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

RUTHIANO SIMIONI MUNARETTI

Um Ambiente para Descrição de Cenários Detalhados de Falhas

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

Prof. Dra. Taisy Silva Weber
Orientadora

Porto Alegre, junho de 2010

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Munaretti, Ruthiano Simioni

Um Ambiente para Descrição de Cenários Detalhados de Falhas / Ruthiano Simioni Munaretti. – Porto Alegre: PPGC da UFRGS, 2010.

78 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2010. Orientadora: Taisy Silva Weber.

1. Injeção de falhas. 2. Especificação de cenários. 3. Sistemas distribuídos. I. Weber, Taisy Silva. II. Título.

AGRADECIMENTOS

Agradeço primeiramente a Deus, por iluminar meu caminho nesta jornada. Agradeço à minha orientadora Taisy, pela oportunidade, pela sua dedicação na orientação em todas as etapas do trabalho e por toda a experiência adquirida nesta fase da minha vida. Deixo também um agradecimento especial ao professor Sérgio Cechin, bem como a todo o Grupo de Tolerância a Falhas da UFRGS - as contribuições de ambos foram essenciais para a plena efetivação deste trabalho. Também faço referência aqui ao professor Marinho Barcellos, por me introduzir na pesquisa ainda na época da graduação - sem a sua ajuda, certamente eu não estaria aqui.

À minha família (Roberto, Olenca, Robianca, Olívio e Olímpia), devo um eterno agradecimento pela compreensão, carinho, paciência e amor em cada segundo da realização do mestrado. À Daniela Ramos, uma pessoa raríssima por seu amor, carinho e ensinamentos, devo outro eterno agradecimento, pela imensa felicidade que tive de nossas vidas terem se cruzado (a vida não seria tão doce sem a tua presença, sabes disso). A todos os amigos/colegas que me acompanharam nessa caminhada, também deixo um agradecimento todo especial.

Finalmente, deixo também minha consideração para todas as empresas pelo qual trabalhei durante a realização da pesquisa - o Serpro (Serviço Federal de Processamento de Dados), a Corsan (Companhia Riograndense de Saneamento) e a Infraero (Empresa Brasileira de Infraestrutura Aeroportuária). Este agradecimento se estende, é claro, a todos os colegas de equipe, chefes e gerentes que me acompanharam durante toda esta jornada, especialmente pela compreensão que tiveram ao longo da realização do mestrado.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

DOCTOR	Integrated Software Fault Injection Environment
FAIL	Fault Injection Language
FCI	FAIL Cluster Implementation
FIONA	Fault Injector Oriented to Network Applications
FIRMAMENT	Fault Injection Relocatable Module for Advanced Manipulation and Evaluation
FIRMI	Fault Injector for RMI
JavaCC	Java Compiler Compiler
JVMTI	Java Virtual Machine Tool Interface
NFTAPE	Networked Fault Tolerance and Performance Evaluator
P2P	Peer-to-Peer
RMI	Remote Method Invocation
RTP	Real-time Transport Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A utilização de várias ferramentas de injeção de falhas em um mesmo experimento de testes fornece mais subsídios para os resultados alcançados, tornando a atividade mais efetiva e menos sujeita a erros de interpretação. Neste sentido, as *cargas de falhas* possuem um importante papel, visto que elas compõem a principal entrada a ser fornecida nestas ferramentas. No entanto, os mecanismos oferecidos, nas ferramentas de injeção de falhas existentes, para esta especificação de cargas de falhas, possuem um baixo grau de usabilidade e expressividade. Por este motivo, o presente trabalho aborda uma metodologia, na qual cenários detalhados de testes, que envolvam experimentos com injeção de falhas, possam ser especificados de maneira simples, homogênea e padronizada. Para isso, é proposta a criação de um ambiente para a especificação destas cargas de falhas, denominado como *jFaultload*. Este ambiente, por sua vez, utiliza-se de um subconjunto da linguagem Java para a especificação destas cargas de falhas, ficando responsável ainda pela *tradução*, desta carga em Java, para os respectivos formatos de carga referentes a cada injetor de falhas utilizado em um dado experimento. Para efeito de exemplo e validação do ambiente proposto, as ferramentas FIRMAMENT, MENDOSUS e FAIL/FCI são integradas neste ambiente, tornando assim o cenário de testes amplamente detalhado. O serviço a ser testado, visando a demonstração da usabilidade e expressividade da solução proposta, foi uma sessão de *video streaming*, utilizando-se para isso do protocolo RTP, onde uma campanha de testes foi realizada com o injetor FIRMAMENT.

Palavras-chave: Injeção de falhas, especificação de cenários, sistemas distribuídos.

An Environment for Detailed Fault Scenarios Description

ABSTRACT

Use of two or more fault injection tools in a test campaign enriches the scenario obtained from a test execution. Faultloads represent the main input for these tools but their specification mechanisms lack usability and expressiveness. This thesis presents a full test scenario featuring the use of *jFaultload*, which applies Java for the specification of faultloads and translates them to specific formats that are appropriate to each available fault injector. *FIRMAMENT*, *MENDOSUS* and *FAIL/FCI*, fault injectors for communication systems, were integrated in the environment and complete the test scenario. The service under test used to demonstrate the usability and expressiveness of our solution is a video streaming session using RTP Protocol, which a test campaign was executed through the FIRMAMENT fault injector.

Keywords: fault injection, scenarios description, distributed systems.

1 INTRODUÇÃO

Este capítulo inicia a discussão referente à pesquisa realizada neste trabalho. Neste sentido, a seção 1.1 aborda as ideias que motivaram a realização deste trabalho, com foco nos problemas a serem resolvidos. A seção 1.2, por sua vez, explica os objetivos do trabalho proposto, enquanto que a seção 1.3 ilustra a metodologia adotada e a seção 1.4 mostra os resultados alcançados com a pesquisa realizada. Finalmente, a seção 1.5 explica como o restante do trabalho está organizado.

1.1 Motivação

A utilização de sistemas computacionais tornou-se imprescindível, seja qual for o ramo de atividade considerado. Visando atender aos diversos requisitos exigidos, a complexidade destes sistemas tende a aumentar substancialmente, tanto em tamanho quanto nas próprias funcionalidades. Por este motivo, a realização de testes em sistemas computacionais passa a ser uma tarefa essencial e, ao mesmo tempo, desafiadora.

Nestas tarefas relacionadas aos testes, o *engenheiro de testes* é o profissional responsável por todas as questões relacionadas à *execução* dos testes. Logo, várias atividades estão envolvidas neste contexto, dentre as quais podem ser citadas: a definição do processo de testes, a escolha das ferramentas de testes a serem utilizadas e a criação dos experimentos de testes. Nestas atividades, a cobertura de erros deve ser a mais adequada possível, com o intuito de se atingir os objetivos demandados pelos testes.

