE para esperar por um disparo de falha.

O propósito de um mecanismo para disparar a injeção é informar a um LWFI quando injetar uma falha. Dois exemplos simples de gatilhos são temporizadores e *breakpoints*. Em um exemplo mais complicado, uma falha pode ser disparada quando um sistema ou aplicação entra em um estado particular. Para injetar falhas quando uma aplicação está em um estado específico, a aplicação pode ser modificada para suportar o disparo da falha. Em NFTAPE, a saída de um gatilho é uma mensagem solicitando a injeção de uma falha. Como um LWFI pode receber uma mensagem deste tipo de qualquer processo que dispare injeção (LWFI e os mecanismos que disparam a injeção usam uma interface padrão), NFTAPE pode reusar um gatilho desenvolvido por uma equipe com um injetor de falhas desenvolvido por outra.

Segundo Stott et al. (2000), um injetor de falhas existente pode ser usado com NFTAPE sem modificação. Isto é atingido adicionando um programa empacotador (*wrapper*), que aguarda um evento de disparo usando a API do NFTAPE, e, então, dispara uma falha no injetor de falhas existente utilizando o método de entrada requerido por aquele injetor.

2.2.7 ORCHESTRA

ORCHESTRA (DAWSON; JAHANIAN; MITTON, 1996) (DAWSON et al., 1996) (DAWSON; JAHANIAN; MITTON, 1997) é uma ferramenta de injeção de falhas por

software para o teste de propriedades temporais e de dependabilidade de protocolos distribuídos.

ORCHESTRA é baseado em um *framework* simples, mas poderoso, chamado sondagem e injeção de falhas dirigidas por *scripts* (DAWSON; JAHANIAN; MITTON, 1996). A ênfase desta abordagem é em técnicas experimentais que visam a identificar "problemas" específicos em um protocolo ou em sua implementação preferencialmente a avaliar a dependabilidade de um sistema através de métricas estatísticas como cobertura de falhas. Assim, o foco é no desenvolvimento de técnicas de injeção de falhas que podem ser empregadas no estudo de três aspectos de um protocolo alvo (DAWSON; JAHANIAN; MITTON, 1996): (1) detecção de erros de projeto e de implementação, (2) identificação de violações de especificações de protocolo e (3) obtenção de discernimento sobre as decisões de projeto tomadas pelos implementadores.

A abordagem de sondagem e injeção de falhas dirigidas por *scripts* considera um protocolo distribuído como uma abstração através da qual uma coleção de participantes se comunicam através da troca de um conjunto de mensagens (DAWSON; JAHANIAN; MITTON, 1996). Cada protocolo é especificado como uma camada na pilha de protocolos de modo que cada camada provê um serviço abstrato de comunicação para camadas superiores. Na abordagem de ORCHESTRA, uma camada de sondagem/injeção de falhas (PFI) é inserida entre quaisquer duas camadas consecutivas em uma pilha de protocolos. A camada PFI pode executar *scripts* de teste determinísticos ou gerados randomicamente para sondar os participantes e injetar várias falhas no sistema. Interceptando e filtrando mensagens entre duas camadas de uma pilha de protocolos, a camada PFI pode atrasar, descartar, reordenar, duplicar e modificar mensagens.

2.2.8 Loki

Loki (CHANDRA et al., 2004) é um injetor de falhas para sistemas distribuídos que leva em consideração o estado global do sistema distribuído como gatilho de falhas. Os autores de Loki classificam NFTAPE (STOTT et al., 2000) e DOCTOR (HAN; SHIN; ROSENBERG, 1995) como ferramentas que usam gatilhos baseados em estado local. Loki foi projetado com os objetivos de baixa intrusividade temporal, alta precisão e alta flexibilidade em relação a escolha dos tipos de falhas e dos tipos de sistemas que podem ser estudados. Segundo Chandra et al. (2004), Loki atinge seus objetivos usando as ideias de visão parcial do estado global, sincronização otimista e análise posterior à execução.

Loki atinge flexibilidade por meio da separação entre mecanismos usados para realizar injeção de falhas e a política da injeção de falhas. Por mecanismos usados para realizar injeção de falhas, entende-se tarefas independentes de aplicação como execução de experimentos, coleta de medidas, sincronização de relógios, análise de resultados e obtenção de medidas. Por política da injeção de falhas, entende-se tarefas dependentes de aplicação como especificação de máquina de estados, tipos de falhas e especificação de mecanismos para disparar a injeção de falhas (gatilhos, do inglês *triggers*).

Loki faz injeção de falhas com base em uma visão parcial do estado global do sistema. Considerando o princípio de que, em geral, os mecanismos para disparar a injeção de falhas em um nodo particular dependem do estado de somente alguns outros nodos, Loki otimiza a quantidade de tráfego de atualização de estado na rede enviando notificações de mudança de estado somente entre os nodos necessários e monitorando somente a parte necessária do estado global em cada um dos nodos.

Enquanto controla mecanismos para disparar a injeção de falhas e realiza injeção, Loki assume, de forma otimista, que os nodos estão sincronizados corretamente devido

às notificações de mudança de estado, isto é, assume que a visão local do estado global é correta. Entretanto, esta suposição pode resultar em injeções de falhas nos estados globais errados. Isso é remediado durante uma análise posterior à execução por meio da verificação de cada injeção de falhas e descarte dos experimentos com injeções de falhas incorretas. Esta análise também estabelece eventos e injeções em uma única linha de tempo global.

A capacidade de estimação de medidas estatísticas da ferramenta é aplicada aos experimentos com injeções de falhas corretas para estimar medidas de desempenho e dependabilidade especificadas pelo usuário. A ferramenta inclui uma linguagem flexível de medidas projetada para facilitar a especificação de uma grande quantidade de medidas e uma interface gráfica que assiste ao usuário na especificação, execução e obtenção de medidas em campanhas de injeção de falhas.

2.2.9 FAIL-FCI

FAIL-FCI (HOARAU; TIXEUIL; VAUCHELLES, 2007) é uma ferramenta para injeção de falhas em aplicações distribuídas. FAIL (*FAult Injection Language*) (HOARAU; TIXEUIL; VAUCHELLES, 2007) é uma linguagem abstrata de alto nível desenvolvida para a descrição de cenários de falhas. FCI (*FAIL Cluster Implementation*) (HOARAU; TIXEUIL; VAUCHELLES, 2007) é uma plataforma distribuída de injeção de falhas que usa FAIL como linguagem de entrada para a descrição de cenários de falhas. Um cenário de falhas elaborado em FAIL descreve máquinas de estado que modelam ocorrências de falhas. Por meio da linguagem FAIL, também são descritas associações entre cada máquina de estados e um computador ou um grupo de computadores onde a aplicação será executada.

A plataforma FCI é composta de diversos blocos de construção, como o compilador FCI, a biblioteca FCI e o *daemon* FCI (HOARAU; TIXEUIL; VAUCHELLES, 2007). O compilador FCI é responsável pela pré-compilação dos cenários escritos em FAIL, gerando arquivos fonte C++ e arquivos de configuração padrão. A biblioteca FCI é empacotada juntamente com os arquivos gerados pelo compilador FCI e o arquivo resultante é distribuído para as máquinas alvo de acordo com arquivos de configuração definidos pelo usuário. *Daemons* FCI são executáveis gerados pela extração e compilação dos arquivos fonte distribuídos. Eles são responsáveis pela execução da aplicação sob teste, permitindo sua instrumentação e seu tratamento de acordo com o cenário de falhas.

FAIL-FCI é capaz de injetar falhas usando uma abordagem *quantitativa* ou *qualitativa*. A abordagem quantitativa é a encontrada mais usualmente em artigos sobre avaliação de tolerância a falhas e é usada para obtenção de medidas quantitativas. A abordagem qualitativa corresponde à injeção de falhas em pontos muito específicos do código fonte da aplicação alvo para estudar seu comportamento em casos particulares.

Quanto aos tipos de falhas de comunicação que podem ser emulados por FAIL-FCI, só foram encontrados relatos sobre injeção de falhas de colapso de processo (e não de nodo) e de suspensão de processo para emulação de uma máquina sobrecarregada.

2.2.10 FIT

FIT (Fault Injection Technology) é um *framework* desenvolvido pela Universidade de Leeds para injeção de falhas no nível de rede visando à avaliação de dependabilidade de *middlewares*. Com ele, não é necessário modificar o código fonte da aplicação alvo.

WS-FIT (LOOKER; MUNRO; XU, 2004) é uma implementação do *framework* FIT para a avaliação de dependabilidade de *Java Web Services* implementados usando Apache

Axis.

Grid-FIT (LOOKER; XU, 2007) é uma implementação do *framework* FIT para avaliação de serviços do *middleware* para *grade* (do inglês *grid*) Globus (FOSTER, 2006), mais especificamente, o Globus Toolkit 4. Uma grande parte do Globus Toolkit 4 é construída com base em Web Services, utilizando Apache Axis. Grid-FIT é uma derivação de WS-FIT. Os autores demonstram que Grid-FIT pode ser aplicado para avaliar a dependabilidade de grades baseadas em Globus, auxiliando na remoção de falhas.

CROWN-FIT (TOWNEND et al., 2008) é uma implementação do *framework* FIT para a avaliação dos *middlewares* CROWN-C e Globus Toolkit.

Grid-FIT e CROWN-FIT são implementados como *plug-ins* para o Eclipse, que é um *framework* independente de plataforma para o desenvolvimento de aplicações. Em ambos, as falhas são injetadas na camada de transporte do *middleware*. Ao invés de interceptar pacotes em um nível mais baixo, as mensagens SOAP (Simple Object Access Protocol) do *middleware* é que são interceptadas como entidades completas. As mensagens interceptadas são decodificadas em tempo real e é possível corromper, reordenar e descartar mensagens completas. Decodificando as mensagens do *middleware* e injetando falhas neste nível de abstração, é possível realizar perturbações de parâmetros similares àquelas possíveis através de alteração direta do código fonte do sistema alvo, mas com a vantagem de não necessitar realizar essa modificação, evitando a intrusividade espacial.

2.3 Injeção de Falhas em Aplicações Multiprotocolo

Esta Seção mostra exemplos de aplicações Java multiprotocolo (Seção 2.3.1), apresenta o problema de injeção de falhas em aplicações multiprotocolo (Seção 2.3.2) e identifica os requisitos importantes para uma ferramenta de injeção de falhas voltada a aplicações Java multiprotocolo (Seção 2.3.3). Além disso, é feita uma análise do potencial das ferramentas e *frameworks* de injeção de falhas apresentados na Seção 2.2 para o teste de aplicações Java multiprotocolo, mostrando sua inadequação para esse problema (Seção 2.3.4). Não foram encontrados trabalhos sobre ferramentas de injeção de falhas na literatura que tratem diretamente a questão do uso de múltiplos protocolos por uma aplicação.

2.3.1 Aplicações Multiprotocolo

Conforme abordado anteriormente, considerando aplicações de rede, algumas delas são baseadas em mais de um protocolo de comunicação da arquitetura TCP/IP. Neste trabalho, tais aplicações são denominadas aplicações *multiprotocolo*. O interesse maior reside, aqui, nas aplicações multiprotocolo escritas em Java e baseadas em protocolos de comunicação pertencentes a diferentes camadas acima do nível de rede na pilha de protocolos TCP/IP.

Aplicações multiprotocolo são relativamente comuns. Isto se deve ao fato de que diferentes protocolos de comunicação podem ser empregados para diferentes propósitos em uma mesma aplicação de rede, já que cada protocolo possui diferentes características. O protocolo de transporte TCP, por exemplo, garante transferência confiável de pacotes, uma vez que uma conexão tenha sido estabelecida, usando uma técnica conhecida como confirmação positiva com retransmissão. Já UDP é um protocolo de transporte mais simples, que não garante entrega confiável, isto é, pacotes podem ser perdidos e entregues fora de ordem. Por outro lado, UDP provê um desempenho superior ao TCP. Ainda, há protocolos de mais alto nível, como RMI. Java RMI abstrai a interface de comunicação ao nível de uma invocação de método, evitando que o programador seja obrigado a definir

um protocolo de aplicação. Nada impede que um desenvolvedor de aplicações use, por exemplo, em uma mesma aplicação de rede, RMI para invocação de métodos remotos, UDP para comunicação de voz e TCP para detecção de certos tipos específicos de falha.

Na literatura sobre aplicações distribuídas tolerantes a falhas, não é muito comum que exista menção detalhada aos protocolos de comunicação utilizados, já que, em geral, as soluções são apresentadas em um nível mais alto de abstração. Ainda assim, foram encontrados exemplos interessantes de aplicações Java multiprotocolo tolerantes a falhas, como Zorilla (DROST; NIEUWPOORT; BAL, 2006) e Anubis (MURRAY, 2005). Algumas aplicações exemplo do *middleware* para comunicação de grupo JGroups (BAN, 2002) também são multiprotocolo. Tais aplicações fazem uso simultâneo dos protocolos, ou seja, podem usar mais de um protocolo de comunicação em uma única execução. A seguir, são apresentados, com maiores detalhes, os exemplos Anubis e Zorilla.

Anubis (MURRAY, 2005) é um serviço distribuído de monitoramento que provê capacidades de descoberta, monitoramento de estado e detecção de defeitos. Ele é voltado ao gerenciamento de aplicações adaptativas, ou seja, programadas para serem reconfiguradas dinamicamente de modo que podem ser adaptadas aos recursos disponíveis e recuperar-se de defeitos. O serviço Anubis é implementado em Java como um grupo de servidores. Cada servidor inclui um gerente de partição e um gerente de estado. O gerente de partição usa *multicast* UDP para descoberta e comunicação de tempo com outros servidores. Além disso, usa TCP para prover comunicação entre servidores com garantia de ordenamento das mensagens.

Zorilla (DROST; NIEUWPOORT; BAL, 2006) é um *middleware peer-to-peer*, implementado em Java, que visa à execução de aplicações de supercomputação em grades computacionais. O projeto de Zorilla é baseado em uma rede de nodos ou *peers*. Cada nodo é capaz de tratar submissão, escalonamento e execução de *jobs* e armazenamento de arquivos. Em sua implementação, um nodo faz *broadcast* de pacotes UDP periodicamente na rede local para procurar por outros nodos. O endereço de um ou mais nodos existentes da rede é necessário para se unir a um *overlay*. Através de TCP, nodos se conectam diretamente um a outro para o envio eficiente de grandes quantidades de dados. Também através de TCP, é permitido que programas externos se conectem a Zorilla para, por exemplo, submeter um *job*.

2.3.2 Dificuldades na Injeção de Falhas em Aplicações Java Multiprotocolo

Há algumas dificuldades relacionadas à injeção de falhas de comunicação em aplicações Java multiprotocolo. Caso a emulação de uma falha que afete a troca de mensagens não leve em consideração todos os protocolos simultaneamente utilizados, o comportamento emulado durante um experimento poderá ser diferente daquele observado na ocorrência de uma falha real. Consequentemente, podem ser obtidos resultados inconsistentes sobre o comportamento da aplicação alvo em presença da falha quando não é considerada a possibilidade desta falha afetar diferentes protocolos em uso simultâneo.

A fim de ilustrar as dificuldades relacionadas à injeção de falhas de comunicação para o teste de aplicações multiprotocolo, é útil imaginar como um engenheiro de testes procederia caso recebesse a tarefa de testar uma aplicação de tal tipo. Um exemplo de aplicação multiprotocolo, como Zorilla (DROST; NIEUWPOORT; BAL, 2006), que usa UDP e TCP como protocolos de comunicação, pode ser considerado como a aplicação a ser testada.

Para avaliar o comportamento de uma aplicação multiprotocolo em presença de falhas de comunicação usando injeção de falhas, um injetor de falhas de comunicação adequado

deveria ser selecionado ou desenvolvido para o seu teste. Como requisito essencial, a ferramenta deve ser capaz de injetar falhas de comunicação em aplicações Java multiprotocolo. Como será mostrado, é inconsistente primeiramente testar uma aplicação multiprotocolo em que há uso simultâneo de protocolos usando um injetor de falhas voltado ao teste de aplicações Java baseadas somente em UDP, como FIONA (JACQUES-SILVA et al., 2006) (JACQUES-SILVA, 2005), e então testá-la usando um injetor de falhas voltado ao teste de aplicações Java baseadas em TCP, como FIERCE (GERCHMAN; WEBER, 2006).

Os métodos `send()` e `receive()` de `java.net.DatagramSocket`, responsáveis pelo envio e recebimento de mensagens UDP, são instrumentados em FIONA para emulação de falhas de omissão, colapso de *link* e colapso de nodo. A implementação de colapso de nodo é feita através da supressão da comunicação UDP entre o nodo onde FIONA está sendo executado e os demais nodos do sistema, enquanto a implementação de colapso de *link* é feita pela supressão da comunicação UDP em um determinado canal. Já a implementação de falhas de omissão é feita através do descarte de pacotes UDP, conforme especificado pela carga de falhas. Assim, caso FIONA seja usado para o teste de uma aplicação que usa TCP e UDP, como Zorilla, colapsos e omissões não serão emulados de forma realista, pois *somente a comunicação UDP será afetada*. Outro injetor de falhas de comunicação que visa ao teste de aplicações Java baseadas em UDP (FARCHI; KRASNY; NIR, 2004), e que possui algumas características similares à FIONA, leva aos mesmos problemas no teste de aplicações multiprotocolo. A ferramenta FIERCE (GERCHMAN; WEBER, 2006) é voltada ao teste de aplicações Java construídas sobre o protocolo TCP e, assim como FIONA, faz a emulação das falhas somente afetando os fluxos de seu protocolo alvo. Portanto, FIERCE tampouco poderia ser usado para o teste de Zorilla.

A Figura 2.2 mostra um exemplo de tentativa de injeção de falhas no *middleware* Zorilla usando FIONA, visando à ilustrar a inadequação de ferramentas de injeção de falhas voltadas ao teste de aplicações baseadas em um único protocolo para o teste de aplicações multiprotocolo. Os nodos 3, 4 e 5 compõem uma rede de nodos Zorilla. O nodo 1 está fazendo *broadcast* de pacotes UDP com o objetivo de se unir à rede. O nodo 2 representa um programa externo conectando-se a Zorilla para submissão de um *job*. A injeção de uma falha de colapso no nodo 5 usando FIONA irá resultar em um teste inconsistente no qual o colapso não é percebido pelo nodo 2, pois somente a comunicação UDP está sendo suprimida. Problemas similares irão ocorrer caso seja usado um injetor voltado ao teste de aplicações baseadas somente em TCP, como FIERCE. A solução proposta, que será apresentada nos próximos Capítulos, lida com múltiplos protocolos, superando esse problema.

Zorilla (DROST; NIEUWPOORT; BAL, 2006) foi utilizado como exemplo aqui, pois ele é bastante adequado para ilustrar o problema de forma facilmente compreensível. O teste de outras aplicações Java multiprotocolo sofreria de problemas similares caso uma ferramenta voltada ao teste de aplicações baseadas em um único protocolo, como FIONA, fosse usada.

2.3.3 Requisitos Identificados para a Solução

Além de ser capaz de injetar falhas em aplicações Java multiprotocolo, alguns outros requisitos importantes foram identificados para a solução. Tais requisitos, que ajudam a minimizar as dificuldades enfrentadas por um engenheiro de testes, são listados abaixo:

- Deve ser capaz de emular os tipos de falhas de comunicação que podem ocorrer co-

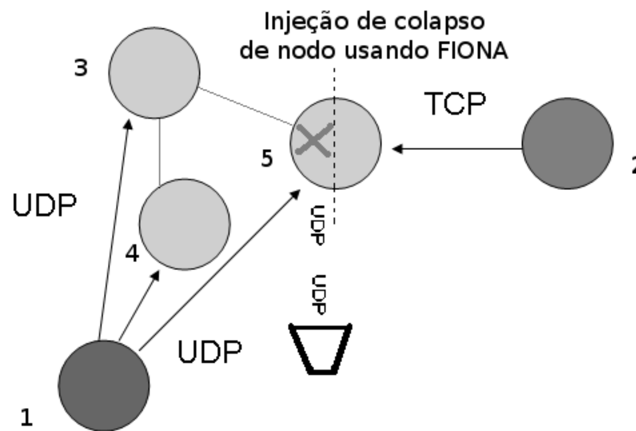


Figura 2.2: Inadequação de injeção de falha de colapso em um nodo Zorilla usando FIONA.

mumente em ambientes de rede, como, por exemplo, colapsos de nodos, colapsos de *links*, falhas de temporização, omissão e particionamento de rede. Caso contrário, a cobertura dos testes pode ser pobre em muitos casos, como no teste de aplicações que possuem altos requisitos de dependabilidade.

- Deve preservar o código fonte da aplicação alvo, pois o código fonte nem sempre está disponível e, mesmo se disponível, sua modificação leva a uma grande intrusividade espacial.
- Deve ser capaz de injetar falhas tanto independentemente do conhecimento do código fonte da aplicação alvo (para testes de caixa preta) como também levando em consideração o código fonte da aplicação alvo (para testes de caixa branca).
- Deve prover um mecanismo adequado para descrição de cargas de falhas, evitando dificuldades que levem o engenheiro de testes à desistência da condução de certos experimentos.
- Deve, preferencialmente, prover portabilidade, como, por exemplo, não ser exclusiva para um único sistema operacional. Assim, a aplicabilidade da ferramenta é maior.

Na literatura, conforme abordado anteriormente, há muitos outros injetores de falhas capazes de injetar falhas de comunicação. Entretanto, a maioria deles não é capaz de testar aplicações Java multiprotocolo ou impõe desvantagens relevantes aos engenheiros de teste. Na próxima Seção, algumas dessas ferramentas são analisadas e é indicado como elas atendem ou não aos requisitos identificados para a solução.

2.3.4 Potencial de Injetores para o Teste de Aplicações Java Multiprotocolo

A Seção 2.3.2 abordou a inadequação de ferramentas voltadas ao teste de aplicações Java baseadas somente em UDP (JACQUES-SILVA et al., 2006) (FARCHI; KRASNY; NIR, 2004) ou TCP (GERCHMAN; WEBER, 2006) para o teste de aplicações multiprotocolo. FIRMI (VACARO; WEBER, 2006) é um injetor de falhas cujas aplicações alvo são aquelas escritas em Java e baseadas no protocolo RMI, de modo que ele também não é voltado ao teste de aplicações multiprotocolo. Embora essas ferramentas não atinjam

ao objetivo principal deste trabalho, que é a injeção de falhas de comunicação em aplicações multiprotocolo, elas preenchem alguns dos outros requisitos considerados importantes para uma solução (definidos na Seção 2.3.2) e foram inspiração para a modelagem e desenvolvimento de Comform, o protótipo para injeção de falhas de comunicação desenvolvido no contexto desse trabalho. Em especial, FIRMI é útil tanto para testes de caixa preta como para testes de caixa branca.

As diferentes implementações do *framework* FIT – WS-FIT (LOOKER; MUNRO; XU, 2004), Grid-FIT (LOOKER; XU, 2007) e CROWN-FIT (TOWNEND et al., 2008) – interceptam mensagens SOAP em sistemas baseados em *Web Services*, não tratando outros tipos de protocolo. Tais abordagens trabalham somente em um alto nível de abstração, não sendo adequadas aos propósitos deste trabalho.

Como as ferramentas voltadas a aplicações construídas sobre um protocolo específico não são adequadas à injeção de falhas em aplicações multiprotocolo, caberia a um responsável por testes passar a analisar a possibilidade de uso de outros tipos de ferramentas.

NFTAPE (STOTT et al., 2000) é um *framework* para avaliação de dependabilidade em sistemas distribuídos usando injeção de falhas. Diferentemente de outras abordagens, ele faz distinção entre *injetores leves* (*lightweight fault injectors* ou LFI) – componentes responsáveis pela injeção da falha – e *gatilhos* (*triggers*) – responsáveis por disparar a injeção de falhas. Para testar aplicações Java multiprotocolo com NFTAPE, seria necessário projetar e implementar corretamente estes componentes, já que não há relatos sobre sua existência na literatura. Em outras palavras, a abordagem de NFTAPE não provê uma solução para injeção de falhas de comunicação em aplicações multiprotocolo e um grande esforço seria necessário para criar componentes adequados.

FAIL-FCI (HOARAU; TIXEUIL; VAUCHELLES, 2007) é um injetor de falhas para aplicações distribuídas. Ele pode ser usado para injeção de falhas em aplicações tanto de modo quantitativo, como qualitativo, e não requer a modificação do código fonte da aplicação alvo. FAIL-FCI tem potencial para o teste de aplicações Java multiprotocolo, mas com a forte desvantagem de que somente é capaz de emular colapsos de processos e suspensão de processos. Em geral, é muito importante testar aplicações de rede com mais tipos de falhas, como colapsos de nodos, colapsos de *links* e falhas de omissão.

DOCTOR (HAN; SHIN; ROSENBERG, 1995), apesar de injetar falhas de comunicação sem limitação a um protocolo específico, é uma ferramenta voltada a um sistema alvo específico, que já se encontra obsoleto. Assim, é inadequada ao teste de aplicações Java multiprotocolo.

Loki (CHANDRA et al., 2004) é um injetor de falhas para sistemas distribuídos que leva em consideração o estado global do sistema para injeção de falhas. Ele tem potencial para o teste de aplicações multiprotocolo, mas leva a diversas dificuldades. Loki requer a modificação do código fonte da aplicação alvo. Ainda, foi implementado em C/C++ e não pode testar diretamente aplicações Java.