A importância dos testes, no contexto de um sistema computacional, advém da necessidade de garantir um nível mínimo de *qualidade* e *disponibilidade* deste sistema. Estas duas métricas estão relacionadas aos serviços fornecidos pelo sistema, no sentido de que o usuário possa usufruir destes serviços da melhor maneira possível. Um importante conceito, considerando a forma de *mensurar* estas métricas, está relacionada à *dependabilidade* de um sistema computacional. Neste sentido, dependabilidade (Avizienis et al., 2004) indica o nível de confiança que pode ser justificadamente colocada no serviço fornecido pelo sistema.

Entre as técnicas existentes para realizar a avaliação de dependabilidade de um determinado sistema computacional, pode ser destacada a *injeção de falhas* (Hsueh et al., 1997). Injeção de falhas é uma técnica que tem como objetivo provocar falhas de forma *controlada* em um sistema alvo. A partir desta injeção, pode-se investigar o comportamento do sistema durante a presença de falhas, verificando o seu funcionamento, bem como a eficiência e correção dos mecanismos de tolerância a falhas implementados. Em comparação com outras técnicas de avaliação de dependabilidade (como *modelagem analítica*, por exemplo), a injeção de falhas (especialmente relacionada à injeção de falhas por software (Voas and McGraw, 1998)) mostra-se mais adequada para a investigação de sis-

temas complexos (Arlat et al., 2003).

Comunicação é uma das principais funcionalidades desejadas nos sistemas computacionais de larga escala. Considerando estes sistemas, a comunicação é realizada a partir de uma *rede*, que é composta por *nodos* (que são as unidades de execução, tais como computadores, servidores, switches e roteadores) e *caminhos* (que interligam dois nodos quaisquer de uma rede, permitindo a comunicação entre eles). A ocorrência de falhas em uma rede é inevitável, sendo estas denominadas *falhas de comunicação*. Como exemplos de falhas de comunicação, podem ser considerados o *colapso* (parada na execução) de algum nodo ou caminho da rede, assim como o *atraso*, o *descarte* e a *duplicação* de pacotes (unidades de dados utilizadas para comunicação) entre nodos e caminhos desta rede.

Ao mesmo tempo em que as falhas, nas redes de comunicações, são inevitáveis, a probabilidade de *ocorrência* das mesmas pode ser considerada *baixa*, relacionada a uma execução real do sistema alvo. Por este motivo, em um ambiente operacional, a identificação e reprodução de falhas pode ser difícil de ser realizada, especialmente se o sistema alvo em questão for grande e complexo. Neste sentido, o uso de *injetores de falhas* possibilita a criação de cenários detalhados de falhas, de forma que testes precisos dos mecanismos de tolerância a falhas possam ser realizados, permitindo uma relativa melhora no nível de dependabilidade do respectivo sistema alvo.

No contexto de testes em sistemas computacionais, um *experimento* representa um conjunto de ações, que são definidas pelo engenheiro de testes. O principal objetivo é o de validar a execução destes sistemas, diagnosticando se o resultado obtido está ou não de acordo com a especificação. Uma *campanha de testes*, por sua vez, abrange um conjunto de experimentos, de forma a permitir a realização de um mesmo cenário de testes sob diferentes perspectivas.

Em uma campanha de testes por injeção de falhas, o *cenário de falhas* abrange todas as configurações necessárias, de forma que este experimento possa ser realizado plenamente. Neste sentido, a criação de um cenário de falhas envolve duas entradas, a serem especificadas por um engenheiro de testes: uma *carga de trabalho* e uma *carga de falhas*. Estas cargas são ilustradas na figura 1.1 e explicadas nos itens seguintes.

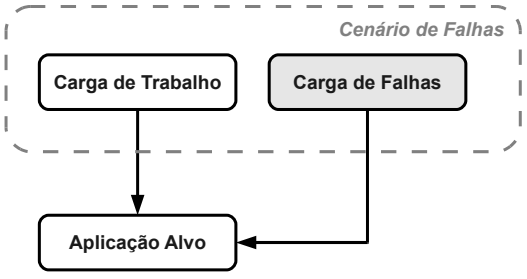

Figura 1.1: Cenário de falhas

- **Carga de Trabalho:** envolve a execução, propriamente dita, da aplicação alvo. Em outras palavras, a carga de trabalho é responsável tanto pelas configurações da aplicação alvo (parâmetros de inicialização, chamadas, entre outras), como também pela manutenção de sua respectiva execução. Assim, a carga de trabalho se faz presente em todos os momentos da execução de uma aplicação alvo, independentemente se a mesma está ou não em condições de testes sob falhas.

- **Carga de Falhas:** a carga de falhas, por sua vez, abrange a emulação das falhas a serem injetadas em um dado experimento de testes, bem como suas respectivas configurações (momentos de ativação, duração, ocorrência, entre outros). Por este motivo, a carga de falhas é o principal objeto no contexto de um experimento. Isto se justifica pelo fato de que uma carga de falhas, ao ser especificada, delimita as possibilidades de injeção de falhas que podem ser disponibilizadas durante a criação de um cenário de falhas.

Ao realizar injeção de falhas em uma determinada aplicação, uma característica que deve ser levada em consideração diz respeito a *forma* pela qual uma carga de falhas é descrita. Esta característica é importante, visto que a mesma define o *escopo* do experimento, ou seja, que tipos de falhas devem ser injetadas e quais devem ser desconsideradas. No contexto de uma ferramenta de injeção de falhas, o mecanismo de descrição de cargas de falhas interfere diretamente na usabilidade da ferramenta, influenciando seu uso efetivo em cenários realistas.

No contexto das ferramentas de injeção de falhas, a criação de cargas de falhas detalhadas é uma tarefa difícil, especialmente se considerarmos sistemas complexos. Esta dificuldade advém das abordagens distintas, para essa descrição de cargas, adotadas pelos diferentes injetores existentes. Logo, o principal problema está relacionado a *escolha* do injetor a ser utilizado para um dado experimento. Por este motivo, um esforço adicional é necessário nesta etapa, transcendendo assim a já árdua tarefa de elaboração de experimentos para testes de sistemas computacionais.

A existência de vários injetores de falhas torna mais difícil a escolha da ferramenta mais apropriada para um dado experimento. Ao mesmo tempo, a utilização de dois ou mais injetores em uma campanha de testes pode *enriquecer* os resultados obtidos. Este enriquecimento advém principalmente da possibilidade de uma comparação minuciosa entre as ferramentas utilizadas, sendo a melhor abordagem escolhida de uma maneira fácil e direta.

Desta forma, para a execução de um experimento de testes, um engenheiro de testes deve investir um certo esforço. Primeiramente, um conjunto de ferramentas de injeção de falhas deve ser escolhido. A partir daí, o engenheiro de testes torna-se o responsável por *todas* as tarefas relacionadas a criação das cargas de falhas para cada injetor, bem como as configurações relacionadas a cada uma das ferramentas. A figura 1.2 ilustra estas tarefas de forma esquemática.

Devido a todas estas dificuldades mencionadas, a abordagem de injeção de falhas pode não ser utilizada, de forma efetiva, na execução dos testes de mecanismos de tolerância a falhas, presentes nos sistemas alvo sob validação. A principal razão desta dificuldade está relacionada ao tempo de aprendizado de cada abordagem existente, que pode reduzir o uso das ferramentas como um todo. No pior dos casos, a injeção de falhas pode até mesmo ser completamente ignorada, sendo substituída por técnicas empíricas (como a desconexão de um cabo de rede, por exemplo).

1.2 Objetivos

O objetivo deste trabalho é prover um ambiente para descrição de falhas de comunicação em um alto nível de abstração, gerando, a partir desta descrição, cargas de falhas específicas para diferentes ferramentas de injeção de falhas. Para atingir este objetivo, realizou-se um estudo aprofundado das diferentes abordagens para descrição de cargas de falhas adotadas pelos injetores. Este estudo, por sua vez, visava permitir um aumento

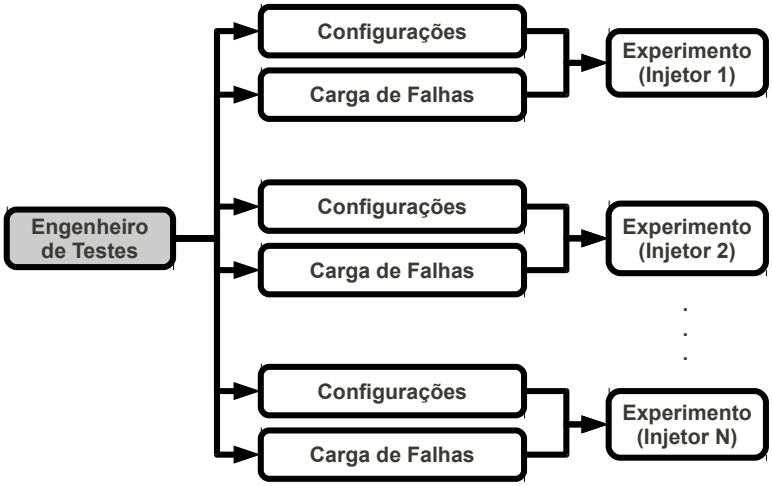

Figura 1.2: Tarefas relacionadas ao Engenheiro de Testes

substancial na usabilidade dos injetores existentes. Desta maneira, espera-se que a execução de experimentos que envolvam injeção de falhas aumentem na mesma proporção, beneficiando inclusive a área de tolerância a falhas como um todo.

Para isso, é proposta a construção de um *ambiente*, cujo foco é voltado a elaboração de cargas de falhas. Este ambiente, denominado *jFaultload*, consiste em um framework para elaboração de cargas de falhas em Java, com o intuito de facilitar a especificação de cargas de falhas por um engenheiro de testes, bem como viabilizar, de forma prática e direta, a utilização de um ou mais injetores de falhas em uma dada campanha. Ou seja, o ambiente proposto não visa substituir as ferramentas de injeção de falhas existentes, tampouco executar a injeção de falhas propriamente dita. Ao contrário, o ambiente visa facilitar o uso destas ferramentas, no sentido de fornecer uma *interface de uso comum* para as mesmas. A figura 1.3 ilustra o contexto no qual o ambiente está inserido, considerando a descrição de cargas de falhas em ferramentas de injeção de falhas.

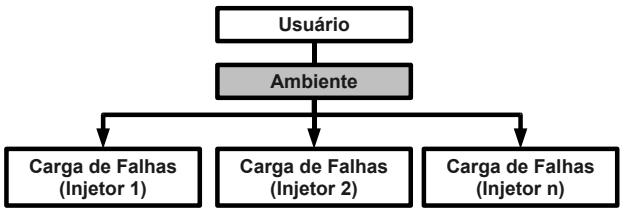

Figura 1.3: Contexto do ambiente proposto

Com o uso da ferramenta *jFaultload*, as tarefas demandadas ao engenheiro de testes são reduzidas e simplificadas. Neste caso (conforme ilustrado na figura 1.4), o ambiente *jFaultload* passa a ser o responsável por todos os passos referentes a geração das cargas de falhas, bem como pelas configurações de cada ferramenta a ser utilizada. A única tarefa demandada ao engenheiro de testes está relacionada a criação de uma *carga de falhas em Java*, juntamente com as informações de *topologia* e *aplicação alvo*, que são independentes das ferramentas utilizadas. Assim, passando a realizar menos tarefas, o engenheiro de testes pode dispender um esforço maior na campanha de testes propriamente dita, que é o assunto que realmente interessa em suas atribuições.

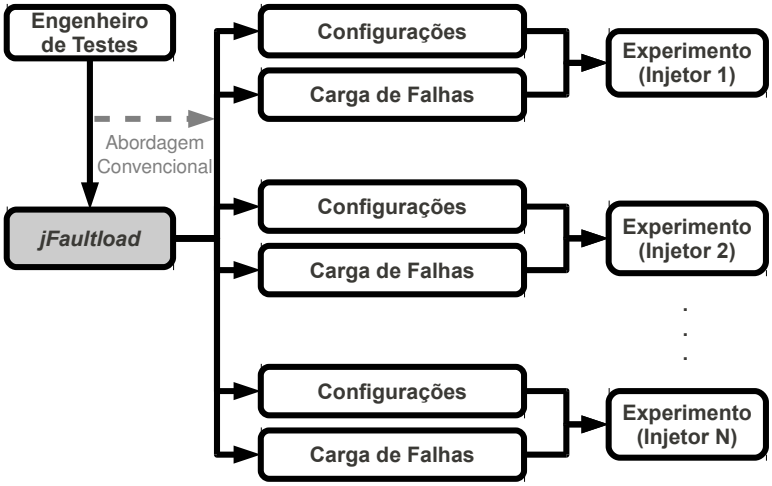

Figura 1.4: Tarefas do Engenheiro de Testes, com o uso de jFaultload

Assim, como já mencionado anteriormente, o principal objetivo do ambiente proposto consiste em, exatamente, facilitar e expandir o uso dos diversos injetores de falhas existentes na literatura. Com isso, pretende-se aumentar a usabilidade destas ferramentas. Para que isso aconteça, o ambiente proposto é concebido a partir de um conjunto de premissas. Estas premissas são explicadas nos itens que seguem.

- **Foco em falhas de comunicação:** na criação de cenários, ênfase será dada às falhas de comunicação, por serem as principais fontes de defeito no contexto de sistemas baseados em troca de mensagens.
- **Independência de injetor de falhas:** os procedimentos de criação de cenários de falhas serão os mesmos para quaisquer injetores de falhas que forem utilizados. Desta forma, cada cenário criado será convertido, de forma automática pelo ambiente, para a interface específica do injetor.
- **Elaboração de cenários detalhados:** o ambiente terá suporte a falhas múltiplas não simultâneas, onde uma sequência de falhas é definida e executada pelo injetor específico, na ordem em que as mesmas forem definidas no respectivo ambiente.
- **Extensão/flexibilidade:** na construção do ambiente proposto, será prevista uma futura extensibilidade do modelo, de forma que o ambiente possa ser facilmente adaptado para a construção de novos cenários de falhas, de acordo com a necessidade do engenheiro de testes.