FIRMAMENT (DREBES, 2005) é uma ferramenta de injeção de falhas cujo propósito principal é testar protocolos baseados em IP. Ele é implementado como um módulo do núcleo Linux, de modo que somente sistemas que podem ser executados nesse ambiente podem ser testados com a ferramenta. FIRMAMENT intercepta o envio e recebimento de pacotes IP em um nodo. Cargas de falhas (*faultloads*), especificadas usando uma linguagem de *bytecode*, são capazes de alterar conteúdo de pacotes e retornar a ação final a ser realizada sobre eles. FIRMAMENT pode ser usado para o teste de aplicações Java multiprotocolo, mas somente teste de caixa preta é viável, pois é muito difícil construir *faultloads* nos quais falhas são ativadas em pontos específicos da execução de aplicações,

já que o injetor é “cego” à semântica da aplicação alvo. Mesmo para teste de caixa preta de aplicações, pode ser muito difícil escrever *faultloads*, principalmente para aplicações baseadas em protocolos de mais alto nível, como RMI (VACARO; WEBER, 2006). Um problema mais geral consiste em que, já que todas as mensagens enviadas ou recebidas por um nodo são interceptadas, a seleção daquelas específicas de um determinado processo requer um grande esforço para codificação de *faultloads*. Ferramentas como FIONA e FIRMI, apesar de não serem ferramentas multiprotocolo, visam ao teste de *aplicações* Java e superam muitas das dificuldades impostas por ferramentas que visam ao teste de *protocolos*.

ORCHESTRA (DAWSON et al., 1996), assim como FIRMAMENT, tem como objetivo principal o teste de protocolos. Além disso, somente possui implementações para sistemas operacionais obsoletos. Portanto, tampouco é apropriada para o teste de aplicações Java multiprotocolo.

Há outros exemplos de ferramentas ou *frameworks* de injeção de falhas capazes de injetar falhas de comunicação. Entretanto, a pesquisa realizada na literatura não revelou uma solução apropriada aos propósitos deste trabalho. Os Capítulos 4 e 5 apresentam a solução proposta nesse trabalho para injeção de falhas de comunicação em aplicações Java multiprotocolo. A solução procura cumprir os requisitos identificados na Seção 2.3.2, superando deficiências das ferramentas analisadas.

2.4 Comparação de Comform a Trabalhos Relacionados

A Tabela 2.1 compara, sumariamente, Comform a Loki (CHANDRA et al., 2004), FAIL-FCI (HOARAU; TIXEUIL; VAUCHELLES, 2007) e FIRMAMENT (DREBES, 2005). Contrariamente à abordagem de Comform, estas últimas três ferramentas não cumprem todos os requisitos identificados para a solução. Apesar disso, diferentemente das outras ferramentas apresentadas na Seção 2.2, elas podem potencialmente testar aplicações multiprotocolo. Para as células sobre as quais não foi possível chegar a uma conclusão, foi atribuído o valor “?”.

Tabela 2.1: Comparando Comform a trabalhos relacionados.

	Comform	FAIL-FCI	Loki	FIRMAMENT
Capaz de injetar falhas em aplicações Java	sim	sim	não	sim
Vários tipos de falhas de comunicação	sim	não	?	sim
Preserva código fonte inalterado	sim	sim	não	sim
Teste de caixa branca de aplicações	sim	sim	sim	não
Teste de caixa preta de aplicações	sim	sim	não	sim
Descrição facilitada de faultloads	sim	?	?	não
Propriedades de portabilidade	sim	?	?	não

Loki, FAIL-FCI e FIRMAMENT não visam injetar falhas em aplicações Java multiprotocolo. Deve-se notar, entretanto, que elas constituem projetos maduros e são adequadas aos propósitos para os quais foram projetadas.

2.5 Conclusões do Capítulo

Este Capítulo apresentou conceitos básicos de injeção de falhas e ferramentas e *frameworks* de injeção de falhas capazes de injetar falhas de comunicação. A seguir, tratou da injeção de falhas em aplicações multiprotocolo, mostrando exemplos de aplicações Java multiprotocolo, apresentando o problema de injeção de falhas em aplicações multiprotocolo e identificando os requisitos importantes para uma ferramenta de injeção de falhas voltada a aplicações Java multiprotocolo.

O Capítulo também mostrou a análise do potencial das ferramentas e *frameworks* de injeção de falhas apresentados anteriormente para o teste de aplicações Java multiprotocolo. Esta análise mostrou sua inadequação para esse problema e evidenciou a necessidade de construção de uma nova ferramenta de injeção de falhas de comunicação voltada especificamente ao teste deste tipo de aplicação.

Por fim, foi apresentada uma comparação de Comform, o protótipo para injeção de falhas em aplicações Java multiprotocolo desenvolvido no contexto desse trabalho, a trabalhos relacionados. Os critérios utilizados para comparação foram os requisitos identificados para a solução na Seção 2.3.3.

As maiores contribuições do Capítulo foram a definição de requisitos para uma ferramenta de injeção de falhas voltada à avaliação de aplicações Java multiprotocolo e a análise do potencial de ferramentas e *frameworks* existentes para a injeção de falhas em aplicações Java multiprotocolo, que levou à conclusão sobre a necessidade de uma nova ferramenta.

O próximo Capítulo apresenta as implementações dos protocolos UDP, TPC e RMI em Java, com o propósito de fornecer embasamento para futura compreensão de alguns aspectos da modelagem e arquitetura de Comform como a instrumentação de classes.

3 UDP, TCP E RMI EM JAVA

Este Capítulo introduz os protocolos UDP, TCP e RMI no contexto das implementações destes protocolos em Java, focando na versão 6 da plataforma. O estudo direcionado das APIs de Java para comunicação foi importante para a construção do injetor de falhas Comform. Assim, o propósito deste Capítulo é fornecer um embasamento para a futura compreensão de alguns aspectos da modelagem e arquitetura de Comform. O enfoque do Capítulo é dado aos aspectos de maior interesse ao escopo deste trabalho, como criação de *sockets*, estabelecimento de conexões e envio e recebimento de dados. Vários aspectos relativos às APIs, apesar de importantes, não são tratados por estarem fora do escopo do trabalho. O leitor que possui conhecimento sobre essas APIs e o funcionamento dos protocolos em questão pode evitar a leitura desse Capítulo sem maiores prejuízos à compreensão do restante do texto.

Os principais protocolos de transporte da arquitetura TCP/IP são os protocolos UDP e TCP. O protocolo UDP, descrito na RFC 768 (POSTEL, 1980), não garante entrega confiável, isto é, pacotes podem ser perdidos e entregues fora de ordem. Por outro lado, UDP provê um desempenho superior ao protocolo TCP. O protocolo TCP garante transferência confiável de pacotes, uma vez que uma conexão tenha sido estabelecida, usando uma técnica conhecida como confirmação positiva com retransmissão. Ele está descrito na RFC 793 (POSTEL, 1981).

O pacote `java.net` (SUN MICROSYSTEMS, 2008), disponível desde a versão 1.1 de Java, provê classes para a implementação de aplicações de rede usando os protocolos TCP e UDP. A entrada e saída neste pacote é realizada em modo bloqueante. A partir da versão 1.4 de Java, foi introduzida uma nova API para implementação de aplicações de rede, como parte do pacote `java.nio` (SUN MICROSYSTEMS, 2008). Esta API, localizada mais especificamente no pacote `java.nio.channels`, também possui classes voltadas à comunicação usando os protocolos TCP e UDP, sendo alternativa à API de `java.net`, e não uma substituta. O conceito de *canal* (*channel*) foi introduzido com esse pacote, como uma nova abstração para entrada e saída. Canais representam conexões a entidades, como arquivos e *sockets*, capazes de realizar operações de entrada e saída. Os canais podem ser usados em modo bloqueante e não-bloqueante. Antes da versão 1.4 de Java, quando o pacote `java.nio` ainda não existia, o uso de *threads* era a única alternativa para permitir concorrência. Porém, servidores que usam o modo bloqueante acabam tendo uma relação de praticamente uma *thread* para cada cliente, o que pode levar a problemas de desempenho e de falta de escalabilidade.

Quando a comunicação bloqueante do pacote `java.nio` é utilizada, o comportamento de um canal é semelhante ao de um *socket* do pacote `java.net`. Porém, quando a comunicação não-bloqueante é utilizada, os *canais* devem trabalhar em conjunto com a classe `Selector` para a realização de multiplexação de entrada e saída. *Selectors* são

elementos que dizem qual dentre um conjunto de canais tem eventos de entrada e saída e escalonam os eventos a seus respectivos tratadores. Além dos conceitos de *canal* (*channel*) e *Selector*, outros conceitos importantes no contexto deste pacote são os de *buffers* e *selection keys*. *Buffers* são *arrays* que podem ser diretamente lidos ou escritos por *canais*. Por fim, *Selection Keys* representam o registro de um *canal* a um `Selector`.

Java RMI (SUN MICROSYSTEMS, 2006a) abstrai a interface de comunicação ao nível de uma invocação de método, evitando que o programador seja obrigado a definir um protocolo de aplicação. O pacote `java.rmi` (SUN MICROSYSTEMS, 2008) contém a API para implementação de aplicações de rede baseadas neste protocolo.

A Seção 3.1 trata das implementações do protocolo UDP nos pacotes `java.net` e `java.nio` e a Seção 3.2 trata das implementações do protocolo TCP nestes pacotes. Por fim, a Seção 3.3 trata do protocolo RMI, enfocando em aspectos de sua arquitetura.

3.1 UDP

A Seção 3.1.1 trata da implementação de UDP no pacote `java.net` (SUN MICROSYSTEMS, 2008). Em seguida, a Seção 3.1.2 trata da implementação de UDP no pacote `java.nio` (SUN MICROSYSTEMS, 2008).

3.1.1 UDP no pacote java.net

Considerando a implementação de UDP no pacote `java.net`, a comunicação utilizando esse protocolo compreende os seguintes passos:

- Criação de um *socket*, implementado pela classe `DatagramSocket`. Ao contrário de TCP, *sockets* UDP cliente e servidor são representados por uma mesma classe e não há necessidade de estabelecimento de conexão para troca de mensagens.

- Criação de um pacote, representado pela classe `DatagramPacket`.

- Envio de pacotes UDP por meio do método `send(DatagramPacket)` e recebimento de pacotes UDP por meio do método `receive(DatagramPacket)`, ambos da classe `DatagramSocket`. Caso o destinatário de um pacote não esteja aguardando recebimentos, o pacote é perdido.

Um `DatagramSocket` pode ou não ser conectado a um endereço remoto. Se estiver conectado, ele somente pode receber e enviar pacotes de/para o endereço remoto.

São cinco os construtores de `DatagramSocket` suportados para a criação de uma instância dessa classe. Alguns deles tratam da associação (*binding*) do *socket* a um endereço e porta locais invocando o método `bind(SocketAddress addr)`. Outros, que não tratam desta associação, requerem que o programador invoque o método `bind(SocketAddress addr)` diretamente.

Um `DatagramPacket` contém um *buffer* de *bytes* com os dados (atributo *byte buffer*), o tamanho dos dados (atributo *length*), uma indicação de qual parte do *buffer* é relevante (atributo *offset*) e um endereço de *socket* composto por endereço IP e porta (atributo *address*). O endereço de *socket* corresponde ao destinatário do pacote. Ainda, adicionalmente a esses dados, um pacote contém implicitamente o endereço de sua origem.

O método `send(DatagramPacket)` envia um pacote ao endereço especificado no atributo *address*. O método `receive(DatagramPacket)` lê um pacote recebido e preenche um `DatagramPacket` com os dados e o endereço recebidos. Este método

fica bloqueado até que um pacote chegue ou até que um determinado *timeout*, que pode ser definido pelo programador, seja atingido.

O pacote `java.net` também conta com a classe `MulticastSocket`, que visa à comunicação utilizando IP *multicasting*. A classe `MulticastSocket` estende a classe `DatagramSocket` com habilidades adicionais para a realização de entrada (*joining*) em grupos *multicast*. Quanto ao envio e recebimento de pacotes UDP, os métodos `send(DatagramPacket)` e `receive(DatagramPacket)`, herdados da classe mãe, são utilizados.

3.1.2 UDP no pacote java.nio

Considerando a implementação de UDP no pacote `java.nio`, a comunicação utilizando esse protocolo compreende os seguintes passos:

- Abertura de um `DatagramChannel`. Ao contrário de TCP, *canais* UDP cliente e servidor são representados por uma mesma classe e não há necessidade de estabelecimento de conexão para troca de mensagens.

- Definição sobre a comunicação ser bloqueante ou não-bloqueante por meio do método `configureBlocking(boolean)`. Caso seja definida comunicação não-bloqueante, `DatagramChannel` deve trabalhar em conjunto com a classe `Selector` para a realização de multiplexação de entrada e saída.

- Criação de *buffers* para leitura ou escrita de dados.

- Envio e recimento de pacotes UDP fazendo uso dos métodos `send(ByteBuffer src, SocketAddress target)` e `receive(ByteBuffer dst)` ou por meio de métodos `write()` e `read()`, todos de `DatagramChannel`. Caso o destinatário de um pacote não esteja aguardando mensagens, ele é perdido.

A abertura de um `DatagramChannel` é feita pela invocação do método estático `open()`.

O método `send(ByteBuffer src, SocketAddress target)` é responsável pelo envio de datagramas no canal onde é invocado. Os dados de um *buffer* (*src*) são enviados ao endereço de destino (`target`). Já o método `receive(ByteBuffer dst)` é responsável pelo recebimento de um datagrama (`dst`) a partir do canal onde é invocado. Ainda, existem três diferentes assinaturas para métodos `write()`, sendo que todos eles escrevem um datagrama, passado como parâmetro por meio de um *buffer* ou um *array* de *buffers*, no canal sobre o qual são invocados. Existem também três diferentes assinaturas para métodos `read()`, sendo que todos eles lêem um datagrama a partir do canal sobre o qual são invocados e preenchem um *buffer* ou um *array* de *buffers* com os dados lidos.

O pacote `java.nio`, na versão 6 de Java, ainda não provê suporte à IP *multicasting*. Porém, a versão 7 (SUN MICROSYSTEMS, 2009), que ainda estava em versão preliminar (*draft*) durante a escrita dessa dissertação, deve prover esse suporte.

3.2 TCP

A Seção 3.2.1 trata da implementação de TCP no pacote `java.net` (SUN MICROSYSTEMS, 2008). A seguir, a Seção 3.2.2 trata da implementação de TCP no pacote `java.nio` (SUN MICROSYSTEMS, 2008).

3.2.1 TCP no pacote java.net

Considerando a implementação de TCP no pacote `java.net`, o estabelecimento de uma conexão TCP compreende os seguintes passos:

- Criação de um *socket*, sendo que a classe `Socket` implementa *sockets* clientes, enquanto a classe `ServerSocket` implementa *sockets* servidores.

- Abertura de conexão, sendo a abertura ativa feita pelos métodos `connect()` de `Socket` e a abertura passiva feita pelo método `accept()` de `ServerSocket`.

- Criação de objetos *stream* (*input stream* para recebimento de dados e *output stream* para envio de dados).

- Envio e recebimento de dados através dos objetos *stream*.

São nove os construtores de `Socket` suportados para a criação de uma instância dessa classe. Cada construtor recebe diferentes parâmetros de entrada, sendo que boa parte deles têm *host* e porta do destino entre os parâmetros. Quanto à abertura ativa de uma conexão, o método `connect(SocketAddress endpoint)` conecta o *socket* ao servidor passado como parâmetro, enquanto o método `connect(SocketAddress endpoint, int timeout)`, alternativo a ele, faz o mesmo, mas considerando um *timeout*, passado como parâmetro. A conexão não precisa ser realizada diretamente pelo programador nos casos em que são usados construtores que recebem *host* e porta destino como parâmetros, pois o método construtor invoca automaticamente um método `connect()`. Nestes casos, se a conexão for estabelecida sem problemas, a invocação do método construtor já retorna um *socket* conectado. Nos demais casos, é necessário que o programador invoque um método `connect()`. Constatou-se que é possível que, no momento de invocação de um método `connect()`, o *socket* cliente ainda não esteja associado a uma porta local. Nestes casos, o núcleo do sistema operacional irá escolher uma porta efêmera para esse propósito.

No caso de `ServerSocket`, são quatro os construtores suportados para a criação de uma instância dessa classe. Cada construtor recebe diferentes parâmetros de entrada, sendo que três deles têm entre seus parâmetros o número de uma porta local ao qual o *socket* servidor deve se associar. Um *socket* servidor deve se associar (realizar *binding*) a uma porta local e aguardar requisições de conexão. O *binding* pode ser realizado tanto no escopo de um método construtor, nos casos em que o construtor recebe porta local como parâmetro, como pela invocação de um método `bind()` pelo programador. A função de um *socket* servidor é realizar uma abertura passiva de conexão, aguardando conexões. Para aguardar o recebimento de conexões, o método `accept()` deve ser invocado. Este método fica bloqueado até que chegue uma requisição de conexão ou até que um determinado *timeout*, que pode ser definido pelo programador, seja atingido. Quando uma conexão é aceita e estabelecida, uma instância de `Socket` é criada, representando-a. Analisando a implementação do método `accept()`, percebe-se que ele invoca o método `implAccept()`, que, por fim, interage com as funções nativas.

A entrada e saída em Java é tratada através de objetos *stream*. No caso de *sockets*, as classes `SocketInputStream` e `SocketOutputStream` são responsáveis, respectivamente, pelo recebimento e envio de dados. Os objetos *stream* de entrada e saída são obtidos de um *socket* por meio da invocação, respectivamente, dos métodos `getInputStream()` e `getOutputStream()`. Objetos `SocketInputStream` e

`SocketOutputStream`, em geral, não são manipulados diretamente pelo programador. Visando permitir a criação de estruturas complexas para o processamento de entrada e saída, Java possibilita o "encadeamento" de *streams*, ou seja, uma instância de um *stream* pode ser passada como parâmetro ao construtor de outro. Este tópico foge do escopo deste trabalho e maiores detalhes podem ser encontrados em livros sobre computação distribuída em Java (REILLY; REILLY, 2002) . O mais importante, no contexto deste trabalho, é que constatou-se que objetos das classes `SocketInputStream` e `SocketOutputStream` serão sempre utilizados para o envio e recebimento de dados usando TCP no pacote `java.net`. São estes os objetos que interagem com as funções nativas responsáveis por realizar efetivamente a comunicação, ou seja, estão no nível mais baixo da JVM antes que as funções nativas sejam chamadas. Os principais métodos das classes `SocketInputStream` e `SocketOutputStream` são, respectivamente, os métodos `read()` e `write()`. Eles são os responsáveis pelo envio e recebimento de dados entre participantes de uma comunicação usando TCP. Existem três diferentes assinaturas pra os métodos `read()` e três diferentes assinaturas para os métodos `write()`. As três implementações do método `write()` invocam o método `socketWrite(byte b[], int off, int len)`, que interage com a função nativa responsável pelo envio de mensagens.

3.2.2 TPC no pacote java.nio

Considerando a implementação de TCP no pacote `java.nio`, o estabelecimento de uma conexão TCP compreende os seguintes passos:

- Abertura de um `ServerSocketChannel` no qual aceitar conexões e de um `SocketChannel` para estabelecer conexões.
- Definição sobre a comunicação ser bloqueante ou não-bloqueante usando o método `configureBlocking(boolean)`. Em caso de comunicação não-bloqueante, `ServerSocketChannel` e `SocketChannel` devem trabalhar em conjunto com a classe `Selector` para a realização de multiplexação de entrada e saída.
- Abertura de conexão, sendo que a abertura ativa é feita pelo método `connect()` de `SocketChannel`, enquanto a abertura passiva é feita pelo método `accept()` de `ServerSocketChannel`.
- Criação de *buffers* para leitura ou escrita de dados.
- Envio e recebimento de dados, respectivamente, por meio de métodos `write()` e `read()` de `SocketChannel`.

A abertura de um `ServerSocketChannel` é feita pela invocação do método estático `open()`.

Já a abertura de um `SocketChannel` pode ser feita pela invocação dos métodos estáticos `open()` ou `open(SocketAddress remote)`. No primeiro caso, para estabelecer conexão, o programador deverá invocar, posteriormente ao método `open()`, o método `connect(SocketAddress remote)`. Já no segundo caso, o próprio método trata também da conexão com o endereço remoto. Se a conexão for estabelecida sem problemas, a invocação daquele método já retorna um objeto `SocketChannel` conectado.

Quanto ao envio e recebimento de dados, existem três diferentes assinaturas para métodos `write()`, sendo que todos eles escrevem uma sequência de *bytes*, passada como

parâmetro através de um *buffer* ou *array* de *buffers*, no canal sobre o qual são invocados. Existem também três diferentes assinaturas para métodos `read()`, sendo que todos eles lêem uma sequência de *bytes* a partir do canal sobre o qual são invocados e preenchem um *buffer* ou *array* de *buffers* com os dados lidos.

3.3 RMI

O sistema de invocação remota de métodos da linguagem de programação Java (SUN MICROSYSTEMS, 2006a) abstrai a interface de comunicação ao nível de uma invocação de método, evitando que o programador seja obrigado a definir um protocolo de aplicação. O sistema foi desenvolvido para uso no ambiente homogêneo da JVM, e pode, portanto, aproveitar as vantagens do modelo de objetos da plataforma Java sempre que possível.

No modelo de objetos distribuídos da plataforma Java, um objeto remoto é aquele cujos métodos podem ser invocados de outra JVM, potencialmente em um *host* diferente. Um objeto deste tipo é descrito por uma ou mais interfaces remotas, escritas na linguagem de programação Java, que declaram os métodos do objeto remoto. Para que um objeto possa ser invocado remotamente, ele deve implementar a interface `java.rmi.Remote` e a declaração de cada método deve conter a exceção `java.rmi.RemoteException` na cláusula `throws`.

A invocação remota de métodos (RMI) é o ato de invocar um método de uma interface remota em um objeto remoto. A sintaxe de invocação de métodos em objetos remotos é a mesma que a utilizada para invocação de métodos em objetos locais.

RMI não é apenas uma implementação de protocolo desenvolvida pela Sun, mas também uma especificação (SUN MICROSYSTEMS, 2006a) que define os aspectos necessários para a comunicação baseada em objetos remotos. Existem outras implementações do protocolo, além da desenvolvida pela Sun, mas esta é o foco deste trabalho.

As aplicações construídas sobre Java RMI são frequentemente compostas por dois programas distintos: um servidor e um cliente. Um servidor típico cria objetos remotos, torna acessíveis referências a estes objetos remotos e espera que clientes invoquem métodos nestes objetos. Um cliente típico obtém uma referência remota para um ou mais objetos remotos no servidor e então invoca métodos nestes objetos. Em geral, a referência a um objeto remoto é obtida no registro de objetos RMI – `rmiregistry`.

A Figura 3.1 ilustra um exemplo de aplicação distribuída construída sobre Java RMI que usa o `rmiregistry` para obter referências a um objeto remoto. O servidor cria objetos remotos, exporta esses objetos e chama o `rmiregistry` para os registrar. Na exportação de um objeto, é alocado um *socket* (endereço e porta) para o estabelecimento de conexões, transformando um objeto comum em um objeto remoto. Já o registro corresponde à associação de um nome a um objeto remoto. O cliente procura o objeto remoto pelo seu nome no `rmiregistry` do servidor e então invoca um método neste objeto. O `rmiregistry` pode estar localizado tanto na mesma máquina que o servidor quanto em outra máquina.

Na comunicação por invocação remota de métodos, objetos podem ser passados como parâmetros e retornados como resultado de uma execução. RMI provê os mecanismos necessários para carregar os *bytecodes* das classes de um objeto, assim como para transmitir os seus dados. A Figura 3.1 mostra que o sistema RMI usa um servidor *web* para carregar *bytecodes* de classes escritas em Java, para objetos, do servidor para o cliente e do cliente para o servidor, quando uma classe não está definida em uma JVM. RMI pode carregar *bytecodes* de classes usando qualquer protocolo URL (por exemplo, HTTP, FTP, etc.) que

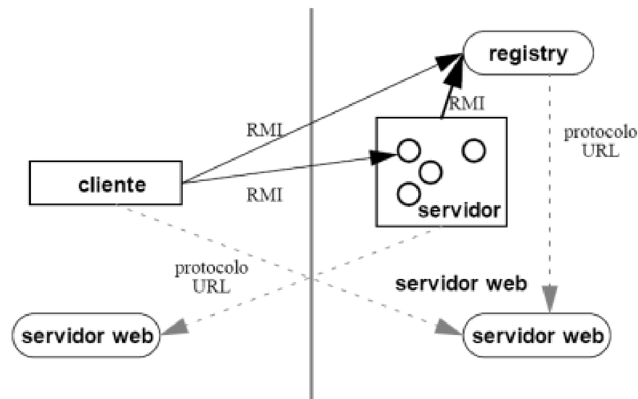


Figura 3.1: Exemplo de aplicação distribuída baseada em RMI (SUN MICROSYSTEMS, 2006a).

é suportado pela plataforma Java.