1.3 Metodologia

O trabalho foi realizado segundo algumas diretrizes. Neste sentido, o principal intuito é o de, exatamente, identificar o ponto-chave do problema para que, posteriormente, uma possível solução seja proposta. Assim, a metodologia de realização deste trabalho foi dividida nos itens apresentados a seguir.

- **Pesquisa de injetores de falhas de comunicação:** realiza um levantamento das ferramentas existentes, com ênfase nos mecanismos utilizados para a descrição de cargas de falhas.

- **Elaboração de um modelo do ambiente:** visa definir quais os componentes estarão presentes no ambiente proposto, bem como a interação entre os mesmos.
- **Construção da arquitetura correspondente:** a partir do modelo definido, a arquitetura do ambiente é construída, com foco na implementação do protótipo.
- **Implementação e validação de um protótipo:** finalmente, esta parte pretende mostrar uma prova de conceito da pesquisa realizada, através de uma utilização prática do ambiente proposto.

1.4 Resultados Alcançados

Neste trabalho, foi realizado um levantamento das principais ferramentas de injeção de falhas, com ênfase nas ferramentas que dedicam algum esforço ao facilitar a descrição de cargas de falhas. Logo após, a modelagem do ambiente para a descrição de cargas de falhas foi elaborada, com o intuito de avaliar as necessidades deste ambiente, no que se refere a componentes, interações e mecanismos a serem utilizados para a especificação de cargas de falhas. Em seguida, a arquitetura do respectivo ambiente foi construída, com o intuito de vislumbrar como este ambiente poderia ser implementado.

Nas etapas de modelagem e arquitetura, foi definido o uso de *Java* como abordagem principal para a definição de cargas de falhas, pelas características de *flexibilidade* e *expressividade* proporcionadas pela mesma. A partir desta arquitetura, um protótipo foi implementado e validado, utilizando-se para isso de um conjunto de injetores.

Para efeito de exemplo do funcionamento do ambiente, uma carga de falhas em Java é fornecida como entrada. O ambiente, por sua vez, realiza a leitura desta carga em Java e, como saída, fornece as cargas de falhas referente a cada injetor utilizado no experimento de testes.

Na utilização do ambiente proposto, é possível constatar um aumento de usabilidade, referente à especificação das cargas de falhas. Isto se justifica pelo fato de que um engenheiro de testes, em uma campanha, precisa apenas interagir com uma linguagem de alto nível, sem a necessidade de aprender a abordagem de um determinado injetor de falhas. Com isso, o engenheiro pode trabalhar com mais ênfase no seu objeto de estudo (a campanha de testes em si), fornecendo mais subsídios aos experimentos gerados.

1.5 Organização do Trabalho

Considerando a pesquisa realizada, o restante do trabalho está dividido da seguinte maneira: primeiramente, o capítulo 2 realiza um levantamento dos principais injetores de falhas de comunicação existentes na literatura, com foco nos assuntos referentes a descrição de cargas de falhas. O capítulo 3, por sua vez, explica a construção do modelo do ambiente proposto, abrangendo cada detalhe do mesmo. A partir deste modelo, o capítulo 4 aborda a arquitetura construída para o ambiente, visando a preparação do mesmo para a elaboração de um protótipo, que é descrito em detalhes no capítulo 5, com sua respectiva validação no capítulo 6, realizada através de experimentos práticos. Finalmente, conclusões sobre a pesquisa realizada são abordadas no capítulo 7.

2 INJETORES DE FALHAS DE COMUNICAÇÃO

Este capítulo visa apresentar alguns dos principais injetores de falhas existentes na literatura. Primeiramente, serão descritos três injetores desenvolvidos no Grupo de Tolerância a Falhas da UFRGS: FIONA, FIRMAMENT e FIRMI, nas seções 2.1, 2.2 e 2.3, respectivamente. Em seguida, serão abordados seis injetores conhecidos na literatura: DOCTOR (seção 2.4), ORCHESTRA (seção 2.5), NFTAPE (seção 2.6), LOKI (seção 2.7), MENDOSUS (seção 2.8) e FAIL/FCI (seção 2.9). Finalmente, a seção 2.10 apresenta um comparativo destes injetores e a seção 2.11 aborda uma conclusão do capítulo.

2.1 FIONA

FIONA (Jacques-Silva, 2005) é um injetor de falhas com foco em sistemas distribuídos de *larga escala*. A abordagem utilizada no mesmo consiste na *instrumentação de código*, utilizando-se para isso da ferramenta JVMTI. O foco do injetor é o protocolo UDP, muito embora estejam previstas extensões para outros protocolos, tais como o TCP, por exemplo (Gerchman and Weber, 2006). A seguir, serão abordados o modelo de falhas adotado neste injetor, bem como a arquitetura utilizada no mesmo, além da metodologia empregada para a carga de falhas e os pontos de extensão do injetor.

O modelo de falhas adotado pelo FIONA foi o definido por *Birman* (Birman, 1996a). Desta forma, os seguintes tipos de falhas são previstos neste injetor: colapso, omissão de envio, omissão de recepção, temporização, arbitrária e particionamento de rede. Destes tipos de falhas, o particionamento de rede é o tipo mais desafiador tratado nessa ferramenta, uma vez que o mesmo exige uma configuração específica para cada nodo integrante do sistema distribuído em questão.

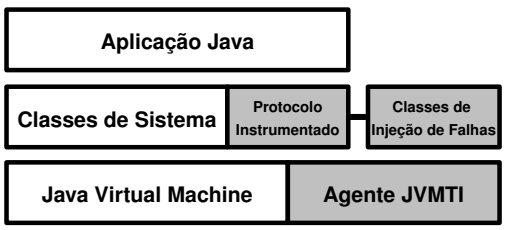

Figura 2.1: Arquitetura local de FIONA

A arquitetura da ferramenta FIONA é formada por duas partes: uma arquitetura *local* e uma arquitetura *distribuída*. A arquitetura local é constituída por três componentes: o *agente JVMTI* (localizado no nível da máquina virtual), as *classes de injeção de falhas* e

o *protocolo instrumentado* (ambos localizados no nível das classes do sistema). A figura 2.1 esquematiza a arquitetura local de FIONA.

A arquitetura *distribuída* de FIONA, assim como a local, também é constituída por três componentes: o *injetor principal* (responsável pelo gerenciamento do experimento), o *injetor de site* (que gerencia o conjunto de máquinas integrantes do respectivo site) e o *injetor local* (que aplicam as falhas propriamente ditas na aplicação alvo). A figura 2.2 ilustra esta arquitetura distribuída.