RMI utiliza *stubs* e *skeletons* como mecanismo de comunicação padrão entre objetos remotos (SUN MICROSYSTEMS, 2006a). Um *stub* para um objeto remoto atua como um representante local ou *proxy* do objeto remoto no cliente, implementando o mesmo conjunto de interfaces remotas que um objeto remoto implementa (SUN MICROSYSTEMS, 2006a).

No mecanismo de invocação de métodos, o requisitante invoca um método em um *stub* local, o qual é responsável por realizar a chamada de método no objeto remoto. O *stub* esconde a serialização de parâmetros e a comunicação ao nível de rede a fim de apresentar um mecanismo simples de invocação ao requisitante. Quando um método de um *stub* é invocado, o *stub* realiza as seguintes atividades (SUN MICROSYSTEMS, 2006a):

- inicia uma conexão com a JVM remota que contém o objeto remoto;
- escreve e transmite os parâmetros para a JVM remota;
- espera pelo resultado da invocação do método;
- lê o valor de retorno ou exceção retornada;
- retorna o valor ao requisitante.

Cada objeto remoto pode ter um *skeleton* correspondente na JVM remota, o qual é responsável por entregar a chamada para a real implementação do objeto remoto. Quando um *skeleton* recebe uma invocação de método, ele realiza as seguintes atividades (SUN MICROSYSTEMS, 2006a):

- lê os parâmetros do objeto remoto;
- invoca o método na implementação real do objeto;
- escreve e transmite o resultado (valor de retorno ou exceção) ao requisitante.

A partir do JDK 1.2, foi eliminada a necessidade de *skeletons* devido à introdução de um protocolo *stub* adicional e as atividades do *skeleton* passaram a poder ser realizadas por código genérico.

3.4 Conclusões do Capítulo

Este Capítulo introduziu os protocolos de comunicação UDP, TCP e RMI no contexto de sua arquitetura e implementação na linguagem Java, focando na versão 6 da plataforma. Sua principal contribuição foi sintetizar o funcionamento desses protocolos, enfocando os aspectos de maior interesse ao escopo deste trabalho, como criação de *sockets*, estabelecimento de conexões e envio e recebimento de dados. Tal conhecimento é útil para a compreensão de alguns aspectos da modelagem e arquitetura de Comform que serão apresentados posteriormente.

O próximo Capítulo apresenta questões de projeto importantes para o desenvolvimento de uma solução para injeção de falhas de comunicação em aplicações multiprotocolo de acordo com os requisitos identificados no Capítulo 2.

4 QUESTÕES DE PROJETO

Para injetar falhas de comunicação corretamente em aplicações Java multiprotocolo, é necessário um modo de tratar adequadamente todos os protocolos usados por elas. Além disso, outros requisitos foram identificados para a solução (apresentados na Seção 2.3.3) e devem ser levados em consideração. Constituem tais requisitos a emulação de vários tipos de falhas de comunicação, a preservação do código fonte da aplicação alvo, a utilidade da ferramenta tanto para testes de caixa branca como para testes de caixa preta, a facilidade para descrição de cenários de falhas (*faultloads*) e portabilidade.

Limitando o escopo da solução, optou-se por tratar aplicações baseadas nos protocolos UDP, TCP e RMI. UDP e TCP são os protocolos de transporte mais comuns da arquitetura TCP/IP. RMI é um protocolo de mais alto nível amplamente utilizado em aplicações Java.

Este Capítulo está organizado como segue. A Seção 4.1 apresenta um modelo de solução para injeção de falhas de comunicação em aplicações multiprotocolo e a Seção 4.2 discute possibilidades de abordagens a serem empregadas no projeto de um injetor de falhas para aplicações Java.

4.1 Modelo Genérico

Esta Seção apresenta um modelo genérico de solução para injeção de falhas de comunicação em aplicações multiprotocolo. Na Seção 4.1.1, é tratada a modelagem de falhas. Na Seção 4.1.2, são abordados cuidados que devem ser tomados na emulação de falhas de colapso em aplicações baseadas em TCP. Por fim, na Seção 4.1.3 é tratado como deve ser realizada a seleção e manipulação de mensagens.

4.1.1 Modelagem de Falhas

A solução deve ser capaz de emular os tipos de falhas de comunicação que são comuns em ambientes de rede, que foram tratados na Seção 2.1.3, como colapso de nodos, colapso de *links*, falhas de temporização, falhas de omissão, falhas de particionamento de redes e falhas bizantinas, como corrupção de mensagens.

O descarte de todas as mensagens de um nodo antes que elas sejam entregues à aplicação alvo e antes que elas sejam enviadas para outros nodos é usado para emulação de colapso de nodos. Apenas “matar” processos não é uma estratégia adequada para emular esse tipo de falha. Essa estratégia só é útil para a emulação de colapsos de processo, isto é, término de processos. Quando um colapso de nodo ocorre, um nodo permanentemente pára de enviar e receber mensagens. Considerando, por exemplo, uma aplicação servidora baseada em TCP e um colapso de nodo ocorrendo na máquina onde ela está sendo executada, o módulo TCP local não envia qualquer mensagem a outros nodos indicando esta

condição. Este comportamento é distinto do término de processos, no qual mensagens relacionadas à finalização da conexão são trocadas entre módulos TCP.

Colapsos de *links* tem comportamento similar a colapsos de nodos, mas os estados dos *links* podem variar no tempo, de modo que às vezes o *link* esteja ativo e às vezes inativo. Estes comportamentos são modelados usando taxas de falhas e distribuições de probabilidade. Falhas de omissão podem ser divididas em falhas de omissão de envio e falhas de omissão de recepção. Falhas de omissão de envio são caracterizadas pela perda de mensagens depois que elas são enviadas pela aplicação, mas antes que elas sejam efetivamente enviadas para a rede. Falhas de omissão de recepção são similares a falhas de omissão de envio, mas elas se manifestam no lado destinatário. Assim como colapsos de *link*, falhas de omissão (tanto de envio como de recepção) podem ser emuladas pelo descarte de mensagens de acordo com taxas de falhas e distribuições de probabilidades. Particionamento de rede é caracterizado pela divisão de uma rede em uma ou mais subredes desconectadas. Ele pode ser representado pela emulação de vários colapsos de *links*, de modo que também é modelado pelo descarte de mensagens. Falhas de temporização representam atrasos no envio e recebimento de mensagens. Contrariamente a outros tipos de falhas descritos previamente, falhas de temporização podem ser emuladas em um nível mais alto de abstração, por meio do atraso do envio e recebimento de mensagens. Falhas bizantinas relacionadas à corrupção de mensagens também podem ser emuladas em um alto nível de abstração por meio da alteração de parâmetros das mensagens interceptadas.

4.1.2 Cuidados para Emulação de Colapso em Aplicações TCP

Protocolos de transporte, como UDP e TCP, são implementados no núcleo do sistema operacional. A emulação de falhas de comunicação em aplicações Java baseadas *somente* em UDP pode ser feita completamente no nível da JVM (JACQUES-SILVA et al., 2006) (FARCHI; KRASNY; NIR, 2004). No caso de falhas de colapso de nodo, por exemplo, essa emulação pode ser feita por meio da inibição do envio e recebimento de mensagens UDP no nível da JVM. Isso é suficiente devido à simplicidade do protocolo UDP. Por outro lado, considerando o protocolo TCP e aqueles protocolos de mais alto nível que o têm como base, falhas de comunicação que envolvem o descarte de mensagens dificilmente podem ser emuladas somente no nível da JVM. Se falhas de colapso de nodo, por exemplo, são emuladas por meio do descarte de mensagens TCP no nível da JVM, como na ferramenta FIERCE (GERCHMAN; WEBER, 2006), essas falhas são emuladas de modo inconsistente. Vacaro (2007), ao analisar a possibilidade do uso de FIERCE para a injeção de falhas em aplicações baseadas em Java RMI, mostra a inadequação da abordagem dessa ferramenta para a emulação de falhas de colapso.

A Figura 4.1 ajuda a explicar, de modo simplificado, a inadequação de emular falhas de colapso de nodo (ou outros tipos de falhas caracterizados pelo descarte de mensagens) em aplicações Java baseadas em TCP, e/ou em protocolos baseados em TCP, apenas no nível da JVM. O nodo A está executando uma aplicação Java cliente, baseada em TCP, enquanto o nodo B está executando sua correspondente aplicação servidor. Uma conexão já foi estabelecida e tenta-se injetar uma falha de colapso em B por meio do descarte de mensagens TCP no nível da JVM de B. Pode-se observar o cliente enviando mensagens TCP ao servidor. Embora essas mensagens sejam descartadas no nível da JVM do servidor, o seu sistema operacional (SO) ainda irá enviar ao cliente confirmações relacionadas ao recebimento das mensagens, de modo que a falha não é emulada corretamente.

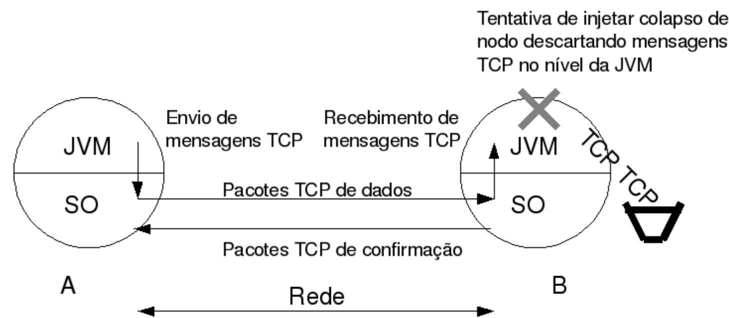


Figura 4.1: Emulação inconsistente de falhas de colapso no nível da JVM para aplicações Java baseadas em TCP.

4.1.3 Seleção e Manipulação de Mensagens

Um modelo genérico de solução para injeção de falhas de comunicação em aplicações multiprotocolo que possam ser baseadas no protocolo TCP e/ou em protocolos de mais alto nível baseados em TCP deve levar em consideração as observações apresentadas na Seção 4.1.2. Para que falhas que envolvem o descarte de pacotes sejam emuladas com representatividade em um nodo, pacotes relativos a todos os protocolos utilizados pela aplicação alvo devem ser passíveis de seleção e manipulação antes que eles sejam entregues à aplicação alvo (no caso de recebimento de pacotes) e antes que eles sejam enviados a outros nodos (no caso de envio de pacotes). Assim, essa seleção e manipulação deve ser feita no contexto do sistema operacional. Por outro lado, ao contrário de falhas que são caracterizadas pelo descarte de pacotes, falhas de temporização podem ser injetadas pela seleção e manipulação de mensagens em um nível mais alto de abstração.

Visando atender aos requisitos identificados para a solução, o modelo também requer a interceptação de mensagens em um nível de abstração mais alto do que o nível do sistema operacional de modo que informações específicas relacionadas à aplicação alvo possam ser facilmente recuperadas para a ativação e desativação de falhas. Por exemplo, considerando uma aplicação baseada em RMI, pode ser interessante a um engenheiro de testes ter a possibilidade de realizar um teste de caixa branca em que uma falha seja ativada antes da invocação de um determinado método remoto ou depois de um certo número de invocações de métodos remotos. Este tipo de informação é muito difícil de ser obtida no nível do núcleo do sistema operacional, mas pode ser facilmente obtida em um nível mais alto de abstração.

4.2 Abordagens para Injeção de Falhas em Aplicações Java

Esta Seção discute possibilidades de abordagens a serem empregadas no projeto de um injetor de falhas de comunicação para aplicações Java. São apresentadas abordagens que já foram utilizadas no projeto de injetores de falhas encontrados na literatura.

Na Seção 4.2.1, são apresentados JVMTI (*Java Virtual Machine Tool Interface*) (SUN MICROSYSTEMS, 2006b) e o pacote `java.lang.instrument` (SUN MICROSYSTEMS, 2007), opções que provêm suporte à instrumentação e são parte da API da plataforma Java. Conforme será detalhado nesta Seção, essas opções não são capazes de realizar a manipulação de *bytecode* propriamente dita, mas são capazes de interceptar o carregamento de classes provendo o suporte à instrumentação.

Na Seção 4.2.2, a biblioteca de manipulação de *bytecode* Javassist, que não faz parte da API de Java, é apresentada. Na Seção 4.2.3, a abordagem do uso de programação orientada a aspectos para o desenvolvimento de injetores de falhas é introduzida. Por fim, na Seção 4.2.4, o uso de *firewalls* para emulação de falhas é abordado.

4.2.1 Suporte à Instrumentação em Ambientes Java

Esta Seção apresenta o uso de JVMTI e do pacote `java.lang.instrument` para propósitos de projeto de injetores de falhas. Ambas as abordagens tem o benefício de não exigirem a modificação do código fonte da aplicação alvo.

4.2.1.1 JVMTI

JVMTI (*Java Virtual Machine Tool Interface*) (SUN MICROSYSTEMS, 2006b), disponível a partir da versão 1.5 de Java, é uma interface de programação nativa cujo objetivo inicial era o uso no desenvolvimento de ferramentas de monitoramento e depuração. JVMTI provê capacidades tanto de inspecionar o estado como de controlar a execução de aplicações executadas na JVM. No desenvolvimento da ferramenta FIONA (JACQUES-SILVA, 2005), JVMTI foi utilizada como abordagem para injeção de falhas por *software* em aplicações Java.

Um cliente JVMTI, denominado *agente*, é implementado como uma biblioteca dinâmica em linguagem nativa (qualquer linguagem nativa que suporte convenções de chamada da linguagem C e definições de C ou C++) e pode ser notificado sobre ocorrências interessantes por meio de *eventos*. Exemplos de eventos incluem, entre outros, a *inicialização completa da JVM*, o *carregamento de uma classe* ou a *morte da JVM*. Os eventos de interesse devem ser registrados pelo agente para que ele seja notificado durante a execução da aplicação. Ao ser notificado, um agente pode chamar funções de tratamento para o evento, também registradas anteriormente, relacionadas à inspeção ou controle da execução de uma aplicação. Também é possível que essa inspeção e controle sejam realizados independentemente dos eventos. Agentes são executados no mesmo processo da JVM que está executando a aplicação alvo e comunicam-se diretamente com essa JVM usando JVMTI. Eles são carregados opcionalmente pela JVM durante sua inicialização.

JVMTI *permite* a manipulação de *bytecodes*, ou seja, JVMTI não faz a alteração de *bytecodes* propriamente dita, mas somente permite que ela seja realizada. Esse recurso permite o seu uso para a inserção de código de injeção de falhas em aplicações Java. A instrumentação pode ser feita estaticamente, em tempo de carga ou dinamicamente. No caso da instrumentação estática, as classes são alteradas antes de serem carregadas pela máquina virtual. Na instrumentação em tempo de carga, os `bytes` do arquivo da classe são enviados ao agente para instrumentação quando ocorre um evento de carregamento de classe. Na instrumentação dinâmica, uma classe já carregada precisa ser alterada durante a execução da aplicação. A ferramenta FIONA (JACQUES-SILVA, 2005) fornece uma versão instrumentada estaticamente do código de `java.net.DatagramSocket` para que, em tempo de carga, este código seja carregado no lugar da versão original. Essa classe é alterada de modo a interagir com classes de injeção de falhas. Para a realização de monitoramento com FIONA, eventos relevantes para monitoramento de experimentos de injeção de falhas, como a morte da máquina virtual e a captura de exceções, são notificados ao agente JVMTI.

Uma das desvantagens do uso de JVMTI como suporte à instrumentação é o fato de requerer que o agente JVMTI seja implementado em código nativo, comprometendo a portabilidade (BINDER; HULAAS; MORET, 2007) (JACQUES-SILVA, 2005). Ainda,

JVMTI pode não estar disponível em todas as implementações de JVM.

4.2.1.2 Pacote java.lang. instrument

O pacote `java.lang.instrument` (SUN MICROSYSTEMS, 2007), disponível a partir da versão 1.5 de Java, provê serviços que permitem que *agentes* da linguagem de programação Java instrumentem aplicações que estão sendo executadas na JVM.

No contexto do pacote `java.lang.instrument`, um *agente* é uma biblioteca plugável desenvolvida em Java, cujo formato é um arquivo JAR, capaz de interceptar o processo de carregamento de classes da JVM. Um agente é carregado na inicialização da JVM (pelo carregador de classes de sistema) e a interceptação realizada por ele provê acesso ao *bytecode* das classes e possibilita a sua instrumentação. Cabe ressaltar que, assim como JVMTI, o pacote `java.lang.instrument` não é capaz de fazer a manipulação de *bytecodes* propriamente dita. Ambas as abordagens apenas permitem que a instrumentação seja realizada. Para manipulação de *bytecode*, existem bibliotecas, como Javassist (CHIBA, 1998) (CHIBA, 2000), que será apresentada adiante.

Um agente deve implementar o método `premain` (similar ao método `main` de uma aplicação). Depois da inicialização da JVM, cada método `premain` será chamado na ordem que os agentes foram especificados e, só então, o método `main` da aplicação será chamado. Um agente provê uma implementação da interface *ClassFileTransformer* para transformar os *bytecodes* de uma classe antes que ela seja definida pela JVM. Essa interface é constituída pelo método *transform*, cuja implementação pode modificar o *bytecode* de uma classe, passada como parâmetro, e retornar uma versão alterada da classe, que será usada como substituta.

Na ferramenta FIRMI (VACARO, 2007), o pacote `java.lang.instrument` foi utilizado para a interceptação do carregamento de classes na JVM e consequente provimento de acesso ao *bytecode* de classes Java relacionadas ao protocolo RMI. A principal vantagem de uso do pacote como alternativa a JVMTI é que sua implementação é feita completamente em Java, favorecendo a portabilidade. Além disso, ao contrário de JVMTI, cujos focos principais são monitoramento e depuração, esse pacote é específico para instrumentação.

Devido às vantagens de sua utilização, esse pacote também foi utilizado em Comform para a interceptação do carregamento de classes na JVM. Deste modo, é obtido o acesso às classes Java relacionadas aos protocolos de interesse.

4.2.2 Javassist: Biblioteca de Manipulação de Bytecode

Javassist (Java Programming Assistant) (CHIBA, 1998) (CHIBA, 2000) é uma biblioteca de classes para a edição de *bytecodes* em Java, que permite que programas Java definam uma nova classe em tempo de execução e que modifiquem um arquivo de uma classe em tempo de carga pela JVM.

Além de Javassist, existem outros editores de *bytecode* como BCEL (FOUNDATION, 2002), Serp (WHITE, 2002) e ASM (CONSORTIUM, 1999), mas Javassist é uma opção vantajosa em relação a eles por oferecer uma API de alto nível para a manipulação de *bytecode*. Javassist também oferece uma API de baixo nível, como esses outros editores. Se os usuários escolhem usar a API de alto nível, eles podem editar um arquivo de classe sem conhecer as especificações de *bytecode* de Java, trabalhando apenas no nível do código fonte usando vocabulário da linguagem Java. Tais alterações são compiladas por Javassist em tempo de execução. Já com a API de baixo nível, os usuários podem alterar diretamente uma classe, da mesma forma que os outros editores.

Javassist foi utilizado nas ferramentas Jaca (MARTINS; RUBIRA; LEME, 2002) e FIRMI (VACARO, 2007) anteriormente. No caso da arquitetura de Jaca, Javassist é utilizado como plataforma para *reflexão computacional*, separando claramente a instrumentação necessária para injeção de falhas e monitoramento, dos requisitos funcionais do sistema. Uma extensão de Jaca para injetar falhas de comunicação em aplicações Java baseadas em UDP foi desenvolvida (JACQUES-SILVA et al., 2004). Porém, essa extensão de Jaca exige a modificação do código fonte da aplicação alvo, já que, devido ao mecanismo de carregamento de classes em Java, não é possível modificar classes de sistema (CHIBA, 2007), como `java.net.DatagramSocket`, em tempo de execução. No caso de FIRMI, o pacote `java.lang.instrument`, que é capaz de interceptar o carregamento de classes de sistema, é utilizado para a interceptação de classes, de modo que esse problema é superado. Em FIRMI, Javassist é utilizado somente para manipulação de *bytecode*.

Devido às vantagens de sua utilização, Javassist também foi utilizado em Comform juntamente com o pacote `java.lang.instrument`, como será tratado no próximo Capítulo.

4.2.3 Programação Orientada a Aspectos

A Programação Orientada a Aspectos (POA) (KICKZALES et al., 1997) visa à modularização de *interesses transversais* (*crosscutting concerns*), isto é, comportamentos que afetam diversos módulos de um sistema, em unidades chamadas *aspectos*. O código de interesses transversais geralmente encontra-se espalhado em diversos módulos de uma aplicação, dificultando sua manutenção e compreensão.

POA introduz alguns conceitos à Programação Orientada a Objetos, como *pontos de junção* (*joint points*), *pontos de corte* (*pointcuts*), *adendos* (*advices*) e *aspectos* (*aspects*). Um ponto de junção é um ponto bem definido no fluxo de execução de um programa, como uma chamada de método. Um ponto de corte separa alguns pontos de junção e alguns valores nesses pontos de junção. Um adendo é um código que é executado quando um certo ponto de junção é atingido. Um aspecto é uma unidade modular que atravessa a estrutura de outras unidades. Ele é similar a uma classe tendo um tipo, estendendo outras classes e outros aspectos.

Na ferramenta FICTA (SILVEIRA, 2005) (SILVEIRA; WEBER, 2006), é explorada uma estratégia baseada em POA para injeção de falhas de comunicação em aplicações Java baseadas em UDP. A estratégia utiliza os recursos de POA para realização de interceptação e instrumentação das primitivas de envio e recepção de pacotes UDP, de modo que o código fonte da aplicação alvo não é alterado.

Um injetor de falhas pode ser visto como um interesse transversal da aplicação alvo e cada tipo de falhas pode ser visto como um aspecto. O conceito de adendo simplifica a inserção de código instrumentado na aplicação. O código do adendo é executado quando um ponto de junção é capturado por algum ponto de corte. A estratégia utilizada para o desenvolvimento de FICTA foi a implementação de aspectos que definem pontos de corte em termos de primitivas de comunicação do protocolo UDP e implementam adendos que substituem os códigos originais. Os aspectos são criados e integrados (processo de *weaving*) às classes que devem ser afetadas por eles em tempo de carga. A instrumentação de primitivas de comunicação *send* e *receive* é realizada efetivamente em tempo de execução com a inclusão de interfaces e chamadas a adendos que implementam lógica adicional.

Segundo Silveira (2005) (2006), o uso de Programação Orientada a Aspectos na construção de FICTA resultou em uma ferramenta altamente modular, reusável e flexível, que

não causa intrusividade espacial no código fonte da aplicação (não modifica o código da aplicação alvo). Por outro lado, em relação à intrusividade temporal, é esperado que FICTA tenha desempenho inferior a FIONA, por residir em um nível muito mais alto de abstração (SILVEIRA; WEBER, 2006).

4.2.4 Uso de Firewalls para Emulação de Falhas

Conforme apresentado na Seção 4.1, um modelo genérico para injeção de falhas de comunicação em aplicações multiprotocolo inclui a seleção e manipulação de pacotes antes que eles sejam entregues à aplicação alvo e antes que eles sejam enviadas a outros nodos. Nesse sentido, o uso de um *firewall* possibilita que falhas que envolvem o descarte de mensagens sejam corretamente emuladas.

Um *firewall*, como o IPTables (RUSSEL; WELTE, 2002), controla as regras de filtragem de pacotes, inserindo e apagando regras em uma tabela. Quando pacotes chegam ao *kernel* ou antes de serem enviados pela rede, seus cabeçalhos são analisados e é decidida qual ação deve ser tomada sobre cada pacote com base nas regras. As principais ações possíveis são descartar o pacote e aceitar o pacote.

O uso de um *firewall* para emulação de falhas de colapso permite reproduzir com precisão o comportamento do protocolo TCP em presença de falhas reais. Características do protocolo, como *timeout* para a retransmissão de pacotes, confirmação de entrega de dados e gerenciamento dos *buffers* de comunicação são efetivamente realizadas, já que pacotes são descartados pelo próprio sistema operacional e não pelos níveis superiores da aplicação.

Firewalls foram utilizados para emulação de falhas nas ferramentas Loki (LEFEVER; CUKIER; SANDERS, 2003) e FIRMI (VACARO, 2007). No caso de FIRMI, a emulação de falhas é feita em dois níveis de abstração: mensagens RMI são interceptadas no nível da JVM, obtendo informações sobre o protocolo RMI, e falhas de colapso são efetivamente injetadas no nível do sistema operacional através do uso de regras de *firewall*. As informações obtidas no nível da JVM são utilizadas para criar as regras de *firewall*. Devido ao sucesso do uso dessa abordagem, ela foi utilizada na arquitetura de Comform, como será tratado no próximo Capítulo.

4.3 Conclusões do Capítulo

Este Capítulo apresentou questões de projeto importantes para o desenvolvimento de uma solução para injeção de falhas de comunicação em aplicações multiprotocolo que siga os requisitos identificados no Capítulo 2.

A inadequação da emulação de falhas de colapso somente no nível da JVM para o teste de aplicações baseadas em TCP foi abordada. Essa inadequação implica na necessidade de se trabalhar no nível do sistema operacional para a emulação de falhas que envolvem o descarte de mensagens.

A maior contribuição do Capítulo foi a apresentação de um modelo genérico para injeção de falhas de comunicação em aplicações multiprotocolo. A elaboração do modelo levou em consideração o problema da inadequação de emulação de falhas de colapso somente no nível da JVM para o teste de aplicações baseadas em TCP. A fim de tratar aplicações multiprotocolo, o modelo prevê que pacotes relacionados a todos os protocolos de interesse, isto é, todos os protocolos usados pela aplicação alvo, devem ser passíveis de seleção e manipulação. Visando atender aos requisitos identificados para a solução, o modelo também requer a interceptação de mensagens em um nível de abstração mais alto

do que o nível do sistema operacional de modo que informações específicas relacionadas à aplicação alvo possam ser facilmente recuperadas para a ativação e desativação de falhas.

O Capítulo também discutiu possibilidades de abordagens a serem empregadas no projeto de um injetor de falhas para aplicações Java. O uso de JVMTI e do pacote `java.lang.instrument` como suporte à instrumentação de aplicações Java foi apresentado. Conforme explicado no Capítulo, essas abordagens não são capazes de realizar a manipulação de *bytecode* propriamente dita, mas são capazes de interceptar o carregamento de classes provendo o suporte à instrumentação. Javassist foi apresentada como alternativa para manipulação de *bytecode*. Foi apresentada também a abordagem de projeto de injetores de falhas baseada em programação orientada a aspectos. Por fim, o uso de *firewalls* para emulação de falhas foi abordado. Como explicado no decorrer do Capítulo, a abordagem escolhida para o projeto de Comform foi a combinação do uso do pacote `java.lang.instrument` e de Javassist para instrumentação e de *firewalls* para emulação de falhas que envolvem o descarte de mensagens.

O próximo Capítulo apresenta a arquitetura do injetor de falhas Comform e utiliza o conhecimento apresentado neste Capítulo e também o apresentado no Capítulo 3 sobre as APIs de Java para comunicação usando UDP, TPC e RMI.

5 INJETOR DE FALHAS COMFORM

Este Capítulo mostra como foi desenvolvido o injetor de falhas Comform (COMmunication Fault injector ORiented to Multi-protocol Java applications), uma solução que segue o modelo genérico estabelecido na Seção 4.1 e que cumpre os demais requisitos identificados na Seção 2.3.2. O protótipo pode ser aplicado para injetar falhas em aplicações Java baseadas em qualquer combinação dos protocolos UDP, TCP e RMI (incluindo aquelas baseadas em um único protocolo).

A Seção 5.1 apresenta a arquitetura de Comform. A seguir, são detalhadas informações sobre as classes instrumentadas (Seção 5.2) e sobre componentes importantes da arquitetura como o Filtro de Mensagens (Seção 5.3), o Monitor (Seção 5.4) e modelagem de Falhas e Cargas de Falhas (Seção 5.5). Também é analisada a portabilidade de Comform (Seção 5.6) e apresentada uma discussão sobre a expansão de Comform para novos protocolos (Seção 5.7). Por fim, são discutidos aspectos práticos sobre o uso do protótipo (Seção 5.8).

5.1 Arquitetura de Comform

Esta Seção apresenta a arquitetura básica de Comform, que foi inspirada na arquitetura de FIRMI (VACARO, 2007). Como descrito na Seção 4.1, o modelo genérico requer um modo de selecionar e manipular mensagens no nível do sistema operacional de forma a emular corretamente falhas que envolvem o descarte de pacotes (como falhas de colapso de nodos e *links*). A arquitetura usa um componente Firewall para esse fim. Esta abordagem foi aplicada com sucesso no injetor de falhas FIRMI e seu reuso mostrou-se conveniente. Conforme mencionado anteriormente, a ferramenta Loki também faz uso de um *firewall* para injeção de falhas de particionamento de rede (LEFEVER; CUKIER; SANDERS, 2003). O uso de regras de *firewall* para o descarte de pacotes evita problemas como o descrito na Figura 4.1, pois os pacotes são descartados pelo próprio sistema operacional, antes que eles sejam entregues à aplicação e antes que sejam enviados a outros nodos. Deste modo, os nodos que devem perceber o estado falho do nodo onde a injeção está sendo realizada o farão de modo consistente.

O modelo também requer um modo de interceptar mensagens em um nível de abstração mais alto que o do sistema operacional, de modo que informações específicas relacionadas à aplicação alvo possam ser facilmente recuperadas para *ativação* e *desativação* de falhas em testes de caixa branca. Seguindo a abordagem usada em outras ferramentas ((JACQUES-SILVA et al., 2006), (VACARO; WEBER, 2006), (FARCHI; KRASNY; NIR, 2004)), classes que implementam os protocolos de interesse – UDP, TCP e RMI – são instrumentadas no nível da JVM. Com a sua instrumentação, é possível obter informações de interesse para ativação de falhas e também para emulação de falhas. Para

promover interação entre a parte da arquitetura que opera no nível da JVM e a que opera no nível do sistema operacional, informações de portas locais sendo utilizadas pela aplicação alvo são obtidas por meio da instrumentação e usadas para a construção de regras de *firewall*. Ainda, conforme mencionado na Seção 4.1, falhas de temporização podem ser emuladas no nível da JVM.

Os tipos de falhas de comunicação que podem, atualmente, ser emulados por Comform incluem colapso de nodos, colapso de *links* e falhas de temporização. O protótipo pode ser estendido com a inclusão de novos tipos de falhas, como falhas de omissão, mas já possui uma variedade de tipos de falhas mais rica do que a oferecida por ferramentas como FAIL-FCI (HOARAU; TIXEUIL; VAUCHELLES, 2007).

A Figura 5.1 apresenta uma visão simplificada da arquitetura de Comform. Uma linha tracejada separa o nível da JVM do nível do sistema operacional (SO). O *Firewall* e as implementações dos protocolos TCP e UDP na pilha de protocolos TCP/IP operam no nível do sistema operacional. No nível da JVM, a Aplicação Alvo é executada e as classes de comunicação de interesse da API Java são instrumentadas em tempo de carga. A instrumentação dessas classes provê a interação com o componente Controlador do injetor de falhas. Este componente é responsável pelo controle dos experimentos de injeção de falhas e interage com os outros componentes do injetor. O Monitor coleta informações importantes sobre o experimento. A Carga de Falhas (*Faultload*) inclui as noções de Carga de Falhas propriamente dita e de Carregador de Carga de Falhas. Como será explicado posteriormente, em Comform *Faultloads* são descritos como classes Java e um Carregador de Carga de Falhas é responsável por sua carga. Esses conceitos são reusados de FIRMI (e também foram reusados em um trabalho relacionado (CÉZANE et al., 2009)). A caixa Falha representa as falhas que a ferramenta é capaz de emular. Por fim, o Filtro de Mensagens é responsável pela efetiva injeção das falhas especificadas por um módulo de Carga de Falhas. A interface *Firewall* representa a interação com um *Firewall* de modo a emular corretamente falhas caracterizadas pelo descarte de mensagens.

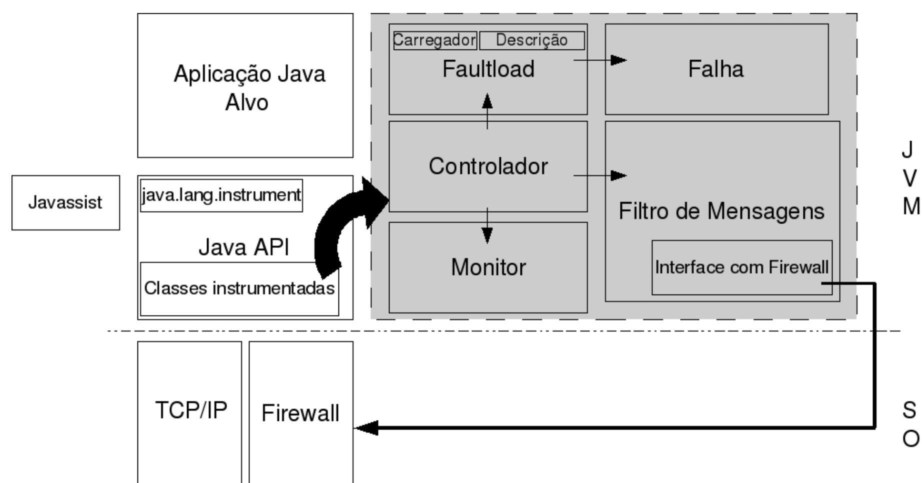


Figura 5.1: Arquitetura simplificada da ferramenta.

O pacote `java.lang.instrument` (SUN MICROSYSTEMS, 2008), apresentado na Seção 4.2.1.2, é utilizado na arquitetura para a interceptação do carregamento de classes na JVM. Deste modo, é obtido acesso ao *bytecode* de classes Java que implementam os protocolos de interesse. A adição de código especial nessas classes, ou seja, a instrumentação dessas classes, é necessária para que elas interajam com o Controlador

do injetor de falhas quando seus objetos forem invocados pela aplicação alvo. Conforme explicado na Seção 4.2.1.2, apesar de prover acesso às classes de interesse e possibilitar adição de *bytecodes*, o pacote `java.lang.instrument` não é capaz de realizar a instrumentação de código propriamente dita. Deste modo, uma biblioteca especializada em instrumentação de *bytecodes* deve ser selecionada para essa tarefa. Javassist (CHIBA, 1998) (CHIBA, 2000), apresentada na Seção 4.2.2, foi escolhida para esse propósito. Como ela não faz parte da API de Java, aparece representada na Figura 5.1 como uma caixa separada. Javassist foi usada anteriormente nos injetores Jaca (MARTINS; RUBIRA; LEME, 2002) e FIRMI (VACARO, 2007).

O funcionamento básico do injetor pode ser resumido da seguinte forma e ficará mais claro no decorrer do Capítulo:

1. Uma instância do Controlador (classe `Controller`) é obtida e é feita instrumentação das classes de interesse dos protocolos alvo durante o seu carregamento. No contexto da criação de uma instância de `Controller`, é feito o carregamento do *Faultload* a ser utilizado no experimento (classe derivada da classe `Faultload` desenvolvida pelo usuário) e execução de seu método construtor. O carregamento do *Faultload* é feito por um Carregador de Módulos de Carga de Falhas (classe `BaseFaultloadLoader`).

2. Quando é realizada a associação de um `socket` a uma porta local no nodo o injetor está sendo executado, a porta local é registrada no Filtro de Mensagens (classe `MessageFilter`).

3. Quando uma mensagem dos protocolos de interesse é interceptada, ela é processada pela classe `Controller`, que aciona a coleta de dados pelo Monitor (classe `Monitor`), registra a mensagem em *log*, invoca o método `update` do `Faultload` para que sejam realizadas possíveis decisões sobre ativação de falhas (podendo considerar o conteúdo da mensagem) e, finalmente, injeta as possíveis falhas ativadas no passo anterior por meio da classe `MessageFilter`.

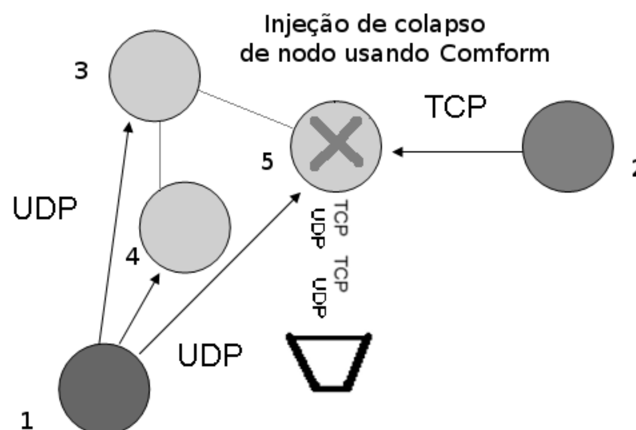


Figura 5.2: Adequação da injeção de falha de colapso em um nodo Zorilla usando Comform.

Fazendo referência à Figura 2.2, se Comform fosse usado, em vez de FIONA, para injeção de colapso no nodo 5, a falha seria emulada de forma adequada. A Figura 5.2

mostra que, com o uso de Comform como substituto de FIONA, tanto mensagens UDP como TCP são descartadas no nodo 5 e todos os demais nodos podem perceber este colapso. Isso é feito usando regras de *firewall* que descartam pacotes relacionados a esses protocolos.

5.2 Classes Instrumentadas

Alguns injetores de falhas de comunicação que realizam instrumentação de classes Java (JACQUES-SILVA et al., 2006) (SILVEIRA, 2005) (FARCHI; KRASNY; NIR, 2004) (GERCHMAN; WEBER, 2006) (VACARO, 2007) foram analisados quanto às classes instrumentadas e guiaram o estudo sobre quais classes e métodos deveriam ser instrumentados. O conhecimento dos trabalhos anteriores foi aproveitado, mas foi insuficiente para o projeto da instrumentação de classes realizada no contexto de Comform, como será explicado nesta Seção. Assim, foi necessário compreender a arquitetura e implementação das classes Java responsáveis pelos protocolos de interesse, introduzidas no Capítulo 4. Além disso, um estudo mais detalhado do próprio código fonte das classes foi essencial para decisões sobre quais classes e métodos deveriam ser instrumentados e para a implementação destas questões no injetor de falhas.

Conforme será mostrado adiante, a instrumentação de classes dos pacotes `sun.*`, que não são parte padrão da plataforma Java, se fez necessária. O estudo do código fonte das classes de interesse deste pacote foi possível por meio de pesquisas no DocJar (JAX SYSTEMS LLC, 2005), um *site* de busca de APIs Java de código aberto.

A Seção 5.2.1 trata da instrumentação das implementações dos protocolos UDP e TCP nos pacote `java.net` (5.2.1.1) e `java.nio` (5.2.1.2). Por fim, a Seção 5.2.2 trata da instrumentação da implementação do protocolo RMI. Para evitar a poluição visual, os parâmetros de alguns métodos serão omitidos em algumas partes do texto, sem prejuízos à compreensão (por exemplo, `send(DatagramPacket p)` é abreviado para `send()`).

5.2.1 Instrumentação de UDP e TCP

As portas locais utilizadas pela aplicação alvo no nodo onde o injetor está sendo executado devem ser obtidas e *registradas* assim que é feita a associação (*binding*) de cada *socket* UDP ou TCP a uma porta local. O *registro de portas* em Comform corresponde ao armazenamento dos números das portas locais em estruturas de dados para possível uso posterior no caso de ativação de falhas de colapso de nodo ou de colapso de *link*. Constatou-se que, no caso de UDP, as portas locais devem ser registradas, obrigatoriamente, antes do envio e recebimento de mensagens. No caso de TCP, as portas locais devem ser registradas, obrigatoriamente, antes da abertura ativa ou passiva de uma conexão. Para o propósito de obtenção e registro de portas locais, após estudo detalhado das classes de interesse, optou-se por instrumentar os métodos `bind()`, que tratam da associação de um *socket* a um endereço e porta locais. Mais adiante, serão explicados detalhes sobre a construção de regras de *firewall* com base nas portas locais registradas.

Métodos relacionados ao envio e recebimento de mensagens UDP e TCP devem ser instrumentados. A instrumentação destes métodos é importante para a obtenção de informações úteis para ativação de falhas e também para a injeção de falhas de temporização. No caso de TCP, métodos relacionados ao estabelecimento de uma conexão também devem ser instrumentados. Além da instrumentação de métodos relativos ao estabelecimento de conexão também ser útil para propósitos de ativação de falhas, ela mostrou-se necessária para obtenção das portas locais sendo utilizadas nos casos em que o método

`bind()` não é invocado pelos construtores (de `Socket` e `ServerSocket`) utilizados, nem é invocado diretamente pelo programador da aplicação alvo. Nestes casos, observou-se que o *binding* é realizado por uma função nativa, cuja instrumentação é indesejável por questões de portabilidade. No caso do pacote `java.net`, o *binding* é feito somente durante a invocação do método `connect()` no *socket* cliente e durante a invocação do método `accept()` no *socket* servidor. Já no caso do pacote `java.nio`, o *binding* é feito somente durante a invocação do método `connect()` de `SocketChannel` e durante a invocação do método `accept()` de `ServerSocketChannel`. Para contornar o problema e obter a porta local sendo utilizada em casos assim, é incluída uma chamada a um método `bind()` no código inserido para instrumentação de métodos relacionados ao estabelecimento de conexão.

As Figuras 5.3 e 5.4 apresentam, resumidamente, os princípios de instrumentação aplicados no projeto e implementação de Comform. Cabe relembrar que o acesso ao código das classes de interesse é provido pelo pacote `java.lang.instrument` em tempo de carga das classes e que os trechos de código são inseridos com o uso de Javassist. Na prática, Javassist permite que trechos de código sejam inseridos ao início e fim do código de um método ou mesmo em pontos no meio do corpo de um método, seguindo determinadas regras (CHIBA, 2007). Para a implementação do injetor, foi necessário estudar detalhadamente o código das classes de interesse a fim de obter o conhecimento de onde exatamente, dentro do corpo de um método, os trechos de código deveriam ser inseridos.

A Figura 5.3 sintetiza a instrumentação de métodos `bind()`. O trecho de código inserido a um método `bind()` é responsável pela invocação de um método do injetor de falhas que faz o registro de portas locais.

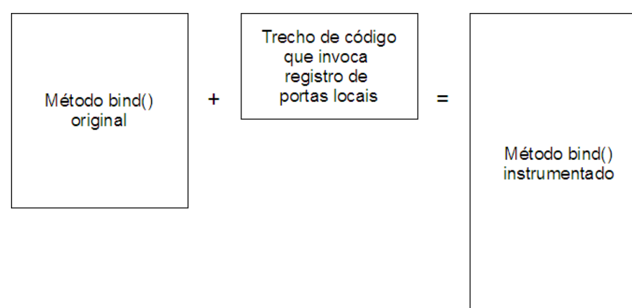


Figura 5.3: Instrumentação de métodos `bind()`.

A Figura 5.4 sintetiza a instrumentação de métodos relacionados a envio e recebimento de mensagens. O trecho de código inserido a um método relacionado a envio ou recebimento de mensagens é responsável pela invocação de um método do injetor de falhas que processa cada mensagem enviada ou recebida.

A Figura 5.5 sintetiza a instrumentação de métodos relacionados ao estabelecimento de conexão. O trecho de código inserido a um método relacionado ao estabelecimento de conexão é responsável pela invocação de um método do injetor de falhas que processa cada mensagem enviada ou recebida e, quando necessário (ou seja, quando ainda não feita associação de portas locais a um nodo antes do estabelecimento de conexão), é responsável pela prévia invocação de um método `bind()`.

As classes instrumentadas são as efetivamente utilizadas pela aplicação alvo durante a sua execução.

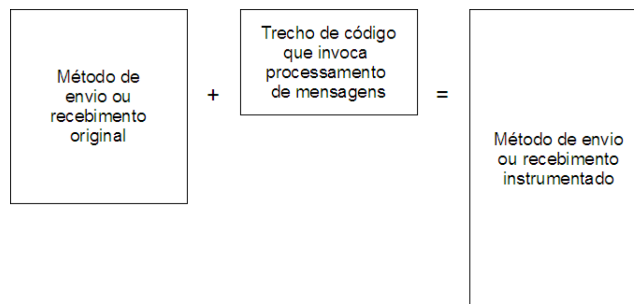


Figura 5.4: Instrumentação de métodos relacionados ao envio e recebimento de mensagens.

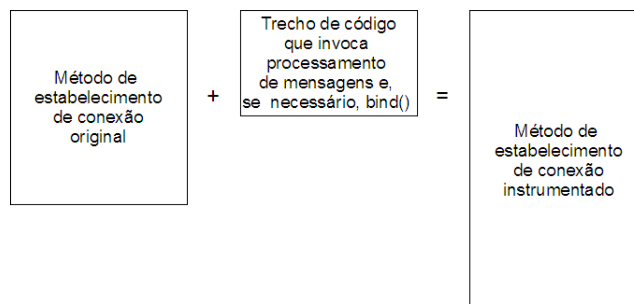


Figura 5.5: Instrumentação de métodos relacionados ao estabelecimento de conexão.

5.2.1.1 *Instrumentação de UDP e TCP no pacote java.net*

Considerando a implementação do protocolo UDP no pacote `java.net` (SUN MICROSYSTEMS, 2008), a classe `DatagramSocket`, que representa um *socket* para o envio e recebimento de pacotes UDP, deve ser instrumentada.

Considerando a implementação do protocolo TCP no pacote `java.net` (SUN MICROSYSTEMS, 2008), as classes `Socket`, `ServerSocket`, `SocketInputStream` e `SocketOutputStream` devem ser instrumentadas. A classe `ServerSocket` implementa *sockets* TCP para o lado servidor enquanto a classe `Socket` representa um *socket* TCP cliente. As classes `SocketInputStream` e `SocketOutputStream` são responsáveis, respectivamente, pelo recebimento e envio de dados usando TCP.

A Tabela 5.1 apresenta as assinaturas dos métodos que devem ser instrumentados em cada uma das classes de interesse.

No injetor de falhas FIERCE (GERCHMAN; WEBER, 2006), voltado a aplicações baseadas em TCP, as classes `Socket`, `ServerSocket`, `SocketInputStream` e `SocketOutputStream` também eram instrumentadas, mas de formas bastante diferentes da realizada para Comform. Na verdade, os problemas apresentados na Seção 4.1.2, sobre a inadequação de emulação de falhas de colapso somente no nível da JVM em aplicações TCP, mostraram a necessidade dessas diferenças e afetaram também a instrumentação do protocolo UDP. Comform, contrariamente a FIERCE, instrumenta o método `bind()` e, assim, obtém as portas locais utilizadas pela aplicação alvo. As portas TCP locais obtidas no nível da JVM são registradas e utilizadas para a configuração de regras de *firewall* no caso de injeção de falhas de colapso, de modo que o descarte de mensagens é feito no nível do sistema operacional. Os métodos `read()`, `write()`, `connect()` e `accept()` são instrumentados tanto em FIERCE como em Comform, mas seguindo abordagens diferentes. No caso de Comform, eles são instrumentados com

Tabela 5.1: Métodos instrumentados em `DatagramSocket`, `ServerSocket`, `Socket`, `SocketInputStream` e `SocketOutputStream`.

Métodos Instrumentados em `DatagramSocket`
`public void bind(SocketAddress addr)`
`public void send(DatagramPacket p)`
`public void receive(DatagramPacket p)`
Métodos Instrumentados em `ServerSocket`
`public void bind(SocketAddress endpoint, int backlog)`
`protected final void implAccept(Socket s)`
Métodos Instrumentados em `Socket`
`public void bind(SocketAddress bindpoint)`
`public void connect(SocketAddress endpoint, int timeout)`
Métodos Instrumentados em `SocketInputStream`
`public int read(byte b[], int off, int length)`
Métodos Instrumentados em `SocketOutputStream`
`private void socketWrite(byte b[], int off, int len)`

propósitos de ativação de falhas e injeção de falhas de temporização. FIERCE, que é inspirado em FIONA (JACQUES-SILVA et al., 2006), instrumenta esses métodos de modo que a emulação de falhas é feita somente no nível da JVM.

Em injetores de falhas para aplicações basedas em UDP (JACQUES-SILVA et al., 2006), (SILVEIRA, 2005), (FARCHI; KRASNY; NIR, 2004), `DatagramSocket` era instrumentada, mas de formas bastante diferentes de Comform. Tomando por exemplo a abordagem de FIONA, para emulação de falhas de colapso, é realizada a inibição do envio e recebimento de mensagens no próprio nível da JVM no contexto dos métodos `send()` e `receive()`. Ainda, FIONA instrumenta somente esses métodos. Comform, além de instrumentar os métodos `send()` e `receive()`, instrumenta o método `bind()` para obtenção das portas locais utilizadas pela aplicação alvo. Assim como para TCP, as portas locais UDP obtidas no nível da JVM são registradas e utilizadas para a configuração de regras de *firewall* no caso de injeção de falhas de colapso, de modo que o descarte de mensagens é feito no nível do sistema operacional.

5.2.1.2 Instrumentação de UDP e TCP no pacote java.nio

Conforme abordado na Seção 3, o pacote `java.nio`, introduzido no Java 1.4, provê uma API alternativa para utilização dos protocolos TCP e UDP (essa API não é uma substituta do pacote `java.net`). Estas classes devem ser instrumentadas de modo a cobrir todas as formas providas por Java para comunicação usando esses protocolos. Entretanto, essa API não foi tratada em injetores de falhas anteriores, como FIONA e FIERCE. Em um dos trabalhos analisados (FARCHI; KRASNY; NIR, 2004), a existência da API é apenas mencionada, mas também não é tratada. Assim, Comform é pioneiro na injeção de falhas em aplicações baseadas nesse pacote.

Neste trabalho, foram identificadas as classes que devem ser instrumentadas no pacote `java.nio`. As classes de interesse foram encontradas especificamente no pacote `java.nio.channels`. Para o protocolo UDP, a classe `DatagramChannel` trata de canais relacionados à comunicação UDP. Esta é uma classe abstrata, de modo que a classe concreta que a estende, e é usada pela aplicação alvo, deve ser instrumentada. Para o protocolo TCP, as classes abstratas `SocketChannel` e `ServerSocketChannel` tratam canais relacionados à comunicação TCP. A primeira é relacionada ao lado cliente e a segunda é relacionada ao lado servidor. Por serem abstratas, também é necessário que

as classes concretas que as estendem, e são usadas pela aplicação alvo, sejam instrumentadas.