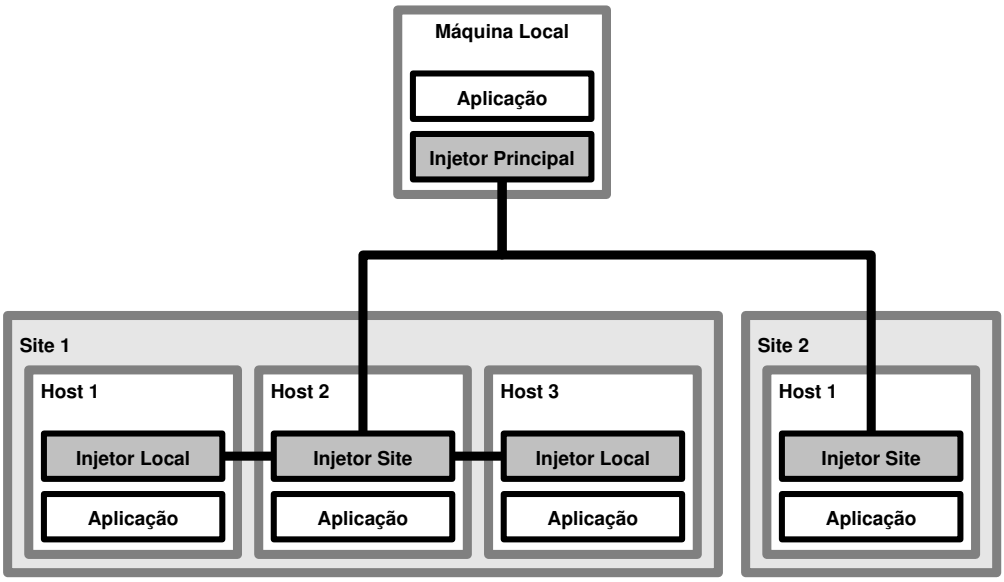

Figura 2.2: Arquitetura distribuída de FIONA

Para a criação de cenários de falhas, FIONA utiliza um arquivo de configuração, exigido somente no *injetor principal*. O funcionamento da carga de falhas na ferramenta FIONA ocorre a partir dos passos descritos a seguir.

- Ao interpretar o arquivo de configuração, um objeto `Fault` (que representa uma falha) é instanciado para cada falha especificada.
- Cada objeto `Fault`, criado no passo acima, será inserido no banco de falhas do experimento, denominado no injetor como `FaultBase`.
- Na execução do experimento, o banco de falhas (`FaultBase`) é consultado toda vez que ocorrer algum envio/recebimento de mensagem, a fim de verificar se a falha será ou não injetada na aplicação alvo.

Em questões de extensibilidade, está prevista na ferramenta FIONA uma expansão do seu modelo de falhas. Esta expansão é baseada em três passos: criação de uma classe especializada (com a lógica de ativação da falha), alteração do banco de falhas (para interpretar a nova falha do arquivo de configuração) e alteração dos métodos `send()` e `receive()` da classe Java responsável pela transmissão de mensagens na aplicação. Considerando-se estes passos, percebe-se que a extensibilidade de FIONA é complexa, sendo assim realizada de forma adequada apenas por desenvolvedores de ferramentas de validação. De qualquer forma, existem trabalhos que expandem o modelo de FIONA, como para o protocolo TCP (Gerchman and Weber, 2006).

2.2 FIRMAMENT

FIRMAMENT (Drebes, 2005) é um injetor de falhas de comunicação que trabalha no nível de sistema operacional. Derivado da ferramenta ComFIRM (Leite, 2000), a mesma é injetada nas aplicações alvo através de módulos do kernel Linux, visando a validação dos mecanismos de tolerância a falhas implementados em sistemas distribuídos, bem como em protocolos de comunicação.

No caso de FIRMAMENT, a abordagem para especificação de cenários de falhas é realizada através do conceito de *faultlet*. Um *faultlet*, em termos gerais, consiste em uma aplicação que emula um comportamento de falhas. A arquitetura de FIRMAMENT é exatamente baseada neste conceito, sendo formada pelos componentes apresentados nos itens a seguir, ilustrados na figura 2.3.

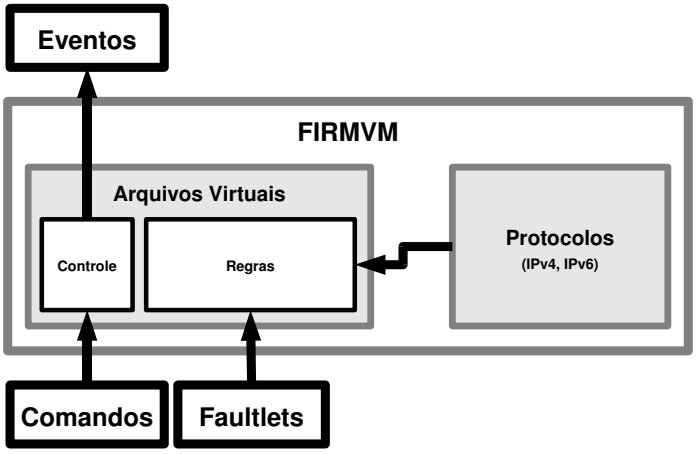

Figura 2.3: Arquitetura de FIRMAMENT

- **FIRMVM:** representa a máquina virtual do injetor de falhas, responsável pelo processamento dos *faultlets*. Vale ressaltar que esta máquina virtual possui suporte aos protocolos IPv4 e IPv6, trabalhando assim com quatro fluxos de dados (dois referentes à entrada e dois referentes à saída de ambos os protocolos).

- **Cão-de-guarda:** mecanismo responsável pela detecção de *faultlets* que levam muito tempo para executar. Isto é necessário para evitar o congelamento do processamento da aplicação alvo, bem como para detectar possíveis falhas de colapso do nodo em questão. O intervalo de tempo para a ativação do cão-de-guarda é configurável, possuindo um valor padrão de 20 milisegundos.

- **Montador:** componente que interpreta uma linguagem (do tipo *assembly*) utilizada pelos *faultlets* para a especificação dos cenários, denominada *FIRMASM*. Desta forma, o montador gera como saída um arquivo do tipo binário, interpretável pela máquina virtual FIRMVM.

- **Arquivos virtuais:** localizados na estrutura `/proc` do sistema operacional Linux, os mesmos servem para leitura/escrita de *faultlets* para cada fluxo de pacotes (neste caso, entradas e saídas dos protocolos IPv4 e IPv6), bem como para um controle geral do injetor de falhas.