A Tabela 5.2 apresenta as assinaturas dos métodos abstratos cujos respectivos métodos concretos devem ser instrumentados. Comform trata da instrumentação de três classes do pacote `sun.nio.ch`: `DatagramChannelImpl`, `SocketChannelImpl`, e `ServerSocketChannelImpl`, que são implementações concretas fornecidas pela Sun, respectivamente, para as classes abstratas `DatagramChannel`, `SocketChannel` e `ServerSocketChannel`.

Tabela 5.2: Métodos de interesse em `DatagramChannel`, `SocketChannel` e `ServerSocketChannel`.

Métodos de Interesse em DatagramChannel
public abstract long write(ByteBuffer srcs, int offset, int length)
public abstract int write(ByteBuffer src)
public abstract long read(ByteBuffer dsts, int offset, int length)
public abstract int read(ByteBuffer dst)
public abstract int send(ByteBuffer src, SocketAddress target)
public abstract SocketAddress receive(ByteBuffer dst)
Métodos de Interesse em SocketChannel
public abstract boolean connect(SocketAddress remote)
public abstract long write(ByteBuffer[] srcs, int offset, int length)
public abstract int write(ByteBuffer src)
public abstract long read(ByteBuffer[] dsts, int offset, int length)
public abstract int read(ByteBuffer dst)
Métodos de Interesse em ServerSocketChannel
public abstract SocketChannel accept()

Cabe observar que o pacote `sun.*` não é uma parte padrão da plataforma Java e, portanto, não há documentação oficial disponível para as classes deste pacote (SUN MICROSYSTEMS, 1996). Estas classes estão presentes no SDK para suportar a implementação da plataforma Java fornecida pela Sun (estas classes podem não estar presentes em implementações de outros fornecedores). Na implementação de Comform, optou-se por tratar da instrumentação da plataforma Java fornecida pela Sun. Em Comform, nomes das classes concretas que devem ser instrumentadas podem ser passados ao injetor como propriedades cujos valores são lidos de um arquivo de configuração do injetor. Por padrão, considera-se que as respectivas classes concretas do pacote `sun.*` devem ser instrumentadas.

5.2.2 Instrumentação de RMI

Conforme explicado na Seção 5.2.1, as portas locais utilizadas pela aplicação alvo no nodo onde o injetor está sendo executado devem ser obtidas e registradas assim que é feita a associação de cada *socket* UDP ou TCP a uma porta local. Considerando Java RMI, que é baseado em TCP, observou-se, estudando o código fonte das classes de RMI, que as classes `Socket` e `ServerSocket` de TCP são utilizadas pela API de RMI (não são utilizadas diretamente pelo programador, mas foram base para a implementação de Java RMI). Deste modo, a instrumentação realizada na implementação do protocolo TCP para obtenção e registro de portas locais é suficiente para tratar obtenção e registro de portas em aplicações baseadas em Java RMI.

Também observou-se que Java RMI faz uso das classes `SocketInputStream` e `SocketOutputStream`, que são responsáveis, respectivamente, pelo recebimento e envio de dados usando TCP (novamente, não são utilizadas diretamente pelo programa-

dor, mas foram base para a implementação de Java RMI). Porém, a instrumentação somente dessas classes de TCP levaria a sérias dificuldades para a realização de *testes de caixa branca*, pois seria muito difícil obter informações particulares sobre o protocolo RMI, uma vez que que essas informações já se encontram serializadas nesse nível de abstração. Por exemplo, seria difícil obter o nome de um método sendo invocado e usar essa informação para ativação de falhas. Assim, para propósitos de ativação de falhas, é interessante instrumentar classes responsáveis por envio e recebimento de mensagens que estejam em um nível de abstração mais alto do que as classes `SocketInputStream` e `SocketOutputStream`.

Segundo estudo realizado no desenvolvimento da ferramenta FIRMI (VACARO, 2007), voltada a aplicações Java baseadas apenas em Java RMI, a localização ideal para instrumentação de envio e recebimento de mensagens RMI, de modo que se obtenha facilmente informações particulares do protocolo para propósitos de ativação de falhas, são as estruturas dos `Stubs` e `Skeletons`. FIRMI instrumenta as classes que implementam as interfaces `java.rmi.server.RemoteRef`, usada em clientes RMI para envio de requisições, e `java.rmi.server.ServerRef`, usada em servidores RMI para recepção de requisições. Tratam-se das estruturas que estão no mais alto nível de abstração do protocolo e a sua instrumentação diminui a intrusividade temporal causada na aplicação alvo, pois, por exemplo, os parâmetros usados na invocação de métodos pela aplicação alvo podem ser diretamente obtidos pelo injetor de falhas (VACARO, 2007).

A Tabela 5.3 apresenta as assinaturas dos métodos abstratos cujos respectivos métodos concretos devem ser instrumentados. Comform trata da instrumentação de duas classes do pacote `sun.rmi.server`: `UnicastRef` e `UnicastServerRef`, que são implementações concretas fornecidas pela Sun, respectivamente, para as interfaces `RemoteRef` e `ServerRef`. Cabe ressaltar que, além da instrumentação deles, a instrumentação dos métodos de `Socket`, `ServerSocket`, `SocketInputStream` e `SocketOutputStream`, cujas assinaturas foram apresentadas na Tabela 5.1, também é essencial para o protocolo RMI em Comform.

Tabela 5.3: Métodos de interesse em `RemoteRef` e `ServerRef`.

Métodos de Interesse em `RemoteRef`
`Object invoke(Remote obj, java.lang.reflect.Method method, Object[] params, long opnum)`
Métodos de Interesse em `ServerRef`
`Object invoke(Remote obj, java.lang.reflect.Method method, Object[] params, long opnum)`

No injetor de falhas FIRMI (VACARO, 2007), diferentemente de Comform, não são instrumentadas as classes de TCP `Socket`, `ServerSocket`, `SocketInputStream` e `SocketOutputStream`. Em FIRMI, para obtenção e registro das portas locais utilizadas pela aplicação alvo, são instrumentadas as classes que implementam as interfaces `RMIClientSocketFactory` e `RMIServerSocketFactory` do pacote `java.rmi.server` e não as classes `Socket` e `ServerSocket`. Outra diferença entre as instrumentações realizadas para FIRMI e Comform diz respeito à implementação de falhas de temporização. No caso de FIRMI, requisições RMI são atrasadas para emulação de falhas de temporização. No caso de Comform, por ser uma ferramenta multiprotocolo, optou-se por usar o mesmo conceito usado para emulação de atraso considerando o protocolo TCP, ou seja, o envio e recebimento de mensagens TCP são atrasados para emulação de falhas de colapso. Assim, em FIRMI, a instrumentação dos métodos

da Tabela 5.3 era útil para emulação de falhas de temporização, além de para ativação de falhas, enquanto para Comform é útil apenas para ativação de falhas.

5.3 Filtro de Mensagens

O Filtro de Mensagens (representado pela classe `MessageFilter` de Comform) é responsável pela efetiva injeção das falhas especificadas por um módulo de Carga de Falhas.

A Figura 5.6 ilustra a modelagem de mensagens dos protocolos UDP, TCP e RMI em Comform. A classe abstrata `Message` é estendida para a definição de mensagens relativas a cada um dos protocolos.

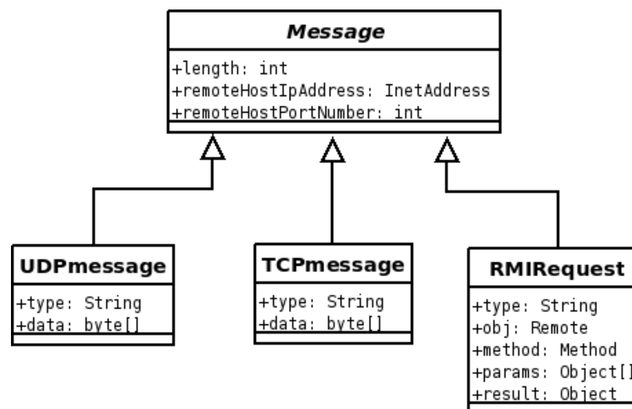


Figura 5.6: Modelagem de mensagens dos protocolos de interesse.

Relembrando a instrumentação dos métodos `bind()`, quando é realizada a associação de um `socket` a uma porta local no nodo o injetor é executado, a porta local é registrada no Filtro de Mensagens (classe `MessageFilter`). Por outro lado, relembrando a instrumentação de métodos relacionados a envio ou recebimento de uma mensagem (incluindo mensagens de estabelecimento de conexão, no caso de TCP), após a interceptação de uma mensagem, ela é modelada conforme a Figura 5.6. Então, a mensagem é processada pelo Controlador (representado pela classe `Controller` de Comform), que trata do armazenamento dos dados da mensagem em *log*, da atualização das medidas coletadas pelo monitor do processamento da mensagem pelo módulo de carga de falhas (que decide a ação a ser executada sobre a mensagem) e, finalmente, do processamento da mensagem pelo Filtro de Mensagens para a efetiva injeção das falhas ativas na aplicação alvo.

A classe `MessageFilter` interage com a classe abstrata `Firewall` para a emulação de falhas que envolvem o descarte de pacotes, como colapsos de nodos e colapsos de *links*. Regras de *firewall* são construídas com base nas portas UDP e TCP locais registradas no Filtro de Mensagens. Quando da ativação de falhas de colapso de nodo, Comform constrói regras correspondentes ao descarte de mensagens dos protocolos UDP e TCP que tem as portas locais registradas como sua origem ou seu destino, de modo que a aplicação não seja mais capaz de se comunicar com outros nodos. Por questões de implementação, Comform não trata a recuperação em caso de colapso de nodo, de modo que não é possível desativar este tipo de falha. No caso de falhas de colapso de *link*, que podem ser ativadas e desativadas durante um experimento, modelando o fato de o *link* oscilar entre estados ativo e inativo, Comform tanto constrói regras correspondentes ao descarte de mensagens como deleta essas regras quando as falhas devem ser desativadas.

A seguir, é apresentado o formato das regras aplicadas ao IPTables por Comform. Para o caso de Comform ser executado em um nodo e falhas precisarem ser percebidas em todos os outros nodos da aplicação alvo, o formato aplicado é o apresentado na Tabela 5.4. No caso de falhas de colapso *link*, é previsto também que Comform seja executado em um nodo e emule falhas somente no *link* entre esse e outro(s) nodo(s) determinados da aplicação alvo. Por exemplo, considerando uma aplicação alvo que seja executada em três nodos, A, B e C, é possível emular falhas de colapso de *link* somente no *link* entre A e C. Nestes casos, o formato das regras é o apresentado na Tabela 5.5. A *tag* "*<inserção ou deleção de regra>*" informa que é necessário definir se uma regra deverá ser inserida ou retirada do IPTables. O valor "-A" representa a adição de uma regra, enquanto o valor "-D" representa a deleção de uma regra. Uma regra pode atuar tanto nos pacotes recebidos (atributo INPUT) como nos pacotes enviados (atributo OUTPUT). A *tag* "*<protocolo>*" informa que é necessário definir em qual protocolo a regra será aplicada. O valor "udp" representa o protocolo UDP, enquanto o valor "tcp" representa o protocolo TCP. A *tag* "*<endereço de origem e possivelmente máscara de subrede>*" informa que é necessário definir a qual endereço de origem a regra será aplicada e, opcionalmente, a quais máscaras de subrede. Uma máscara de subrede é normalmente formada pelos valores "255" ou "0" em cada um dos octetos, onde o primeiro valor indica a parte do endereço IP referente à rede, enquanto o segundo indica a parte do endereço IP referente ao *host*. As *tags* "*<porta de origem>*" e "*<porta de destino>*" informam que é necessário definir a qual porta a regra será aplicada.

Tabela 5.4: Formato de regras de *firewall* aplicado quando Comform é executado em um nodo e falhas de colapso de nodo e de colapso de *link* devem ser percebidas em todos os outros nodos da aplicação alvo.

<inserção ou deleção de regra> **INPUT -p** *<protocolo>* **- -dport** *<porta de origem>*
<inserção ou deleção de regra> **OUTPUT -p** *<protocolo>* **- -sport** *<porta de destino>*

Tabela 5.5: Formato de regras de *firewall* aplicado quando Comform é executado em um nodo e emula falhas de colapso de *link* somente no *link* entre esse e outro(s) nodo(s) determinados da aplicação alvo.

<inserção ou deleção de regra> **INPUT -p** *<protocolo>* **-s** *<endereço de origem e possivelmente máscara de subrede>* **- -dport** *<porta de origem>*
<inserção ou deleção de regra> **OUTPUT -p** *<protocolo>* **-d** *<endereço de destino e possivelmente máscara de subrede>* **- -sport** *<porta de destino>*

Falhas de temporização, ao contrário de falhas que envolvem o descarte de mensagens, são emuladas completamente no nível da JVM. Quando uma mensagem é enviada ou recebida, caso ela seja selecionada para injeção de falha de temporização, a *thread* de execução do método responsável por esse envio ou recebimento é suspensa pelo tempo definido no Módulo de Carga de Falhas. Somente métodos relacionados ao envio e recebimento de mensagens UDP e TCP sofrem atraso. Os métodos de mais alto nível instrumentados para o protocolo RMI não devem ser atrasados, já que levariam à realização de atraso de forma redundante e, portanto, incorreta.

5.4 Monitoramento e Coleta de Dados

O Monitor é informado sobre as mensagens interceptadas e coleta informações sobre os protocolos alvo durante um experimento.

Entre as medidas coletadas pelo Monitor estão o total de *bytes* enviados ou recebidos e o total de requisições RMI efetuadas. A obtenção destas medidas não tem impacto considerável em termos de intrusividade temporal, pois resume-se ao emprego de contadores.

As informações coletadas pelo Monitor são acessíveis aos Módulos de Carga de Falhas e podem ser utilizadas para propósitos de ativação de falhas.

5.5 Falhas e Cargas de Falhas

A estratégia empregada para descrição de Módulos de Carga de Falhas (*Faultloads*) é baseada em classes Java e usa uma API inspirada em FIRMI (VACARO, 2007). Esta abordagem e mesmo a API foram reusadas em um trabalho recente (CÉZANE et al., 2009) e possuem algumas vantagens em relação ao uso de arquivos de configuração (JACQUES-SILVA et al., 2006), sendo a principal a maior flexibilidade na descrição de cargas de falhas. Para o carregamento de *Faultloads*, Comform utiliza um Carregador de Módulos de Carga de Falhas (VACARO, 2007).

Na Seção 5.5.1, é apresentada a modelagem de falhas em Comform. A seguir, na Seção 5.5.2 é explicado como o usuário deve construir Módulos de Carga de Falhas. Por fim, na Seção 5.5.3 é tratado o carregamento de Módulos de Carga de Falhas.

5.5.1 Modelagem de Falhas

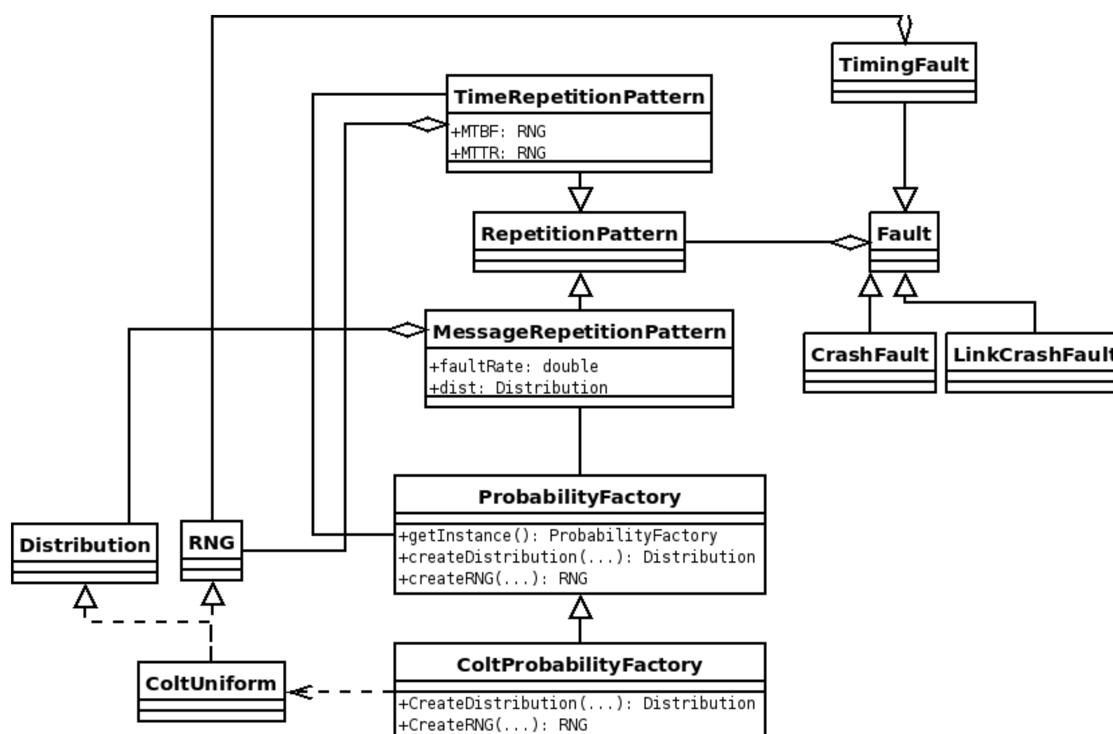


Figura 5.7: Diagrama simplificado de modelagem de falhas.

Esta Seção apresenta os aspectos relacionados à modelagem de falhas em Comform. A modelagem é baseada na arquitetura de especificação de falhas de FIRMI (VACARO,

2007), e é sintetizada na Figura 5.7, que será referenciada ao longo da Seção. São apresentadas as falhas de comunicação suportadas atualmente pelo protótipo (Seção 5.5.1.1), bem como Padrões de Repetição (Seção 5.5.1.2) e Distribuições de Probabilidade e Geradores de Números Pseudorrandômicos (Seção 5.5.1.3), que subsidiam sua modelagem. Por fim, são tratadas opções de ativação de falhas fornecidas aos usuários de Comform (Seção 5.5.1.4).

5.5.1.1 Falhas de Comunicação Suportadas

Os tipos de falhas de comunicação que podem, atualmente, ser emulados por Comform incluem colapso de nodos, colapso de *links* e falhas de temporização. A Figura 5.7 mostra que esses tipos de falhas foram modelados como classes Java que estendem a classe `Fault`. Comform também permite a emulação de falhas bizantinas relacionadas à alteração de parâmetros das mensagens interceptadas. Esse tipo de falha não é modelado por meio de classes Java e é injetado nos próprios Módulos de Carga de Falhas.

A classe `CrashFault` corresponde a falhas de colapso de nodo. Quando um usuário de Comform injeta este tipo de falha, é emulado o colapso do nodo onde o injetor de falhas está sendo executado, que será percebido pelos demais nodos onde a aplicação alvo está sendo executada.

A classe `LinkCrashFault` corresponde a falhas de colapso de *link*. Colapsos de *link* têm comportamento similar a colapsos de nodos, mas os estados dos *links* podem variar no tempo, de modo que às vezes o *link* esteja ativo e às vezes inativo. Estes comportamentos são modelados usando taxas de falhas e distribuições de probabilidade, conforme será visto nas próximas subseções, que apresentam padrões de repetição e distribuições de probabilidade. Quando um usuário de Comform modela este tipo de falha, ele possui algumas opções, conforme introduzido na Seção 5.3. Ele pode emular colapsos de *link* em todos os *links* entre o nodo onde o injetor de falhas está sendo executado e os demais nodos onde a aplicação alvo está sendo executada. Pode também emular colapsos de *link* somente nos *links* entre o nodo onde o injetor está sendo executado e outro(s) nodo(s) determinados da aplicação alvo, passados como parâmetro na instanciação da falha. Há também a opção de o usuário definir máscaras de subrede ao especificar os endereços, o que pode evitar que seja necessário especificar o colapso de cada *link* individualmente.

A classe `TimingFault` corresponde a falhas de temporização. Falhas de temporização representam atrasos no envio e recebimento de mensagens. Quando um usuário de Comform modela este tipo de falha, ele tem opção de emular atrasos em todos os links entre o nodo onde o injetor de falhas está sendo executado e os demais nodos onde a aplicação alvo está sendo executada. Pode também emular atrasos somente nos *links* entre o nodo onde o injetor está sendo executado e outro(s) nodo(s) determinados da aplicação alvo, passados como parâmetro na instanciação da falha. O usuário também deve decidir o valor do atraso aplicado a cada mensagem selecionada, que pode ser um número pseudorrandômico. Por fim, o usuário também precisa determinar como as mensagens serão selecionadas para sofrerem atraso, ou seja, qual padrão de repetição será aplicado. Conforme será explicado a seguir, há opções de uso de padrões de repetição que levam em consideração tempo ou que levam em consideração as mensagens interceptadas.

5.5.1.2 Padrões de Repetição

Conforme mencionado na Seção 5.5.1.1, falhas de colapso de *link* podem se manifestar com a variação dos estados dos *links* no tempo, de modo que às vezes o *link* esteja ativo e às vezes inativo. Falhas de temporização também podem se manifestar de formas

variáveis sobre as mensagens sendo enviadas e recebidas. Padrões de repetição, conceito introduzido no desenvolvimento de FIRMI (VACARO, 2007) e reutilizado neste trabalho, são utilizados para modelar esses comportamentos nas falhas injetadas. Como pode ser observado na Figura 5.7, padrões de repetição são modelados através da classe abstrata `RepetitionPattern` e há dois tipos deles: baseado em tempo e baseado em fluxo de mensagens.

O uso do padrão de repetição baseado em tempo, que é implementado pela classe `TimeRepetitionPattern`, possibilita que a injeção de falhas de colapso de *link* e de temporização seja ativada e desativada com base no avanço do tempo. Este padrão de repetição considera os parâmetros de tempo médio entre falhas (tempo decorrido entre uma ocorrência de falha e a próxima ocorrência) e tempo de duração da falha para modelar a injeção de falhas. Na modelagem de um Módulo de Carga de Falhas que faça uso deste padrão de repetição, um usuário pode optar por usar como valores para esses parâmetros números fixos ou geradores de números pseudorrandômicos.

O uso do padrão de repetição baseado em fluxo de mensagens, implementado pela classe `MessageRepetitionPattern`, possibilita que a injeção de falhas de *temporização* seja ativada e desativada levando em consideração o fluxo de mensagens dos protocolos alvo. O padrão de repetição baseado em fluxo de mensagens considera como parâmetros principais a taxa de falhas e uma distribuição de probabilidade. A taxa de falhas é responsável pelo controle da quantidade de mensagens selecionadas. A distribuição de probabilidade modela o comportamento da seleção das mensagens. O usuário de Comform não pode utilizar este padrão de repetição na modelagem de falhas de colapso de *link*. Isso se deve ao fato de que, após a ativação inicial de uma falha deste tipo e sua injeção, os pacotes dos protocolos alvo são descartados no nível do sistema operacional, impedindo que a informação sobre o recebimento de mensagens relativas a esses protocolos chegue ao nível da JVM. Por outro lado, como falhas de temporização são implementadas somente no nível da JVM, o uso deste padrão de repetição é adequado para a emulação deste tipo de falha.

5.5.1.3 Distribuições de Probabilidade e Geradores de Números Pseudorrandômicos

Como pode ser observado nas duas seções anteriores, Distribuições de Probabilidade e Geradores de Números Pseudorrandômicos são úteis na definição de padrões de repetição de uma falha e na configuração dos parâmetros de alguns tipos de falhas. Estes recursos proporcionam uma maior representativade na emulação de falhas, já que auxiliam na emulação de falhas reais. Alguns trabalhos da literatura evidenciam a importância do uso de modelos probabilísticos para a injeção de falhas de forma realística (WEBER et al., 2009).

Atualmente, Comform possibilita o uso de Distribuição Uniforme de Probabilidade, mas podem ser agregadas a ele opções de uso de outras distribuições. A associação de um Gerador Uniforme de Números Pseudorrandômicos a Funções de Distribuição de probabilidade como Poisson ou Exponencial, possibilita a obtenção de uma distribuição não-uniforme para injeção de falhas (WEBER et al., 2009).

Reutilizando a abordagem adotada em FIRMI (VACARO, 2007), Comform faz uso de um mecanismo baseado no padrão de projeto Abstract Factory (GAMMA et al., 1994), que leva à transparência no uso de qualquer biblioteca de distribuições de probabilidade, de modo que não é estabelecida dependência a uma arquitetura específica. Como pode ser observado na Figura 5.7, a classe `ProbabilityFactory` é o ponto de acesso aos Geradores de Números Pseudorrandômicos (classe `RNG`) e para as Distribuição de Pro-

babilidade (classe `Distribution`). A fábrica concreta atualmente está sendo implementada pela biblioteca Colt (CERN, 1999), que é considerada uma biblioteca bastante confiável pela correção de seus algoritmos (WEBER et al., 2009). A inclusão de uma nova biblioteca de funções de distribuição de probabilidade requer a criação de novas classes concretas para Geradores de Números Randômicos e Distribuições de Probabilidade e também a criação de uma classe que estenda `ProbabilityFactory` e realize a instanciação de objetos dos novos Geradores de Números Pseudorrandômicos e Distribuições de Probabilidade.