2.3 FIRMI

FIRMI (Vacaro and Weber, 2006) é uma ferramenta de injeção de falhas cujo objetivo consiste na emulação de cenários de falhas envolvendo RMI (Remote Method Invocation). RMI, um sistema baseado em objetos distribuídos, possui diversos pontos suscetíveis a falhas, principalmente se considerarmos a rede de comunicação no qual o mesmo é executado. Assim, a ferramenta FIRMI é utilizada com a finalidade de avaliar as aplicações alvo que são construídas sobre RMI, incluindo os mecanismos de tolerância a falhas existentes nestas aplicações.

A arquitetura de FIRMI é definida através de um *diagrama de classes*, ilustrado na figura 2.4. Como é possível visualizar nesta figura, os tipos de falhas são implementados através de *exceções*. Estas exceções são ativadas na aplicação alvo através de uma técnica de *instrumentação*. Esta instrumentação de código, por sua vez, é realizada a partir do *stub* e do *skeleton*, ambos componentes do sistema de comunicação RMI, referentes ao cliente (que solicita as requisições) e ao servidor (que atende as requisições) da aplicação alvo, respectivamente.

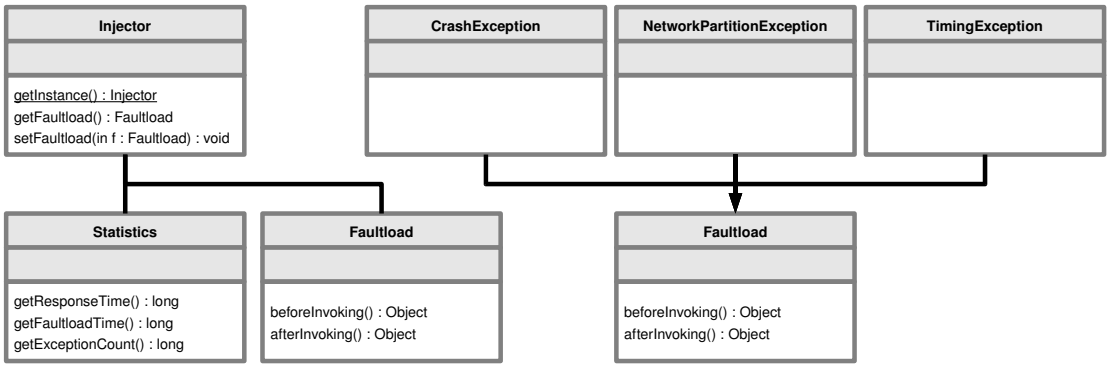

Figura 2.4: Diagrama de classes de FIRMI

Para a especificação de carga de falhas, FIRMI utiliza a classe `Faultload`. Assim, ao especificar uma carga, esta classe deve ser estendida, sendo a funcionalidade específica da mesma implementada utilizando-se das chamadas aos métodos `beforeInvoking()` e `afterInvoking()`. Estes métodos são responsáveis pelas implementações das funcionalidades que ocorrem *antes* e *depois* da carga de falhas do injetor em si, respectivamente.

Referente a extensibilidade do injetor, percebe-se que a abordagem de diagrama de classes facilita a expansão do modelo de falhas implementado, uma vez que sua construção permite uma extensão nas classes (e, consequentemente, a adição de novas funcionalidades) de forma bastante intuitiva. Ao mesmo tempo, a abordagem proposta possui a limitação de ser restrita a desenvolvedores que possuam um conhecimento razoável da ferramenta (assim como ocorre com FIONA), o que pode restringir o seu uso em outros contextos.

2.4 DOCTOR

A ferramenta DOCTOR (Han et al., 1995) consiste em um ambiente de injeção de falhas, com foco em sistemas distribuídos de tempo real. Seu objetivo inicial era injetar

falhas na aplicação de tempo real HARTS (Shin, 1991), aplicação em que DOCTOR foi utilizada de forma extensiva. Além do injetor em si, DOCTOR era formado por uma série de ferramentas auxiliares, tais como coletor de dados, gerador de cargas sintéticas de trabalho e interfaces gráficas (neste último, para interação com o usuário do injetor).

Quanto ao modelo de falhas, mesmo sendo uma ferramenta de injeção de falhas por software, DOCTOR é capaz de emular falhas de hardware tradicionais, como falhas de memória e CPU. Além destas, a ferramenta permite também a injeção de falhas de comunicação. Todas estas falhas podem ser combinadas, de forma a induzir quaisquer tipos de condições anormais no sistema alvo em questão.

Referente à arquitetura, DOCTOR define uma série de componentes. Os principais componentes integrantes desta arquitetura são descritos nos parágrafos a seguir. A figura 2.5 ilustra a arquitetura esquematizada de DOCTOR.

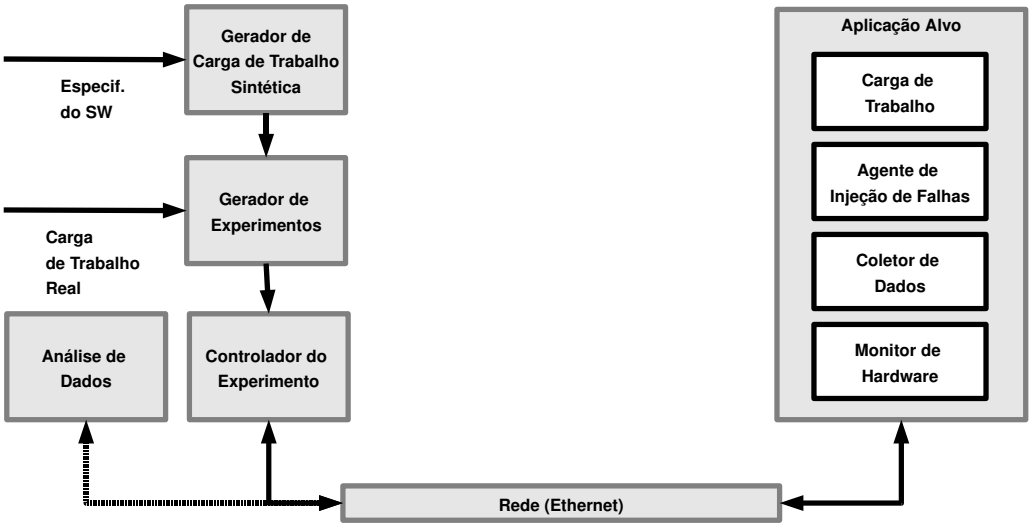

Figura 2.5: Arquitetura de DOCTOR

Primeiramente, o *Gerador de Experimentos* é o responsável pela obtenção da carga de trabalho da aplicação. Esta carga de trabalho pode ser originada tanto da execução propriamente dita da aplicação (indicada pela *Carga de Trabalho Real*), como também pode ser originada a partir da *Especificação do Software* (passando por um *Gerador de Carga de Trabalho Sintética*). Outra função do Gerador de Experimentos é a de realizar a leitura do arquivo de descrição do experimento, que contém informações sobre os tipos de falhas a serem consideradas e o tempo em que as mesmas devem ser injetadas.

Com o experimento gerado, o *Controlador do Experimento* envia comandos para o *Agente de Injeção de Falhas* durante a execução deste experimento. Este agente, por sua vez, executa tais comandos injetando as falhas ou trocando o estado da carga de trabalho (neste caso, para os estados "wait", "start" ou "stop"). Ao mesmo tempo, o Agente de Injeção de Falhas realiza o *log* de suas atividades para os componentes *Coletor de Dados* e *Monitor de Hardware*. Finalmente, o componente *Análise de Dados* é o responsável pela análise dos dados "pós-experimentos", com o intuito de gerar informações estatísticas a respeito dos testes executados.

Desta forma, considerando os quesitos de carga de falhas e extensibilidade, pode-se dizer que a ferramenta DOCTOR apresenta um mecanismo adequado para a elaboração

de cargas de falhas, assim como pontos de extensão bem definidos. Entretanto, a extensibilidade em si é limitada, devido ao foco específico do injetor (em sistemas de tempo real), além da complexidade inerente ao mesmo.

2.5 ORCHESTRA

ORCHESTRA (Dawson et al., 1996b) é uma ferramenta para teste de dependabilidade e propriedades temporais de protocolos distribuídos. A mesma é baseada em um framework, denominado *script-driven probing and fault injection*. Assim, a idéia principal é a de que cada protocolo a ser executado em um dado experimento seja especificado como uma *camada* (de forma análoga ao modelo OSI, por exemplo).

Com a finalidade de injetar falhas, ORCHESTRA insere uma camada denominada *PFI* (Probe/Fault Injection) entre quaisquer duas camadas consecutivas em uma dada pilha de protocolos. Esta camada atua sobre as *mensagens* do sistema, aplicando-se assim a injeção de falhas durante o envio e o recebimento das mesmas. A figura 2.6 (Dawson et al., 1996a) ilustra esta camada de injeção de falhas, juntamente com seus principais componentes.

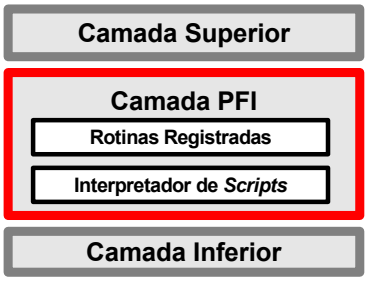

Figura 2.6: Arquitetura da ferramenta ORCHESTRA

O modelo de falhas implementado por ORCHESTRA, assim como na ferramenta DOCTOR, é voltado para sistemas baseados em trocas de mensagens, uma vez que o injetor foi construído para tal finalidade. Neste caso, o foco do modelo está voltado à *interceptação de mensagens*, conforme mencionado no parágrafo anterior. As operações previstas para injeção de falhas em mensagens são: atraso (*delay*), reordenamento, duplicação, modificação e introdução de mensagens.

Para a geração de cargas de falhas, ORCHESTRA implementa uma linguagem de *scripts*. Como esta linguagem é interpretada em tempo de execução, a modificação dos testes pode ser realizada livremente, sem necessidade de recompilar toda a pilha de protocolos existentes na aplicação alvo. Em termos de extensibilidade, a ferramenta apresenta um alto grau de expansão, uma vez que a mesma é implementada como um *framework*, possuindo pontos de extensão (*hot-spots*) bem definidos.

2.6 NFTAPE

NFTAPE (Stott et al., 2000) implementa um modelo simplificado de injetor de falhas para sistemas distribuídos. Sua principal justificativa de utilização é proveniente do argumento de que os injetores de falhas convencionais são construídos de forma a atender uma realidade muito específica do problema em que o mesmo foi pensado. Desta

forma, a *portabilidade* destas ferramentas (para a incorporação de novas funcionalidades, por exemplo) torna-se bastante restrita e complexa, dificultando as tarefas inerentes aos desenvolvedores da aplicação.

Assim, NFTAPE adota o conceito de *LightWeight Fault Injector* (LWFI), um injetor de falhas simplificado, que possui o objetivo de permitir a injeção de falhas com uma maior portabilidade, se comparado aos injetores de falhas convencionais. A partir deste conceito, NFTAPE realiza a moduralização dos principais componentes utilizados em injeção de falhas (neste caso: LWFI, *triggers* e cargas de trabalho), aumentando assim a extensibilidade e a reusabilidade dos mesmos.

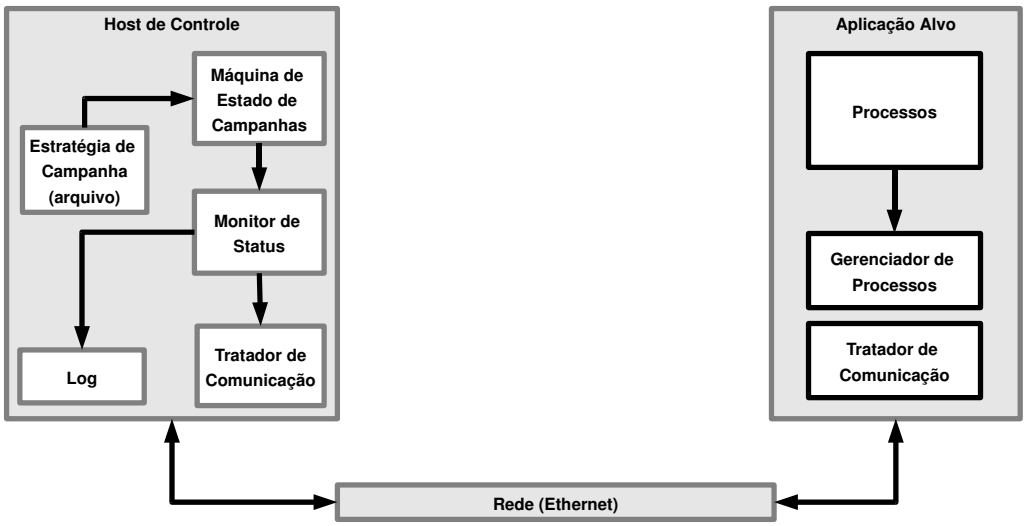

Figura 2.7: Arquitetura do NFTAPE

A arquitetura do NFTAPE, ilustrada na figura 2.7, é dividida em duas partes: o *host de controle* e a *aplicação alvo*. Enquanto o host de controle é o responsável pelo gerenciamento do experimento, a aplicação alvo corresponde ao componente no qual a falha está sendo aplicada, sendo o LWFI introduzido neste nodo.

A criação de cenários de falhas é realizada através de um *script*, denominado na ferramenta como *Estratégia de Campanha*. Neste script, são definidos a forma como o experimento é executado, bem como os valores de todos os parâmetros necessários pelo NFTAPE para que o mesmo possa executar a injeção de falhas. Para facilitar a criação de campanhas, a ferramenta disponibiliza um editor gráfico para a criação e edição dos scripts.