5.5.1.4 Ativação de Falhas

Falhas podem ser ativadas durante a carga do injetor de falhas ou podem ser ativadas mais adiante durante a execução da aplicação alvo. Quando falhas são ativadas durante a carga do injetor, elas ficam ativas desde o início do experimento. Já quando falhas são ativadas durante a execução da aplicação alvo, a ativação pode levar em consideração o conteúdo e a quantidade de mensagens interceptadas.

Os atributos de mensagens dos protocolos de interesse podem ser usados para propósitos de ativação de falhas. A modelagem das mensagens dos protocolos de interesse foi apresentada na Figura 5.6. Uma requisição RMI, por exemplo, é representada pela classe `RMIRequest` e possui como atributos principais o tipo de requisição, o endereço de rede do nodo remoto, a referência remota usada, o método que está sendo invocado e os valores dos parâmetros deste método. É possível construir um Módulo de Carga de Falhas que faça ativação de falhas com base, por exemplo, no nome de um método sendo invocado.

Ainda relembrando a Figura 5.6, mensagens UDP e TCP possuem como atributos o tipo de mensagem, os dados sendo enviados ou recebidos, o tamanho da mensagem, o endereço de rede do nodo remoto e a porta do nodo remoto.

Em especial, os atributos correspondentes a tipos de mensagem UDP e TCP ou tipo de requisição RMI podem assumir vários valores (modelados como `Strings`), conforme a Tabela 5.6. Essas possibilidades de ativação de falhas fornecidas por Comform beneficiam a realização de *testes de caixa branca*. Para cada método relacionado a envio ou recebimento de mensagens (ou estabelecimento de conexão), existe a possibilidade de realizar ativação de falhas antes de sua execução (ao início da execução do método) e após sua execução (logo antes do método encerrar sua execução e retornar). Por exemplo, é possível realizar ativação de falhas antes da execução de um método remoto por um servidor (“RMI_beforeExecuting”) ou logo antes da execução do método remoto retornar (“RMI_afterExecuting”). Os métodos `bind()` não aparecem na Tabela, pois sua instrumentação serve apenas para a obtenção e registro de portas locais junto ao Filtro de Mensagens.

Também é possível ativar falhas com base em informações coletadas pelo Monitor, como o total de *bytes* enviados ou recebidos ou o total de requisições RMI efetuadas. O acesso à instância do Monitor é obtido através do componente Controlador.

5.5.2 Construção de Módulos de Carga de Falhas

Em Comform, Módulos de Carga de Falhas (*Faultloads*) são codificados pelo usuário como extensões da classe abstrata `Faultload` da API de Comform. Um Módulo de Carga de Falhas tem a responsabilidade de decidir a ação a ser executada sobre as mensagens interceptadas pelo injetor.

Após o engenheiro de testes projetar um experimento de injeção de falhas, definindo

Tabela 5.6: Valores para o atributo tipo (`type`) de mensagens UDP, TCP e RMI.

Métodos UDP instrumentados (java.net)	**Valores para atributo tipo**
send	“UDP_send_before”, “UDP_send_after”
receive	“UDP_receive_before”, “UDP_receive_after”
Métodos UDP instrumentados (java.nio)	**Valores para atributo tipo**
long write	“UDPNio_writeLong_before”, “UDPNio_writeLong_after”
int write	“UDPNio_writeInt_before”, “UDPNio_writeInt_after”
long read	“UDPNio_readLong_before”, “UDPNio_readLong_after”
int read	“UDPNio_readInt_before”, “UDPNio_readInt_after”
send	“UDPNio_send_before”, “UDPNio_send_after”
receive	“UDPNio_receive_before”, “UDPNio_receive_after”
Métodos TCP instrumentados (java.net)	**Valores para atributo tipo**
implAccept	“TCP_implAccept_before”, “TCP_implAccept_after”
connect	“TCP_connect_before”, “TCP_connect_after”
read	“TCP_read_before”, “TCP_read_after”
socketWrite	“TCP_socketWrite_before”, “TCP_socketWrite_after”
Métodos TCP instrumentados (java.nio)	**Valores para atributo tipo**
connect	“TCPNio_connect_before”, “TCPNio_connect_after”
long write	“TCPNio_writeLong_before”, “TCPNio_writeLong_after”
int write	“TCPNio_writeInt_before”, “TCPNio_writeInt_after”
long read	“TCPNio_readLong_before”, “TCPNio_readLong_after”
int read	“TCPNio_readInt_before”, “TCPNio_readInt_after”
accept	“TCPNio_accept_before”, “TCPNio_accept_after”
Métodos RMI instrumentados	**Valores para atributo tipo**
invoke (servidor)	“RMI_beforeExecuting”, “RMI_afterExecuting”
invoke (cliente)	“RMI_beforeInvoking”, “RMI_afterInvoking”

o tipo de falha, quando ela deverá ser ativada, o seu padrão de repetição e, opcionalmente, o uso de distribuições de probabilidade, ele deverá construir o Módulo de Carga de Falhas correspondente para Comform seguindo os passos básicos abaixo:

1. Criação de uma classe Java derivada da classe `Faultload`.
2. Implementação do método construtor dessa nova classe com a codificação do que deverá ser executado na inicialização do injetor de falhas no escopo desse método.
3. Implementação do método `update` para que o Módulo de Carga de Falhas seja notificado sobre as mensagens interceptadas durante a execução da aplicação alvo.
4. Instanciação da falha a ser ativada.
5. Ativação da falha (seja no escopo do método construtor ou do método `update` conforme o experimento projetado) usando o método `activate`.

Quando falhas são ativadas no escopo do método construtor, elas ficam ativas desde o início do experimento. Já quando falhas são ativadas no escopo do método `update`, a ativação pode levar em consideração o conteúdo e a quantidade de mensagens interceptadas.

Conforme explicado anteriormente, as falhas ativadas são efetivamente injetadas na aplicação alvo pelo Filtro de Mensagens.

5.5.3 Carregamento de Módulos de Carga de Falhas

No contexto da criação de uma instância de `Controller`, é feito o carregamento do Módulo de Carga de Falhas (*Faultload*) a ser utilizado no experimento e execução de seu método construtor. Conforme visto anteriormente, um Módulo de Carga de Falhas,

em Comform, é uma classe derivada da classe `Faultload` e desenvolvida pelo usuário. O carregamento do *Faultload* é feito pelo método `load()` do Carregador de Módulos de Carga de Falhas (classe `BaseFaultloadLoader`) (VACARO, 2007), que, além de carregar o *Faultload*, retorna o objeto correspondente a ele.

O funcionamento de um Carregador de Módulos de Carga de Falhas se dá com o auxílio da classe `ClassLoader` da API de Java, que é responsável pelo carregamento de classes.

5.6 Portabilidade

Com relação às propriedades de portabilidade de Comform, além de ser implementado completamente em Java, seu projeto favoreceu a portabilidade em outros pontos também, como no uso de *firewalls*.

A solução não é dependente de uma implementação específica de *firewall*, pois podem ser desenvolvidos módulos para o tratamento de vários *firewalls*. Empregando o padrão de projeto *Factory Method* (GAMMA et al., 1994), a arquitetura do injetor de falhas define uma abstração para interação com diferentes implementações de *firewall* de diferentes sistemas operacionais (VACARO; WEBER, 2006). Atualmente, a implementação de Comform trata do IPTables (RUSSEL; WELTE, 2002). Para fins de portabilidade, é mais adequado exigir que um usuário tenha um *firewall* qualquer do que exigir que ele tenha um sistema operacional específico.

Por consequência do padrão de projeto usado, para que um novo *firewall* seja tratado por Comform, deve ser codificada uma classe que estenda a classe abstrata `Firewall` da arquitetura do injetor, implementando adequadamente seus métodos abstratos. A classe `Firewall` representa a interação do injetor de falhas com um *firewall*. A execução de comandos relativos à inclusão de regras no *firewall* específico sendo utilizado deve ser realizada pela classe concreta, pois tratam-se de aspectos específicos de cada diferente implementação de *firewall*.

A classe `IPTables` de Comform é responsável pela interação do injetor com o IPTables. Ela estende a classe `Firewall` implementando seus métodos abstratos. A interação com o IPTables é realizada com o auxílio da classe `Runtime` da API de Java (SUN MICROSYSTEMS, 2008), que permite a interface entre uma aplicação e o ambiente em que está sendo executada.

5.7 Expansão de Comform para Novos Protocolos

Considerando outros protocolos de comunicação, seria útil que Comform tratasse, além de RMI, de outros protocolos de mais alto nível largamente utilizados que sejam baseados em UDP ou TCP.

Quando a implementação de um protocolo faz uso das classes de UDP e TCP instrumentadas em Comform, a ferramenta já é capaz de tratar injeção de falhas em aplicações que usam esse protocolo para o caso de *testes de caixa preta*. Para *testes de caixa branca*, o injetor pode ser estendido para prover as mesmas facilidades para ativação de falhas como as já providas para aplicações Java que usam RMI.

A expansão de Comform para um novo protocolo exige o estudo detalhado da API do protocolo de interesse. Esse estudo deve, em primeiro lugar, identificar se o protocolo de interesse faz uso das classes de UDP ou TCP já instrumentadas por Comform. Caso positivo, o segundo passo é identificar classes, e seus respectivos métodos responsáveis

por envio e recebimento de mensagens, que estejam em um nível de abstração mais alto do que as classes de UDP e TCP. As mensagens do protocolo de interesse devem ser adequadamente modeladas através de uma classe que estenda a classe abstrata `Message`, apresentada na Figura 5.6. Por fim, os métodos de interesse devem ser cuidadosamente instrumentados de modo que as mensagens de interesse sejam processadas.

É preciso tomar cuidado especial na emulação de falhas de temporização. Conforme explicado na Seção 5.2.2, somente as mensagens dos protocolos UDP e TCP, que são base para os protocolos de mais alto nível, é que necessitam sofrer o atraso. As mensagens interceptadas que dizem respeito somente aos protocolos de mais alto nível não devem ser atrasadas, pois, em um nível mais baixo da pilha de protocolos, elas são convertidas em envio e recebimento de mensagens TCP ou UDP e seu atraso levaria à redundância.

5.8 Protótipo de Comform

Atualmente, Comform pode ser utilizado em plataformas Linux e requer o IPTables. Porém, conforme tratado na Seção 5.6, a ferramenta é estensível para outros *firewalls* e, consequentemente, para outros sistemas operacionais, sendo necessário programar um módulo para o *firewall* de interesse. Para que o *firewall* IPTables possa ser utilizado e configurado pelo injetor, Comform deve ser executado em uma conta com permissões de superusuário, já que usuários comuns não tem autorização para configurar o IPTables.

Para configuração de Comform, a ferramenta dispõe do arquivo de configuração *faultinjector.properties*. Neste arquivo, o engenheiro de testes fornece valores para propriedades relativas a, entre outros interesses, *logging* do injetor de falhas, nomes das classes instrumentadas que não são parte padrão da plataforma Java, nome do Módulo de Carga de Falhas a ser usado no experimento, nome da Fábrica de Distribuições de Probabilidade a ser usada e o nome da arquitetura de *firewall*. Em relação a *logging*, um exemplo de propriedade é o diretório onde os *logs* devem ser armazenados. Quanto às propriedades de nomes das classes instrumentadas que não são parte padrão da plataforma Java, um exemplo é o nome da classe que implementa a *interface* `java.rmi.server.RemoteRef`. Quanto ao nome do Módulo de Carga de Falhas a ser usado no experimento, ele deve ser atualizado a cada vez que for desejado realizar experimentos com um novo *Faultload*. Em relação ao nome da Fábrica de Distribuições de Probabilidade a ser usada, a opção disponível atualmente é `ColtProbabilityFactory`. Por fim, em relação ao nome da arquitetura de *firewall*, a opção disponível atualmente, conforme abordado anteriormente, é a classe `IPTables`.

Para que o protótipo seja executado para injeção de falhas em uma aplicação alvo, ou seja, para que seja realizada a interceptação do carregamento das classes de interesse e sua instrumentação para interação com o Controlador do injetor de falhas, a aplicação alvo deve ser executada com a especificação do agente de instrumentação entre seus parâmetros. Ainda, algumas variáveis, como diretório de arquivos executáveis (*bin*) do injetor de falhas, a localização dos arquivos *.jar* utilizados pelo injetor e a localização dos Módulos de Carga De Falhas, devem ser incluídas no *PATH* do sistema operacional.

5.9 Conclusões do Capítulo

Este Capítulo tratou da arquitetura e do protótipo de Comform, uma solução para o problema de injeção de falhas em aplicações Java multiprotocolo que visa suprir a necessidade identificada no Capítulo 2. A arquitetura, incluindo a especificação dos métodos

instrumentados, é importante contribuição deste trabalho.

Primeiramente, foi apresentada uma visão geral da arquitetura e, em seguida, foram detalhadas informações sobre as classes instrumentadas e sobre componentes importantes da arquitetura como o Filtro de Mensagens, o Monitor e modelagem de Falhas e Cargas de Falhas. Também foi analisada a portabilidade de Comform e apresentada uma discussão sobre a sua expansão para novos protocolos. Por fim, foram discutidos aspectos práticos sobre o uso do protótipo.

O desenvolvimento da solução levou em consideração os requisitos identificados na Seção 2.3.2 e também o modelo genérico apresentado na Seção 4.1. Relembrando os requisitos identificados, o primeiro deles é que, para injetar falhas de comunicação corretamente em aplicações Java multiprotocolo, é necessário um modo de tratar adequadamente todos os protocolos usados por elas (limitando o escopo da solução, optou-se por tratar aplicações baseadas nos protocolos UDP, TCP e RMI). Este requisito foi tratado em um alto nível de abstração no desenvolvimento do modelo genérico e teve mapeamento para uma solução concreta na Seção 5.1, que apresentou a arquitetura de Comform. Outros requisitos são a capacidade de emular os tipos de falhas de comunicação comuns em ambientes de rede, a preservação do código fonte da aplicação alvo, a capacidade de realização de testes de caixa branca e de caixa preta, o provimento de um mecanismo adequado para descrição de Módulos de Carga de Falhas e portabilidade.

Quanto à capacidade de emular os tipos de falhas de comunicação que podem ocorrer comumente em ambientes de rede, Comform já é superior a ferramentas como FAIL-FCI (HOARAU; TIXEUIL; VAUCHELLES, 2007), mas ainda pode ser melhorado pela inclusão de novos tipos de falhas em sua arquitetura. Quanto à preservação do código fonte da aplicação alvo, este objetivo é atingido plenamente devido ao uso da combinação do pacote `java.lang.instrument` e de Javassist para instrumentação de classes. Quanto à capacidade de injetar falhas tanto independentemente do conhecimento do código fonte da aplicação alvo (para testes de caixa preta) como também levando em consideração o código fonte da aplicação alvo (para testes de caixa branca), este objetivo foi atingido e ficará mais evidente no próximo Capítulo. Quanto ao provimento de um mecanismo adequado para descrição de Módulos de Carga de Falhas, a Seção 5.5.1 apresentou as facilidades de modelagem de falhas em Comform. Por fim, quanto à portabilidade, a Seção 5.6 apresentou as vantagens de Comform nesse sentido.

O próximo Capítulo apresenta a condução de experimentos de injeção de falhas com o protótipo desenvolvido, mostrando a viabilidade do modelo e da arquitetura propostos para o teste de aplicações Java multiprotocolo.

6 EXPERIMENTOS DE INJEÇÃO DE FALHAS

Este Capítulo apresenta experimentos de injeção de falhas conduzidos com Comform. A Tabela 6.1 apresenta algumas informações sobre as configurações dos computadores disponíveis para os experimentos. Todos eles foram utilizados com o sistema operacional Linux Ubuntu nas versões 8.04 ou 8.10. A primeira coluna da Tabela apresenta o nome de um computador; a segunda, seu modelo; a terceira, seu *clock*; por fim, a quarta, a memória RAM total. Os computadores estão conectados por um *switch* de 100Mbps. A versão 6 da JVM foi utilizada.

Tabela 6.1: Configurações dos computadores disponíveis para experimentos.

nome	**modelo**	**cpu (MHz)**	**memória (KB)**
mercedes	AMD Athlon(tm) processor	1673.784	507720
dkw	Intel Core 2 Duo E4500	2200.000	2067552
jaguar	AMD Athlon(tm) XP 2000+	1673.786	483068
grantorino	AMD Athlon(tm) XP 2400+	1994.397	250884
maverick	AMD Athlon(tm) XP 1800+	1500.000	509500

Algumas aplicações alvo foram escolhidas ou implementadas para realização de experimentos, mas cabe ressaltar aqui que o objetivo desta Seção é mostrar a viabilidade e utilidade do modelo e da arquitetura de Comform. Está fora do escopo deste trabalho avaliar com maiores detalhes essas aplicações. Na Seção 6.1, é apresentado um experimento realizado em uma aplicação sintética baseada em RMI. Na Seção 6.2, é apresentado um experimento realizado tendo Zorilla (DROST; NIEUWPOORT; BAL, 2006) como aplicação alvo. Na Seção 6.3, é apresentado um experimento que tem uma aplicação de demonstração do *middleware* para comunicação de grupo JGroups (BAN, 2002) como alvo.

6.1 Experimento com Aplicação RMI

Este experimento tem dois objetivos: mostrar a habilidade de Comform para injetar falhas em testes de caixa branca e mostrar a correção da estratégia empregada para emulação de colapsos de nodo. O experimento foi conduzido em dois computadores, `mercedes` e `dkw`.

A aplicação alvo baseada em RMI inclui um servidor que implementa um interface remota composta de dois métodos. O primeiro, chamado `multiply`, multiplica duas matrizes recebidas como parâmetro. O segundo, chamado `sum`, soma duas matrizes re-

cebidas como parâmetro. Ambos os métodos retornam o resultado das operações. A aplicação também inclui um cliente que obtém uma referência remota, s, ao servidor visando ser capaz de invocar métodos no servidor. Três matrizes – a, b e c – são declaradas no cliente. A Figura 6.1 ilustra um trecho de código do cliente, mostrando a ordem na qual os métodos remotos são invocados usando essas matrizes como parâmetros.

```
  int [][] d = s.multiply(a, b);
  d = s.multiply(a, c);
  d = s.sum(a, b);
```

Figura 6.1: Um trecho de código do cliente.

A parte da aplicação para o lado servidor foi executada em `dkw`, juntamente com o injetor de falhas. A parte cliente da aplicação foi executada em `mercedes`. Para ilustrar a habilidade de Comform para injetar falhas em pontos específicos da execução de uma aplicação, uma falha de colapso de nodo foi injetada em `dkw` antes da execução da segunda invocação do método `multiply`. Em outras palavras, foi emulado o colapso do servidor depois que o cliente invocou um método, mas antes que o servidor retornasse o resultado ao cliente. Para fins de comparação, o experimento também foi realizado manualmente pela substituição do código do método `multiply` por um laço infinito e remoção do cabo de força do nodo servidor enquanto ele estava executando o laço. O uso do laço infinito como substituto do método `multiply` proveu o tempo necessário para a remoção do cabo de força. A Figura 6.2 mostra o *faultload* desenvolvido para esse teste. A classe `CrashFaultload` estende a classe `Faultload` da API de Comform. A falha é ativada no escopo do método `update`, que recebe como parâmetro mensagens relacionadas aos protocolos sendo utilizados.

```
public class CrashFaultload extends Faultload {
  private CrashFault cf;
  int count = 0;
  public CrashFaultload() throws Exception {
    cf = new CrashFault();}
  public void update(Message msg) throws Exception {
    if (msg.getClass().getName().equals("faultinjector.RMIRequest")){
      if (((RMIRequest)msg).method.getName().equals("multiply") && ((RMIRequest) msg).
          type.equals("RMI_beforeExecuting")) {
        count++;
        if (count==2) cf.activate();}}}
}
```

Figura 6.2: *Faultload* para emulação de colapso.

A situação emulada deixa o cliente esperando que o servidor processe sua requisição. Como a aplicação não é tolerante a falhas, quando um colapso de nodo é injetado, o cliente não é capaz de saber se o servidor ainda está processando a requisição ou se ele sofreu colapso. Após a expiração do *timeout* do TCP, que pode levar um longo tempo dependendo da configuração do cliente, uma exceção é disparada na aplicação cliente indicando que a conexão sofreu *timeout*. Adicionalmente, exatamente a mesma exceção foi capturada tanto no experimento manual como no experimento realizado com Comform, mostrando a adequação da abordagem utilizada para emulação de colapsos de nodo. A Figura 6.3 mostra um trecho da mensagem de exceção.

A realização de um experimento como esse seria inviável com a utilização de um injetor de falhas voltado ao teste de protocolos como FIRMAMENT.

```
Exception in thread "main" java.rmi.UnmarshalException: Error unmarshaling return header
    ; nested exception is:
    java.net.SocketException: Connection timed out
    at sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:209)
    at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:142)
    at java.rmi.server.RemoteObjectInvocationHandler.invokeRemoteMethod(
        RemoteObjectInvocationHandler.java:178)
    at java.rmi.server.RemoteObjectInvocationHandler.invoke(
        RemoteObjectInvocationHandler.java:132)
    at $Proxy0.multiply(Unknown Source)
    at Client.main(Client.java:47)
Caused by: java.net.SocketException: Connection timed out
```

Figura 6.3: Exceção gerada no cliente.

6.2 Experimento com Zorilla

Este experimento tem como propósito demonstrar que Comform pode, de fato, injetar adequadamente falhas de colapso em Zorilla (DROST; NIEUWPOORT; BAL, 2006), uma aplicação multiprotocolo baseada em UDP e TCP, conforme ilustrado na Figura 5.2. A versão *1.0-beta1* de Zorilla, disponível para *download* (VRIJE UNIVERSITEIT, 2007), foi utilizada. Interceptando o carregamento de classes dessa aplicação, constatou-se que ela é baseada nos protocolos TCP e UDP e faz uso tanto das implementações do pacote java.net como do pacote java.nio.

Relembrando a breve descrição apresentada na Seção 2.3.1, Zorilla é um *middleware peer-to-peer*, implementado em Java, que visa à execução de aplicações de supercomputação em grades computacionais. O projeto de Zorilla é baseado em uma rede de nodos ou *peers*. Cada nodo é capaz de tratar submissão, escalonamento e execução de *jobs* e armazenamento de arquivos. Em sua implementação, um nodo faz *broadcast* de pacotes UDP periodicamente na rede local para procurar por outros nodos. O endereço de um ou mais nodos existentes da rede é necessário para se unir a um *overlay*. Através de TCP, nodos se conectam diretamente um a outro para o envio eficiente de grandes quantidades de dados. Também através de TCP, é permitido que programas externos se conectem a Zorilla para, por exemplo, submeter um *job*.

A situação a ser emulada nesse experimento, correspondente à situação apresentada na Figura 5.2, é reapresentada na Figura 6.4 para fins de planejamento, agora com a indicação de qual máquina representa cada nodo no experimento. Na Figura, os nodos `dkw`, `jaguar` e `maverick` constituem um *overlay* Zorilla. O nodo `dkw` sofre uma emulação de colapso, com o uso de Comform, quando o nodo `mercedes` tenta conectar-se a `dkw` para realizar a submissão de um *job*. A seguir, o nodo `grantorino` tenta se unir à rede. Para a emulação correta do colapso, este deve ser percebido por todos os demais nodos do sistema. Como será demonstrado a seguir, ao contrário de ferramentas voltadas à injeção de falhas em aplicações baseadas em um único protocolo, Comform atinge esse objetivo, já que realiza o descarte tanto de mensagens UDP como TCP no nodo `dkw` a partir do momento de ativação da falha.