2.7 LOKI

Com foco em sistemas distribuídos, o injetor de falhas LOKI (Chandra et al., 2000) é construído a partir de uma visão parcial do estado global do sistema. Assim, a ferramenta assume que a execução de qualquer componente do sistema pode ser descrita como uma *máquina de estados*. Logo, a injeção de falhas é aplicada sobre os estados do sistema em questão, juntamente com mecanismos de *notificação de mudança de estado* e *sincronização de relógios*, existentes na ferramenta, de forma a permitir uma injeção consistente das falhas.

A arquitetura da ferramenta é dividida em quatro componentes principais. Estes componentes estão presentes em cada nodo integrante do sistema distribuído alvo. A descrição dos componentes pode ser visualizada nos itens a seguir, juntamente com uma ilustração completa da arquitetura na figura 2.8.

- **Máquina de estados:** responsável pela visão parcial do estado global do sistema.
- **Transportador de máquina de estados:** envia/recebe notificações de mudanças de estado em outros nodos.
- ***Recorder:*** para armazenar as mudanças de estado, bem como os tempos de ocorrência de injeção de falhas.
- ***Fault parser:*** a fim de realizar a leitura das expressões de falhas em cada mudança de estado, além de instruir o "executor" - *probe* - a injetar a falha correspondente quando a respectiva expressão for satisfeita.

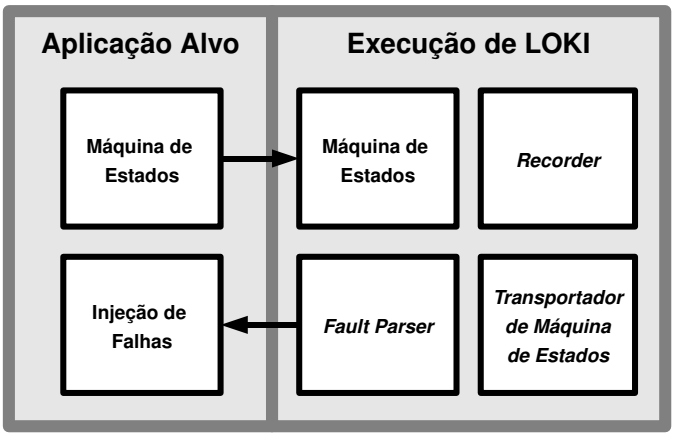

Figura 2.8: Arquitetura da ferramenta LOKI

Referente a descrição de cargas de falhas, LOKI implementa um *editor de campanhas*, de forma semelhante a implementada na ferramenta NFTAPE. Neste editor, a carga de falhas é descrita através de *expressões*, onde podem ser utilizadas variáveis que representam estados, juntamente com operadores lógicos, caso sejam necessários. Em termos de extensibilidade, a ferramenta pode ser expandida através da extensão de métodos específicos (tais como `injectFault()`, por exemplo). Entretanto, a metodologia de máquina de estados deve ser mantida, uma vez que esta é a abordagem principal utilizada pela ferramenta.

2.8 MENDOSUS

A ferramenta MENDOSUS consiste em um ambiente de emulação, com o intuito de permitir uma análise mais apurada do impacto da injeção de falhas, considerando a disponibilidade e a confiabilidade dos serviços fornecidos por uma dada aplicação sob teste. No contexto da ferramenta, a técnica de *emulação* foi justificada pelos autores, a partir da ideia de que a mesma fornece um maior equilíbrio entre três necessidades

conflitantes: (1) facilidade na obtenção/depuração de uma infraestrutura de testes, (2) observação das interações realizadas pelo sistema e (3) flexibilidade necessária para a investigação do *design* do sistema (Li et al., 2002).

A arquitetura definida para este injetor é executada a partir de uma *rede virtual*, denominada pelos autores como *VIA Switch* (conforme ilustrado na figura 2.9). Um *controlador* é utilizado para realizar as tarefas pertinentes a injeção de falhas, que também realiza a manutenção de uma visão global e consistente de toda a rede virtual em que a aplicação é executada. Ao final, cada nodo participante da arquitetura possui três componentes: um *daemon* (responsável pelas comunicações entre o nodo, controlador e emulador), as *aplicações* propriamente ditas e um *driver ethernet emulado* (que mantém a topologia e o status da rede virtual, de forma que as mensagens em trânsito possam ser transmitidas de forma apropriada).

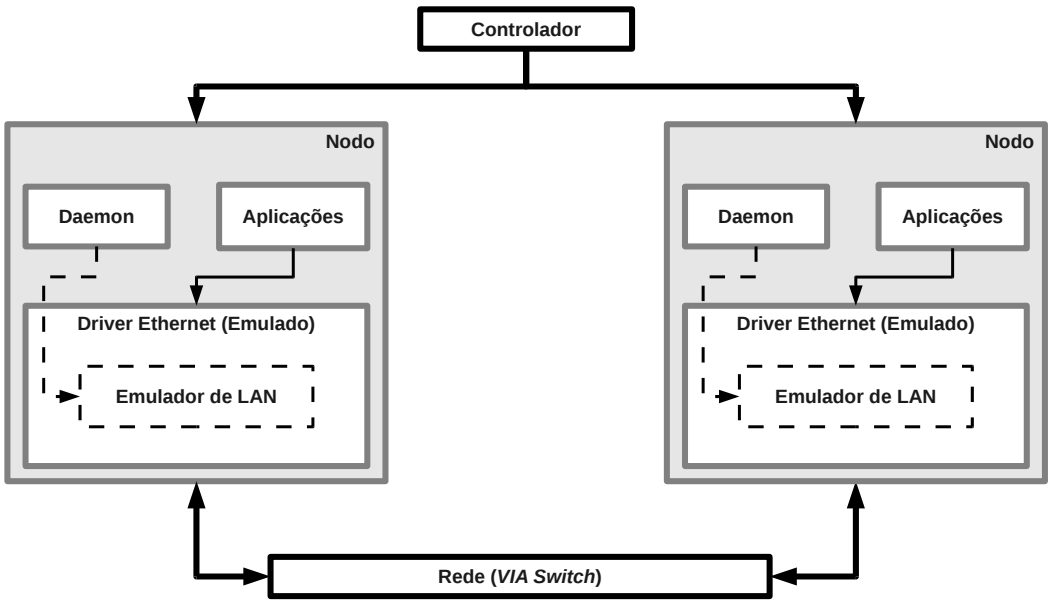

Figura 2.9: Arquitetura do MENDOSUS

Quanto ao mecanismo para especificação de cargas de falhas, MENDOSUS possui uma *linguagem de especificação*. Esta linguagem, além de especificar as falhas a serem injetadas, permite modelar também a topologia completa da rede virtual em que a aplicação alvo será executada. O modelo de falhas, por sua vez, possui falhas relacionadas a colapso (perda de pacotes) e atraso, além de mudanças na topologia em tempo real, emulando uma falha de particionamento.

2.9 FAIL/FCI

FAIL (**FA**ult **I**njection **L**anguage) (