Para a realização do experimento, inicialmente foi formado um *overlay* Zorilla, constituído pelos nodos `dkw`, `jaguar` e `maverick`. Para a inicialização de Zorilla nos nodos `jaguar` e `maverick`, foi utilizado, em cada máquina, o *script zorilla*, que é responsável pela inicialização e é encontrado no diretório *bin* de Zorilla *1.0-beta1*. A Figura 6.5 ilustra a inicialização de Zorilla nodo `jaguar`, que é semelhante à inicialização realizada no nodo `maverick`. Como pode ser observado na última linha da figura, a execução do *script zorilla* teve sucesso e Zorilla foi inicializado. Para a inicialização de Zorilla no

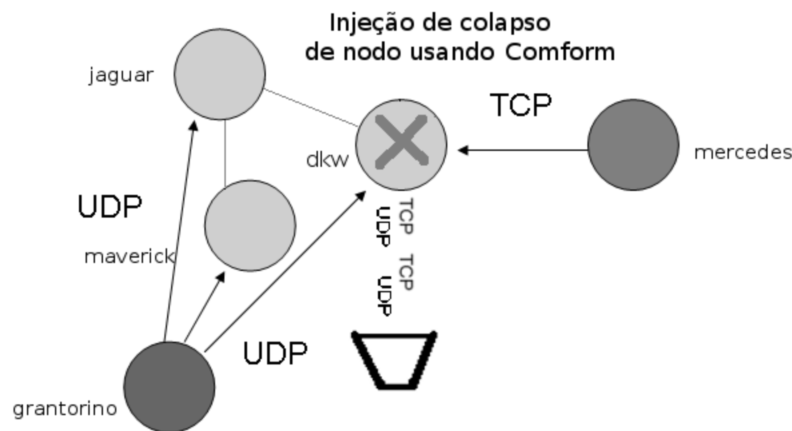


Figura 6.4: Planejamento de injeção de falha de colapso em um nodo Zorilla usando Comform.

nodo `dkw`, que é aquele onde Comform é executado com o propósito de injeção de uma falha de colapso, foi desenvolvido um novo *script* que alia as informações do *script zorilla* às informações necessárias para a instanciação de uma aplicação juntamente com Comform.

```
ccmenegotto@jaguar:~/Desktop/apsteste/zorilla-1.0-beta1/bin$ ./zorilla
22:47:16 WARN  [main] ibis.zorilla.Config - Could not open config file: /home/
    ccmenegotto/.zorilla/zorilla.properties
22:47:16 INFO  [main] ibis.zorilla.job.JobService - Maximum of workers on this node: 1
22:47:16 INFO  [main] ibis.zorilla.job.JobService - Total memory available: 471 Mb
22:47:16 INFO  [main] ibis.zorilla.job.JobService - Total diskspace available: 16350 Mb
22:47:16 INFO  [main] ibis.zorilla.net.DiscoveryService - Started Discovery service
22:47:16 INFO  [main] ibis.zorilla.net.UdpDiscoveryService - Started UDP Discovery
    service on port 5444
22:47:16 INFO  [main] ibis.zorilla.gossip.ARRG - Started ARRG Gossip service
22:47:16 INFO  [main] ibis.zorilla.cluster.VivaldiService - Started Vivaldi service
22:47:16 INFO  [main] ibis.zorilla.cluster.NearestNodeClustering - Started Cluster
    service (nearest neighbour algorithm)
22:47:16 INFO  [main] ibis.zorilla.net.FloodService - Started Flood service
22:47:16 INFO  [main] ibis.zorilla.job.JobService - Started Job service
22:47:16 INFO  [main] ibis.zorilla.net.Network - Started accepting connections on
    143.54.10.93-5444
22:47:16 INFO  [main] ibis.zorilla.net.ZoniService - Zoni Service starter
22:47:16 INFO  [main] ibis.zorilla.www.WebService - Webinterface available on http
    ://143.54.10.93:5446
22:47:16 INFO  [main] ibis.zorilla.Node - Read configuration from /home/ccmenegotto/.
    zorilla
22:47:16 INFO  [main] ibis.zorilla.Node - Saving statistics and logs to /home/
    ccmenegotto/.zorilla/logs
22:47:16 INFO  [main] ibis.zorilla.Node - Saving temporary files to /tmp/zorilla
22:47:16 INFO  [main] ibis.zorilla.Node - Node jaguar:5444 started
```

Figura 6.5: Inicialização do nodo `jaguar`.

O *faultload* desenvolvido para o experimento, apresentado na Figura 6.6, ativa uma falha de colapso de nodo em `dkw` quando da tentativa de conexão feita por `mercedes` com o objetivo de submissão de um *job*. A falha de colapso é ativada no escopo do método `update`, que recebe como parâmetro as mensagens relacionadas aos protocolos sendo utilizados pela aplicação alvo. Quando uma mensagem TCP do tipo (atributo *type* de `TCPmessage`) "TCP_implAccept_after" proveniente de `mercedes` é recebida, a falha de colapso é ativada. O tipo "TCP_implAccept_after" foi utilizado porque, ao fim

do corpo do método `implAccept`, o endereço do nodo que está tentando a conexão já é conhecido, de modo que é possível ativar a falha exatamente quando da tentativa de conexão realizada por `mercedes`.

```
public class CrashFaultloadImplZorilla extends Faultload {
        CrashFault cf;
        int count = 0;
        public CrashFaultloadImplZorilla() throws Exception {
                cf = new CrashFault();
        }
        public void update(Message msg) throws Exception {
                if (msg.getClass().getName().equals("faultinjector.TCPmessage")){
                        if (((TCPmessage)msg).type.equals("TCP_implAccept_after") &&
                        (((TCPmessage)msg).remoteHostIpAddress).toString().equals("
                            /143.54.10.157")){
                                cf.activate();
                        }
                }
        }
}
```

Figura 6.6: *Faultload* para emulação de colapso de nodo em `dkw`.

Depois de formado o *overlay* Zorilla, uma tentativa de submissão de *job* no nodo `dkw` é realizada pelo nodo `mercedes`. Tal tentativa é realizada com o uso do *script zubmit*, que é responsável pela submissão de *jobs* a um nodo Zorilla e é encontrado no diretório *bin* de Zorilla *1.0-beta1*. Para a tentativa de submissão, o nodo `mercedes` procura, primeiramente, estabelecer uma conexão com o nodo `dkw`. Porém, a falha de colapso é ativada em `dkw` quando desta tentativa e a exceção apresentada na Figura 6.7 é gerada em `mercedes`.

```
ccmenegotto@mercedes:~/Desktop/apsteste/zorilla-1.0-beta1/bin$ ./zubmit -na
    143.54.10.205:5444 -c 1 .. ../../satin-2.1/examples/lib/satin-examples.jar
exception on running job: java.net.ConnectException: Connection setup failed
java.net.ConnectException: Connection setup failed
        at ibis.smartsockets.direct.DirectSocketFactory.createSingleSocket(
            DirectSocketFactory.java:1360)
        at ibis.smartsockets.direct.DirectSocketFactory.createSocket(DirectSocketFactory
            .java:1432)
        at ibis.smartsockets.direct.DirectSocketFactory.createSocket(DirectSocketFactory
            .java:1300)
        at ibis.zorilla.zoni.ZoniConnection.<init>(ZoniConnection.java:50)
        at ibis.zorilla.apps.Zubmit.main(Zubmit.java:278)
```

Figura 6.7: Exceção gerada no nodo `mercedes`.

Logo após a ativação da falha, o nodo `grantorino` tenta se unir ao *overlay*. Para tal, é executado o *script zorilla*. Inicialmente, o nodo apresenta um comportamento de inicialização igual ao mostrado na Figura 6.5. A seguir, o comportamento do nodo é o apresentado na Figura 6.8, que mostra que o nodo `grantorino` consegue se comunicar com os nodos `jaguar` e `maverick`, mas gera exceções ao tentar se comunicar com o nodo `dkw`, onde o colapso está sendo emulado. Os nodos `jaguar` e `maverick` também percebem o colapso de `dkw` e exceções do tipo `java.net.SocketTimeoutException` foram registradas em seus *logs* a cada tentativa de comunicação com `dkw`.

Após a realização do experimento, a análise dos *logs* de Comform mostrou que as seguintes classes utilizadas por Zorilla, responsáveis por comunicação usando UDP ou TCP, foram instrumentadas no nodo `dkw`:

```
22:48:30 DEBUG [vivaldi] ibis.zorilla.cluster.VivaldiService - distance to jaguar
    :5444(4-1a5d-47de-bab0-de8d17a7e7b9) is 0.253105 ms
22:48:40 DEBUG [vivaldi] ibis.zorilla.cluster.VivaldiService - distance to maverick
    :5444(b-36f8-4563-b7aa-321ac5d1ec6f) is 0.137727 ms
22:48:50 DEBUG [vivaldi] ibis.zorilla.cluster.VivaldiService - distance to maverick
    :5444(b-36f8-4563-b7aa-321ac5d1ec6f) is 0.140801 ms
22:49:00 DEBUG [vivaldi] ibis.zorilla.cluster.VivaldiService - distance to maverick
    :5444(b-36f8-4563-b7aa-321ac5d1ec6f) is 0.14052 ms
22:49:10 DEBUG [vivaldi] ibis.zorilla.cluster.VivaldiService - distance to jaguar
    :5444(4-1a5d-47de-bab0-de8d17a7e7b9) is 0.251149 ms
22:49:20 DEBUG [vivaldi] ibis.zorilla.cluster.VivaldiService - distance to jaguar
    :5444(4-1a5d-47de-bab0-de8d17a7e7b9) is 0.25087 ms
22:49:40 DEBUG [vivaldi] ibis.zorilla.cluster.VivaldiService - error on pinging
    neighbour dkw:5444(c-8628-408d-b466-9479dbc4b38c)
java.net.SocketTimeoutException: connect timed out
  at java.net.PlainSocketImpl.socketConnect(Native Method)
  at java.net.PlainSocketImpl.doConnect(PlainSocketImpl.java:333)
  at java.net.PlainSocketImpl.connectToAddress(PlainSocketImpl.java:195)
  at java.net.PlainSocketImpl.connect(PlainSocketImpl.java:182)
  at java.net.SocksSocketImpl.connect(SocksSocketImpl.java:366)
  at java.net.Socket.connect(Socket.java:525)
  at ibis.smartsockets.direct.DirectSocketFactory.attemptConnection(DirectSocketFactory.
      java:714)
  at ibis.smartsockets.direct.DirectSocketFactory.createSingleSocket(DirectSocketFactory
      .java:1336)
  at ibis.smartsockets.direct.DirectSocketFactory.createSocket(DirectSocketFactory.java
      :1432)
  at ibis.smartsockets.direct.DirectSocketFactory.createSocket(DirectSocketFactory.java
      :1300)
  at ibis.zorilla.net.Network.connect(Network.java:106)
  at ibis.zorilla.cluster.VivaldiService.ping(VivaldiService.java:49)
  at ibis.zorilla.cluster.VivaldiService.run(VivaldiService.java:143)
  at ibis.util.ThreadPool$PoolThread.run(ThreadPool.java:120)
22:49:40 DEBUG [vivaldi] ibis.zorilla.cluster.VivaldiService - distance to jaguar
    :5444(4-1a5d-47de-bab0-de8d17a7e7b9) is 0.252826 ms
22:49:50 DEBUG [vivaldi] ibis.zorilla.cluster.VivaldiService - distance to jaguar
    :5444(4-1a5d-47de-bab0-de8d17a7e7b9) is 0.248914 ms
```

Figura 6.8: Comportamento do nodo `grantorino` após inicialização.

- `java.net.ServerSocket`;
- `sun.nio.ch.DatagramChannelImpl`;
- `java.net.DatagramSocket`;
- `java.net.Socket`;
- `java.net.SocketInputStream`;
- `java.net.SocketOutputStream`.

O experimento também foi realizado algumas vezes manualmente visando à comparação dos resultados com aqueles obtidos na emulação realizada com Comform. Para esse propósito, a instrumentação do método `implAccept` foi alterada em Comform para que, quando do recebimento de uma mensagem do tipo "TCP_implAccept_after" proveniente de `mercedes`, fosse executado um laço infinito. Tal laço infinito forneceu o tempo necessário para a remoção do cabo de força do nodo `dkw` no contexto da execução do método `implAccept` neste nodo, logo após a entrada do comando *zubmit* em um terminal de `mercedes`. A exceção obtida no nodo `mercedes` foi exatamente igual à obtida com a realização do experimento utilizando Comform. Quanto aos nodos `jaguar` e `maverick`, o resultado obtido foi semelhante, mas, no lugar de exceções de nome

java.net.SocketTimeoutException, que haviam sido obtidas no experimento realizado com Comform, os experimentos manuais resultaram, em sua maioria, em algumas (de zero a 3) exceções java.net.SocketTimeoutException seguidas de exceções de nome java.net.NoRouteToHostException. Já em grantorino, onde Zorilla foi instanciado somente após a ativação da falha, só foram registradas exceções java.net.NoRouteToHostException nos experimentos manuais. Porém, apesar do nome diferente dessa nova exceção registrada nos experimentos manuais, observou-se que todas as demais informações, referentes à origem da exceção, são exatamente iguais às obtidas no experimento realizado com Comform, de modo que o colapso é emulado com precisão adequada. Uma investigação mais detalhada, baseada em outro experimento, no qual o colapso em dkw foi emulado utilizando-se regras do IPTables responsáveis pelo descarte de todos os pacotes recebidos e enviados sem que isso resultasse na geração desse tipo de exceção em mercedes, levou à conclusão de que a geração de exceções NoRouteToHostException só seria viável emulando-se falhas em um nível de abstração mais baixo que o do IPTables.

Como demonstrado, todos os nodos envolvidos no experimento perceberam o colapso de dkw e a falha foi emulada de modo consistente. O mesmo não ocorreria caso um injetor de falhas voltado ao teste de aplicações Java baseadas em um único protocolo (FARCHI; KRASNY; NIR, 2004) (JACQUES-SILVA et al., 2006) (SILVEIRA; WEBER, 2006) (GERCHMAN; WEBER, 2006) tivesse sido utilizado para a realização do experimento. Cabe lembrar também que não foram encontrados registros, na literatura, de injetores de falhas que tratem da instrumentação do pacote java.nio, de modo que Comform é pioneiro nesse sentido.

6.3 Experimento com JGroups

Este experimento mostra um exemplo de aplicação de Comform para o teste de uma aplicação multiprotocolo. Uma aplicação de demonstração do JGroups 2.7.0 (BAN, 2002), chamada Draw, foi usada como sistema alvo. Uma aplicação similar foi testada anteriormente usando FIONA (JACQUES-SILVA et al., 2006), mas, naquele caso, ela precisou ser adaptada para usar somente UDP, já que FIONA injeta falhas em aplicações Java baseadas exclusivamente em UDP. Interceptando o carregamento de classes dessa aplicação, constatou-se que ela é baseada nas implementações do pacote java.net para os protocolos TCP e UDP. A aplicação implementa desenho cooperativo usando comunicação de grupo. Um quadro para desenho local é criado em cada nodo onde a aplicação é executada e os desenhos feitos em um quadro são propagados aos quadros de todos os membros do grupo. A visão do grupo deve ser atualizada sempre que membros deixarem o grupo ou tornarem-se indisponíveis devido a defeitos.

Uma instância da aplicação foi executada em mercedes e outra em dkw. Colapsos de *link* foram emulados entre esses nodos. Comform foi executado em dkw juntamente com uma das instâncias.

O *faultload* desenvolvido para esse experimento é apresentado na Figura 6.9. Trata-se de um exemplo de *faultload* que é útil para teste de caixa preta e que pode ser reusado para o teste de outras aplicações. Este *faultload* define que falhas de colapso de *link* ocorrem durante toda a execução da aplicação com um tempo entre falhas variando entre 3s e 6s de acordo com uma distribuição uniforme de probabilidade. Cada ativação dura entre 2s e 3s, também de acordo com uma distribuição uniforme de probabilidade.

A Figura 6.10 mostra o comportamento da aplicação em uma execução de teste. A

```
public class LinkCrashFaultloadImpl extends Faultload {
  public LinkCrashFaultloadImpl() throws Exception {
    RNG tbf =
ProbabilityFactory.getInstance().createRNG(ProbabilityFactory.UNIFORM, "3 6".split(" "),
      0);
    RNG ttr =
ProbabilityFactory.getInstance().createRNG(ProbabilityFactory.UNIFORM, "2 3".split(" "),
      0);
    LinkCrashFault lc = new LinkCrashFault("mercedes".split(" "), new
        TimeRepetitionPattern(tbf,ttr));
  lc.activate();
  }
  public void update(Message msg) throws Exception {}
}
```

Figura 6.9: *Faultload* para emulação de colapsos de *link*.

primeira linha da Figura mostra que um desenho feito no quadro de `dkw` foi corretamente propagado para o quadro de `mercedes`. Ela também mostra o número dois ao fim da barra de títulos de ambas as janelas, indicando que cada nodo sabe que o grupo possui dois membros. A segunda linha da Figura mostra que um novo desenho feito no quadro de `dkw` não foi propagado para o quadro de `mercedes` devido à ativação de colapso de *link*. Observando o texto da barra de títulos, percebe-se que o protocolo de *group membership* reagiu corretamente à ativação da falha. Cada nodo agora apenas considera ele próprio como membro do grupo. Cabe ressaltar aqui que, como o *faultload* é probabilístico, os comportamentos podem variar de uma execução a outra. Por exemplo, em algumas execuções de testes, a falha foi ativada antes que o primeiro desenho fosse feito. Como esses eram os comportamentos esperados, a emulação está correta.

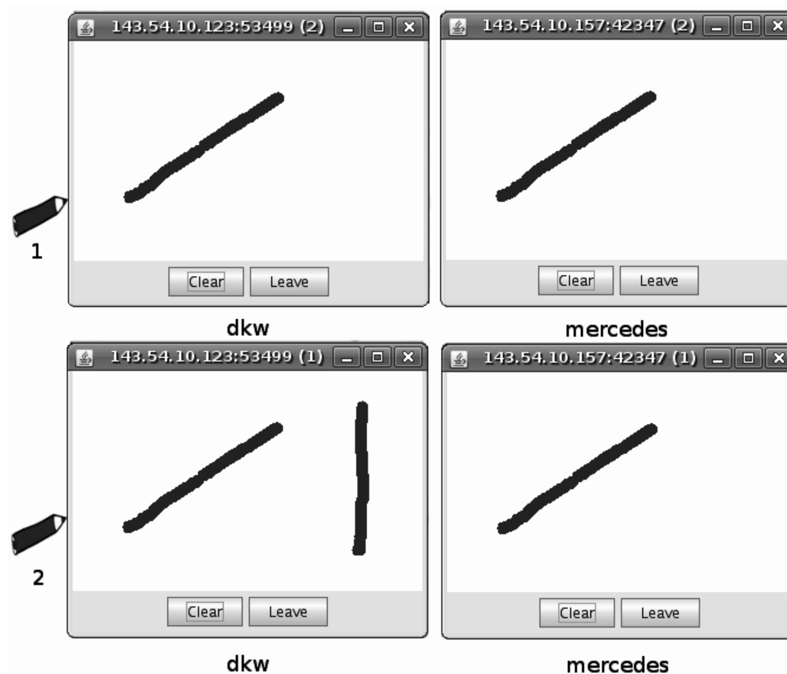


Figura 6.10: Injeção de colapsos de *link*.

6.4 Conclusões do Capítulo

Este Capítulo apresentou experimentos de injeção de falhas de comunicação, conduzidos utilizando Comform, em diferentes aplicações alvo. Tais experimentos mostraram a viabilidade e a utilidade do modelo e da arquitetura de Comform.

O primeiro experimento apresentado teve como alvo uma aplicação baseada em Java RMI e mostrou a habilidade de Comform para injetar falhas em testes de caixa branca. No exemplo apresentado, uma falha de colapso foi ativada quando da segunda invocação de um método remoto específico. Esse tipo de experimento dificilmente poderia ser conduzido com a utilização de injetores de falhas voltados ao teste de protocolos como FIRMAMENT.

O segundo experimento mostrou a capacidade de Comform para injeção de falhas na aplicação multiprotocolo Zorilla. O experimento realizado não poderia ser conduzido com injetores de falhas voltados ao teste de aplicações baseadas em um único protocolo.

Por fim, no terceiro experimento, Comform foi utilizado para injeção de falhas de colapso de *link* em uma aplicação multiprotocolo do *middleware* para comunicação de grupo JGroups.

7 CONSIDERAÇÕES FINAIS

7.1 Conclusões

Aplicações de rede com altos requisitos de dependabilidade devem ser testadas cuidadosamente em condições de falhas de comunicação, que ocorrem comumente em ambientes de rede, para aumentar a confiança no seu comportamento apropriado na ocorrência de falhas. Injeção de falhas de comunicação é a técnica mais apropriada para o teste dos mecanismos de tolerância a falhas destas aplicações em presença de falhas de comunicação. Ela é útil tanto para auxiliar na remoção de falhas como na previsão de falhas.

As aplicações de rede baseadas em mais de um protocolo da arquitetura TCP/IP são denominadas multiprotocolo no contexto desse trabalho. Este tipo de aplicação é comum, pois diferentes protocolos são adequados a diferentes propósitos. Assim como qualquer outra aplicação de rede com altos requisitos de dependabilidade, as aplicações Java multiprotocolo que possuem tais requisitos devem operar conforme sua especificação em presença de falhas. Visando atingir esse objetivo, elas devem ser testadas adequadamente em condições de falhas de comunicação. Caso a emulação de uma falha que afete a troca de mensagens não leve em consideração todos os protocolos simultaneamente utilizados, o comportamento emulado durante um experimento poderá ser diferente daquele observado na ocorrência de uma falha real, de modo que podem ser obtidos resultados inconsistentes sobre o comportamento da aplicação alvo em presença da falha.

Muitos injetores de falhas de comunicação encontrados na literatura não são capazes de testar aplicações Java multiprotocolo. Outros possuem potencial para o teste dessas aplicações, mas impõem grandes dificuldades aos engenheiros de testes. Por exemplo, contrariamente ao enfoque deste trabalho, algumas ferramentas são voltadas à injeção de falhas de comunicação para o teste de protocolos de comunicação, e não de aplicações. Tal orientação ao teste de protocolos costuma levar a grandes dificuldades no *teste de caixa branca* de aplicações. Entre outros exemplos de dificuldades proporcionadas por ferramentas da literatura estão a incapacidade de injetar falhas diretamente em aplicações Java e a limitação quanto aos tipos de falhas que permitem emular. A análise de tais ferramentas motivou o desenvolvimento de uma solução voltada especificamente ao teste de aplicações multiprotocolo desenvolvidas em Java.

Entre os requisitos identificados para a solução, o primeiro deles é a habilidade de injetar falhas de comunicação corretamente em aplicações Java multiprotocolo. Outros requisitos são a capacidade de emular os tipos de falhas de comunicação comuns em ambientes de rede, a preservação do código fonte da aplicação alvo, a capacidade de realização de testes de caixa branca e de caixa preta, o provimento de um mecanismo adequado para descrição de Módulos de Carga de Falhas e portabilidade.

Este trabalho apresentou uma solução para injeção de falhas de comunicação em apli-

cações Java multiprotocolo. A solução opera no nível da JVM, interceptando mensagens de protocolos, e, em alguns casos, opera também no nível do sistema operacional, usando regras de *firewall* para emulação de alguns tipos de falhas que não podem ser emulados somente no nível da JVM. A abordagem atende aos requisitos identificados para a solução, como ser útil para testes de caixa branca e preta e não modificar o código fonte da aplicação alvo.

Comform, um protótipo para injeção de falhas de comunicação em aplicações Java multiprotocolo, foi desenvolvido com base no modelo e na arquitetura propostos para a solução. Atualmente, Comform pode ser aplicado para testar aplicações Java baseadas em qualquer combinação dos protocolos UDP, TCP e RMI (incluindo aquelas baseadas em um único protocolo).

Experimentos de injeção de falhas foram conduzidos em diferentes aplicações alvo utilizando Comform. Tais experimentos mostraram a viabilidade e a utilidade do modelo e da arquitetura de Comform.

7.2 Trabalhos Futuros

Os principais trabalhos futuros incluem a continuação do desenvolvimento da arquitetura e do protótipo de Comform visando lidar com outros tipos de falhas e outros protocolos de comunicação.

Considerando outros tipos de falhas, seria importante implementar falhas de *omissão de envio* e *omissão de recepção* de mensagens e investigar alternativas para implementação de particionamento de rede. Este último tipo de falha requer um projeto cuidadoso, com coordenação da falha de forma distribuída, para que a emulação seja correta. Sua implementação foi tratada de diferentes maneiras em projetos como Loki (CHANDRA et al., 2004) (LEFEVER; CUKIER; SANDERS, 2003) e FIONA (JACQUES-SILVA et al., 2006).

Considerando outros protocolos de comunicação, conforme mencionado na Seção 5.7, que provê indicações sobre como realizar a expansão de Comform para novos protocolos, seria útil lidar com outros protocolos largamente utilizados que sejam baseados em UDP ou TCP. Comform já é capaz de tratar esses protocolos para o caso de *testes de caixa preta*. Para *testes de caixa branca*, o injetor pode ser estendido para prover as mesmas facilidades para ativação de falhas como as já providas para o teste de aplicações Java que usam RMI.

REFERÊNCIAS

AVIZIENIS, A. et al. Basic Concepts and Taxonomy of Dependable and Secure Computing. **IEEE Transactions on Dependable and Secure Computing**, Los Alamitos, CA, USA, v.1, n.1, p.11–33, Jan. 2004.

BALATON, Z. et al. From Cluster Monitoring to Grid Monitoring Based on GRM. In: EUROPEAN CONFERENCE ON PARALLEL PROCESSING, EURO-PAR, 7., 2001, Manchester, UK. **Proceedings...** London: Springer-Verlag, 2001. p.874–881. (Lecture Notes in Computer Science, v.2150).

BAN, B. **JGroups - A Toolkit for Reliable Multicast Communication**. 2002. Disponível em: <http://www.jgroups.org>. Acesso em: jan. 2009.

BARBOSA, R. et al. Verification and Validation of (Real Time) COTS Products using Fault Injection Techniques. In: INTERNATIONAL IEEE CONFERENCE ON COMMERCIAL-OFF-THE-SHELF (COTS)-BASED SOFTWARE SYSTEMS, ICCBSS, 6., 2007, Banff, Alberta, Canada. **Proceedings...** Los Alamitos: IEEE Computer Society, 2007. p.233–242.

BINDER, W.; HULAAS, J.; MORET, P. Advanced Java Bytecode Instrumentation. In: INTERNATIONAL SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PROGRAMMING IN JAVA, PPPJ, 5., 2007, Lisboa, Portugal. **Proceedings...** New York: ACM, 2007. p.135–144.

BIRMAN, K. P. **Building Secure and Reliable Network Applications**. 1st.ed. Greenwich: Manning Publications, Co., 1996.

CERN. **Colt Project**. 1999. Disponível em: <http://acs.lbl.gov/~hoschek/colt/>. Acesso em: set. 2009.

CÉZANE, D. et al. Um Injetor de Falhas para a Avaliação de Aplicações Distribuídas Baseadas no Commune. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS, SBRC, 27., 2009, Recife. **Anais...** Porto Alegre: SBC, 2009. v.1, p.901–914.

CHANDRA, R. et al. A Global-State-Triggered Fault Injector for Distributed System Evaluation. **IEEE Transactions on Parallel and Distributed Systems**, Los Alamitos, CA, USA, v.15, n.7, p.593–605, Jul. 2004.

CHIBA, S. **Javassist**. 1998. Disponível em: <http://www.csg.is.titech.ac.jp/~chiba/javassist/>. Acesso em: jan. 2009.

CHIBA, S. Load-Time Structural Reflection in Java. In: EUROPEAN CONFERENCE OBJECT-ORIENTED PROGRAMMING, ECOOP, 14., 2000, Sophia Antipolis and Cannes, France. **Proceedings. . .** Berlin/Heidelberg: Springer, 2000. p.313–336. (Lecture Notes in Computer Science, v.1850).

CHIBA, S. **Getting Started with Javassist**. 2007. Disponível em: <http://www.csg.is.titech.ac.jp/~chiba/javassist/tutorial/tutorial.html>. Acesso em: jan. 2009.

CIRNE, W. et al. Labs of the World, Unite!!! **Journal of Grid Computing**, Netherlands, v.4, n.3, p.225–246, Sept. 2006.

CLARK, J. A.; PRADHAN, D. K. Fault Injection: a method for validating computer-system dependability. **Computer**, Los Alamitos, CA, EUA, v.28, n.6, p.47–56, Jun. 1995.

CONSORTIUM, O. **ASM**. 1999. Disponível em: <http://asm.ow2.org/>. Acesso em: jan. 2009.

CRISTIAN, F. Understanding Fault-Tolerant Distributed Systems. **Communications of the ACM**, New York, NY, USA, v.34, n.2, p.56–78, Feb. 1991.

DAWSON, S. et al. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, FTCS, 26., 1996, Sendai, Japan. **Proceedings. . .** Los Alamitos: IEEE Computer Society, 1996. p.404–414.

DAWSON, S.; JAHANIAN, F.; MITTON, T. ORCHESTRA: a probing and fault injection environment for testing protocol implementations. In: IEEE INTERNATIONAL COMPUTER PERFORMANCE AND DEPENDABILITY SYMPOSIUM, IPDS, 2., 1996, Urbana-Champaign, IL, USA. **Proceedings. . .** Los Alamitos: IEEE Computer Society, 1996. p.56.

DAWSON, S.; JAHANIAN, F.; MITTON, T. Experiments on Six Commercial TCP Implementations Using a Software Fault Injection Tool. **Software: Practice and Experience**, New York, NY, USA, v.27, n.12, p.1385–1410, 1997.

DREBES, R. J. **FIRMAMENT**: um módulo de injeção de falhas de comunicação para linux. 2005. 87 f. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

DROST, N.; NIEUWPOORT, R. V. van; BAL, H. Simple Locality-Aware Co-allocation in Peer-to-Peer Supercomputing. In: IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID WORKSHOPS, CCGRIDW, 6., 2006, Singapore. **Proceedings. . .** Los Alamitos: IEEE Computer Society, 2006. v.2.

FARCHI, E.; KRASNY, Y.; NIR, Y. Automatic simulation of network problems in UDP-based Java programs. In: INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM, IPDPS, 18., 2004, Santa Fe, New Mexico. **Proceedings. . .** Los Alamitos: IEEE Computer Society, 2004. p.267–.

FOSTER, I. Globus Toolkit Version 4: software for service-oriented systems. **Journal of Computer Science and Technology**, [S.l.], v.21, n.4, p.513–520, 2006.

FOUNDATION, A. S. **BCEL - Byte Code Engineering Library**. 2002. Disponível em: <http://jakarta.apache.org/bcel/>. Acesso em: jan. 2009.

GAMMA, E. et al. **Design Patterns**: elements of reusable object-oriented software. [S.l.]: Addison-Wesley (Addison-Wesley Professional Computing Series)., 1994.

GERCHMAN, J.; MENEGOTTO, C. C.; WEBER, T. S. Geração de cargas de falha para campanhas de injeção de falhas a partir de modelos UML de teste. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 22., 2008, Campinas. **Anais...** Porto Alegre: SBC, 2008. p.17–32.

GERCHMAN, J.; WEBER, T. S. Emulando o Comportamento de TCP/IP em um Ambiente com Falhas para Teste de Aplicações de Rede. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, WTF, 7., 2006, Curitiba. **Anais...** Porto Alegre: SBC, 2006. v.1, p.41–54.

GIL, P. et al. **Fault Representativeness**. 2002. DBench Dependability Benchmarking IST-2000-25425. Deliverable ETIE2.

HAN, S.; SHIN, K. G. Experimental Evaluation of Behavior-Based Failure-Detection Schemes in Real-Time Communication Networks. **IEEE Transactions on Parallel and Distributed Systems**, Los Alamitos, CA, USA, v.10, n.6, p.613–626, Jun. 1999.

HAN, S.; SHIN, K. G.; ROSENBERG, H. A. DOCTOR: an integrated software fault injection environment for distributed real-time systems. In: INTERNATIONAL COMPUTER PERFORMANCE AND DEPENDABILITY SYMPOSIUM, IPDS, 1995, Erlangen, Germany. **Proceedings...** Los Alamitos: IEEE Computer Society, 1995. p.204–213.

HOARAU, W.; TIXEUIL, S.; VAUCHELLES, F. FAIL-FCI: versatile fault injection. **Future Generation Computer Systems**, Amsterdam, The Netherlands, v.23, n.7, p.913–919, Aug. 2007.

HSUEH, M.-C.; TSAI, T. K.; IYER, R. K. Fault Injection Techniques and Tools. **Computer**, Los Alamitos, CA, EUA, v.30, n.4, p.75–82, Apr. 1997.

JACQUES-SILVA, G. **Injeção Distribuída de Falhas para Validação de Dependabilidade de Sistemas Distribuídos de Larga Escala**. 2005. 79 f. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

JACQUES-SILVA, G.; DREBES, R. J.; GERCHMAN, J.; TRINDADE, J. M. F.; WEBER, T. S.; JANSCH-PÔRTO, I. A Network-Level Distributed Fault Injector for Experimental Validation of Dependable Distributed Systems. In: INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, COMPSAC, 30., 2006, Chicago, Illinois. **Proceedings...** Los Alamitos: IEEE Computer Society, 2006. p.421–428.

JACQUES-SILVA, G.; O. MORAES, R. L. de; WEBER, T. S.; MARTINS, E. Validando Sistemas Distribuídos Desenvolvidos em Java Utilizando Injeção de Falhas de Comunicação. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, WTF, 5., 2004, Gramado. **Anais...** Porto Alegre: SBC, 2004.

JAIN, R. **The Art of Computer Systems Performance Analysis**: techniques for experimental design, measurement, simulation and modelling. 1st.ed. New York, USA: John Wiley & Sons, Inc., 1991.

JAX SYSTEMS LLC. **DocJar**: search open source java api. 2005. Disponível em: <http://www.docjar.com/>. Acesso em: ago. 2009.

KANOUN, K.; SPAINHOWER, L. **Dependability Benchmarking for Computer Systems**. Los Alamitos, USA: Wiley-IEEE Computer Society Press, 2008.

KICKZALES, G. et al. Aspect Oriented Programming. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, ECOOP, 1997, Finland. **Proceedings...** Berlin/Heidelberg: Springer, 1997. p.220–242. (Lecture Notes in Computer Science).

LEFEVER, R.; CUKIER, M.; SANDERS, W. An experimental evaluation of correlated network partitions in the Coda distributed file system. In: INTERNATIONAL SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, SRDS, 22., 2003, Florence, Italy. **Proceedings...** Los Alamitos: IEEE Computer Society, 2003. p.273–282.

LEITE, F. O. **ComFIRM**: injeção de falhas de comunicação através de alteração de recursos do sistema operacional. 2000. 117 f. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

LOOKER, N.; MUNRO, M.; XU, J. WS-FIT: a tool for dependability analysis of web services. In: ANNUAL INTERNATIONAL COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, COMPSAC, 28., 2004, Hong Kong. **Proceedings...** IEEE Computer Society, 2004. v.2, p.120–123.

LOOKER, N.; XU, J. Dependability Assessment of Grid Middleware. In: ANNUAL IEEE/IFIP INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 37., 2007, Edinburgh, Scotland. **Proceedings...** IEEE Computer Society, 2007. p.125–130.

MARTINS, E.; RUBIRA, C.; LEME, N. Jaca: a reflective fault injection tool based on patterns. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 2002, Washington, DC, USA. **Proceedings...** Los Alamitos: IEEE Computer Society, 2002. p.483–487.

MASIERO, P. C. et al. Teste de Software Orientado a Objetos e a Aspectos: teoria e prática. In: BREITMAN, K.; ANIDO, R. (Ed.). **Atualizações em Informática**. Rio de Janeiro: Editora PUC-Rio: SBC, 2006. p.13–71.

MASSIE, M. L.; CHUNB, B. N.; CULLER, D. E. **The Ganglia Distributed Monitoring System**: design, implementation and experience. Berkeley, CA: University of California, 2003.

MENEGOTTO, C. C.; VACARO, J. C.; WEBER, T. S. Injeção de Falhas de Comunicação em Grids com Características de Tolerância a Falhas. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, WTF, 8., 2007, Belém. **Anais...** Porto Alegre: SBC, 2007. p.71–84.

MENEGOTTO, C. C.; WEBER, T. S. Injeção de Falhas de Comunicação em Aplicações Multiprotocolo. In: WORKSHOP DE TESES E DISSERTAÇÕES EM ENGENHARIA DE SOFTWARE, WTES, 13., 2008, Campinas. **Anais...** [S.l.: s.n.], 2008. p.1–6.

MENEGOTTO, C. C.; WEBER, T. S. A Practical Methodology for Experimental Fault Injection to Test Complex Network-Based Systems. In: IEEE LATIN-AMERICAN TEST WORKSHOP, LATW, 10., 2009, Armação dos Búzios. **Proceedings...** IEEE Computer Society, 2009. p.1–6.

MURRAY, P. A Distributed State Monitoring Service for Adaptive Application Management. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 2005, Yokohama, Japan. **Proceedings...** Los Alamitos: IEEE Computer Society, 2005. p.200–205.

NEVES, N.; FUCHS, W. K. Fault Detection Using Hints from the Socket Layer. In: SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, SRDS, 16., 1997, Durham, North Carolina, USA. **Proceedings...** Los Alamitos: IEEE Computer Society, 1997. p.64–71.

PEZZÉ, M.; YOUNG, M. **Software Testing and Analysis**: process, principles, and techniques. [S.l.]: John Wiley & Sons, 2008.

POSTEL, J. **RFC768 - User Datagram Protocol**. 1980. Disponível em: <http://www.faqs.org/rfcs/rfc768.html>. Acesso em: ago. 2009.

POSTEL, J. **RFC793 - Transmission Control Protocol**. 1981. Disponível em: <http://www.faqs.org/rfcs/rfc793.html>. Acesso em: ago. 2009.

REILLY, D.; REILLY, M. **Java Network Programming and Distributed Computing**. 1st.ed. [S.l.]: Addison-Wesley Professional, 2002.

RUSSEL, R.; WELTE, H. **Linux netfilter Hacking HOWTO**. 2002. Disponível em: <http://www.netfilter.org/documentation>. Acesso em: jan. 2009.

SCHNEIDER, F. B. **What good are models and what models are good?** In: Mullender, S. (Ed). Distributed Systems. 2nd.ed. ACM Press/Addison-Wesley Publishing Co, 1993. p.17–26.

SHIN, K. G. HARTS: a distributed real-time architecture. **Computer**, Los Alamitos, CA, EUA, v.24, n.5, p.25–35, May 1991.

SILVEIRA, K. K. **Uma Estratégia Baseada em Programação Orientada a Aspectos para Injeção de Falhas de Comunicação**. 2005. 75 f. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

SILVEIRA, K. K.; WEBER, T. S. An Aspect Oriented Fault Injection Tool to Test Fault Tolerant Mechanisms of Dependable Java-Based Network Applications. In: IEEE LATIN-AMERICAN TEST WORKSHOP, LATW, 2006, Buenos Aires, Argentina. **Proceedings...** Porto Alegre: Evangraf, 2006. v.1, p.159–164.

SIQUEIRA, T. F. de; FISS, B.; MENEGOTTO, C. C.; CECHIN, S.; WEBER, T. S. Avaliação experimental de estratégias de tolerância a falhas do protocolo SCTP por injeção de falhas. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS, SBRC, 27., 2009, Recife. **Anais...** Porto Alegre: SBC, 2009. p.915–930.

STOTT, D. et al. NFTAPE: networked fault tolerance and performance evaluator. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 2002, Bethesda, Maryland, USA. **Proceedings...** Los Alamitos: IEEE Computer Society, 2002. p.542.

STOTT, D. T. et al. NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. In: INTERNATIONAL COMPUTER PERFORMANCE AND DEPENDABILITY SYMPOSIUM, IPDS, 4., 2000, Chicago, IL, USA. **Proceedings...** Los Alamitos: IEEE Computer Society, 2000. p.91–100.

SUGETA, T.; MALDONADO, J. C.; WONG, W. E. Mutation Testing Applied to Validate SDL Specifications. In: TESTCOM, 16., 2004. **Proceedings...** [S.l.: s.n.], 2004. p.193–208.

SUN MICROSYSTEMS. **Why Developers Should Not Write Programs That Call 'sun' Packages**. 1996. Disponível em: <http://java.sun.com/products/jdk/faq/faq-sun-packages.html>. Acesso em: ago. 2008.

SUN MICROSYSTEMS. **Java Remote Method Invocation**. 2006. Disponível em: <http://java.sun.com/javase/6/docs/platform/rmi/spec/rmiTOC.html>. Acesso em: ago. 2009.

SUN MICROSYSTEMS. **Java Virtual Machine Tool Interface**. 2006. Disponível em: <http://java.sun.com/javase/6/docs/technotes/guides/jvmti/>. Acesso em: mar. 2009.

SUN MICROSYSTEMS. **Package java.lang.instrument**. 2007. Disponível em: <http://java.sun.com/javase/6/docs/technotes/guides/instrumentation/index.html>. Acesso em: mar. 2009.

SUN MICROSYSTEMS. **Java Platform Standard Ed. 6**. 2008. Disponível em: <http://java.sun.com/javase/6/docs/api/>. Acesso em: jan. 2009.

SUN MICROSYSTEMS. **Java Platform Standard Ed. 7**. 2009. Disponível em: <http://java.sun.com/javase/7/docs/api/>. Acesso em: ago. 2009.

TOWNEND, P. et al. CROWN-C: a high-assurance service-oriented grid middleware system. **Computer**, Los Alamitos, CA, USA, v.41, n.8, p.30–38, Aug. 2008.

VACARO, J. C. **Avaliação de Dependabilidade de Aplicações Distribuídas baseadas em RMI através de Injeção de Falhas**. 2007. 89 f. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

VACARO, J. C.; WEBER, T. S. Injeção de Falhas na Fase de Teste de Aplicações Distribuídas. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 20., 2006, Florianópolis. **Anais...** Porto Alegre: SBC, 2006. v.1, p.161–176.

VRIJE UNIVERSITEIT. **Ibis**: grids as promised. 2007. Disponível em: <http://www.cs.vu.nl/ibis/downloads.html>. Acesso em: out. 2009.

WEBER, T.; VACARO, J.; SIQUEIRA, T. de; JANSCH-PORTO, I. Generating non-uniform distributions for fault injection to emulate real network behavior in test campaigns. In: IEEE LATIN-AMERICAN TEST WORKSHOP, LATW, 10., 2009, Armação dos Búzios. **Proceedings...** IEEE Computer Society, 2009. p.1–6.

WHITE, A. **Serp**. 2002. Disponível em: <http://serp.sourceforge.net/>. Acesso em: jan. 2009.

APÊNDICE A HISTÓRICO DE DESENVOLVIMENTO

Um dos focos de trabalho do Grupo de Tolerância da UFRGS é o desenvolvimento de injetores de falhas de comunicação para avaliação da dependabilidade de aplicações distribuídas. Nos últimos anos, o grupo desenvolveu os injetores FIONA (JACQUES-SILVA et al., 2006), FIRMAMENT (DREBES, 2005), FICTA (SILVEIRA; WEBER, 2006) e FIRMI (VACARO; WEBER, 2006) entre outros. O envolvimento da autora na área de injeção de falhas iniciou em 2006, no contexto do seu trabalho de conclusão do curso de Graduação em Ciência da Computação. Na época, constatou-se que, embora o grupo já tivesse empregado muitos esforços no desenvolvimento de injetores de falhas de comunicação, pouco trabalho havia sido realizado na área experimental de utilização dos injetores para avaliação de aplicações.

O foco do trabalho de conclusão, realizado durante o ano de 2006, foi o planejamento e condução de experimentos de injeção de falhas usando o injetor de falhas FIRMI, que encontrava-se em desenvolvimento, como parte do trabalho de mestrado de Juliano Vacaro (VACARO, 2007). Também era objetivo do trabalho a avaliação de FIRMI, com base na experiência obtida com seu uso, apresentando pontos positivos e negativos da ferramenta, e assim proporcionando uma realimentação ao seu desenvolvimento. Como principal contribuição, esse trabalho teve como resultado o desenvolvimento de uma metodologia simples e prática para injeção de falhas. Esta metodologia visa auxiliar engenheiros de teste na realização de uma campanha completa de injeção de falhas, obtendo métricas relevantes de dependabilidade e identificando partes fracas do sistema alvo.

A metodologia foi aplicada na condução de experimentos de injeção de falhas em uma instalação local da plataforma para computação em grade OurGrid (CIRNE et al., 2006), usando o injetor de falhas FIRMI, obtendo resultados sobre o comportamento do alvo em presença de falhas. A metodologia foi desenvolvida com base nos passos para a condução de um estudo de avaliação de desempenho propostos por Jain (JAIN, 1991), que não eram diretamente aplicáveis à injeção de falhas, de modo que foi necessário estender a metodologia original e modificar requisitos de alguns passos. Os passos originais são os seguintes: estabelecer os objetivos do estudo e definir os limites do sistema, listar serviços do sistema e possíveis resultados, selecionar métricas, listar parâmetros de sistema e de carga de trabalho, selecionar fatores e seus valores, selecionar técnicas de avaliação, selecionar a carga de trabalho, projetar os experimentos, analisar e interpretar os dados e, por fim, apresentar os resultados. Um passo de definição da carga de falhas a ser injetada na aplicação sob teste foi introduzido, o qual deve ser cumprido antes do passo de projeto dos experimentos. A noção de parâmetros de carga de falhas foi introduzida ao passo de listagem de parâmetros de sistema e de carga de trabalho. Por fim, a seleção de ferramentas de injeção de falhas foi acrescentada ao passo de seleção de técnicas de avaliação. Este trabalho gerou uma publicação no WTF 2007 (MENEGOTTO; VACARO; WEBER,

2007). Posteriormente, um artigo foi publicado no LATW 2009 (MENEGOTTO; WEBER, 2009), o qual consolida e detalha a metodologia, abordando com mais cuidado tópicos como a seleção do injetor de falhas.

O curso de mestrado teve seu início em 2007 e a proposta de tema para a dissertação de mestrado foi defendida ao final daquele ano. A ideia de focar o tema do trabalho no estudo de injeção de falhas de comunicação em aplicações multiprotocolo partiu do princípio de que aplicações de rede podem ser desenvolvidas com base em mais de um protocolo de comunicação e de que, para injetar adequadamente falhas de comunicação nessas aplicações, as falhas devem afetar todos os protocolos em uso simultâneo. Ainda, várias ferramentas desenvolvidas anteriormente pelo grupo, como FIONA, FIERCE e FIRMI, são voltadas ao teste de aplicações Java baseadas em um único protocolo, de modo que as evidências indicavam não serem adequadas ao teste de aplicações Java multiprotocolo.

Inicialmente, foram investigadas as dificuldades relacionadas à injeção de falhas de comunicação em aplicações que usam mais de um protocolo da arquitetura TCP/IP e impossibilidades de uso de ferramentas existentes. A análise de ferramentas de injeção de falhas encontradas na literatura evidenciou a importância da construção de uma nova ferramenta voltada especificamente ao teste de aplicações multiprotocolo desenvolvidas em Java. Com a conclusão dessas atividades, a contribuição do trabalho ficou bem estabelecida. Consolidando essa etapa, foi realizada a publicação de um artigo no WTES 2008 (MENEGOTTO; WEBER, 2008).

Em seguida, iniciou-se um processo de especificação de uma solução, quando foram feitas importantes decisões de projeto como os tipos de falhas a serem emuladas, estratégia de injeção de falhas, definição dos métodos a serem instrumentados e estratégia para descrição de carga de falhas. Essas etapas foram seguidas da implementação do protótipo Comform e da realização de experimentos de injeção de falhas.

Paralelamente às atividades relacionadas ao tema da dissertação, a autora auxiliou na escrita de artigos sobre outros temas trabalhados pelo grupo. A integração de injeção de falhas em atividades de modelagem de artefatos de teste, que resultou em U2TP-FI, uma extensão do Perfil UML 2.0 de Teste para injeção de falhas, foi tema de um dos artigos, publicado no SBES 2008 (GERCHMAN; MENEGOTTO; WEBER, 2008). A avaliação experimental das estratégias de tolerância a falhas do protocolo SCTP (Stream Control Transmission Protocol) por injeção de falhas foi o tema de outro artigo, publicado no SBRC 2009 (SIQUEIRA et al., 2009). A metodologia apresentada anteriormente (MENEGOTTO; VACARO; WEBER, 2007) foi seguida para a realização desta avaliação e o injetor de falhas FIRMAMENT foi utilizado.

Artigos publicados

MENEGOTTO, C. C.; VACARO, J. C.; WEBER, T. S. Injeção de Falhas de Comunicação em Grids com Características de Tolerância a Falhas. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, WTF, 8., 2007, Belém. **Anais...** Sociedade Brasileira de Computação, 2007. p.71-84.

GERCHMAN, J.; MENEGOTTO, C. C.; WEBER, T. S. Geração de cargas de falha para campanhas de injeção de falhas a partir de modelos UML de teste. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 22., 2008, Campinas. **Anais...** Sociedade Brasileira de Computação, 2008. p.17-32.

MENEGOTTO, C. C.; WEBER, T. S. Injeção de Falhas de Comunicação em Aplica-

ções Multiprotocolo. In: WORKSHOP DE TESES E DISSERTAÇÕES EM ENGENHARIA DE SOFTWARE, WTES, 13., 2008, Campinas. **Anais...** Sociedade Brasileira de Computação, 2008. p.1-6.

MENEGOTTO, C. C.; WEBER, T. S; WEBER, R. F. A Practical Methodology for Experimental Fault Injection to Test Complex Network-Based Systems. In: IEEE LATIN-AMERICAN TEST WORKSHOP, LATW, 10., 2009, Armação dos Búzios. **Proceedings...** IEEE Computer Society, 2009. p.1-6.

SIQUEIRA, T. F.; FISS, B.; MENEGOTTO, C. C.; CECHIN, S. L.; WEBER, T. S. Avaliação experimental de estratégias de tolerância a falhas do protocolo SCTP por injeção de falhas. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS, SBRC, 27., 2009, Recife. **Anais...** Sociedade Brasileira de Computação, 2009. p.915-930.

PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

.inf INSTITUTO DE INFORMÁTICA UFRGS PPGC

"Injeção de Falhas de Comunicação em Aplicações Multiprotocolo."

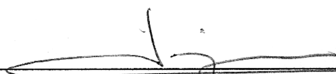
por

Cristina Ciprandi Menegotto

Dissertação Apresentada aos Senhores:

Profa. Dra. Eliane Martins
(UNICAMP)

Prof. Dr. Antônio Marinho Pilla Barcellos
(UFRGS)



Prof. Dr. Ingrid Eleonora Schreiber Jansch Porto
(UFRGS)

Vista e permitida a impressão.
Porto Alegre, 21/12/2005

Prof. Dr. Taisy Silva Weber
Orientadora

Prof. Álvaro Freitas Moreira
Coordenador do Programa de
Pós-Graduação em Computação - PPGC
Instituto de Informática - UFRGS

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Administração
Baixar livros de Agronomia
Baixar livros de Arquitetura
Baixar livros de Artes
Baixar livros de Astronomia
Baixar livros de Biologia Geral
Baixar livros de Ciência da Computação
Baixar livros de Ciência da Informação
Baixar livros de Ciência Política
Baixar livros de Ciências da Saúde
Baixar livros de Comunicação
Baixar livros do Conselho Nacional de Educação - CNE
Baixar livros de Defesa civil
Baixar livros de Direito
Baixar livros de Direitos humanos
Baixar livros de Economia
Baixar livros de Economia Doméstica
Baixar livros de Educação
Baixar livros de Educação - Trânsito
Baixar livros de Educação Física
Baixar livros de Engenharia Aeroespacial
Baixar livros de Farmácia
Baixar livros de Filosofia
Baixar livros de Física
Baixar livros de Geociências
Baixar livros de Geografia
Baixar livros de História
Baixar livros de Línguas

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo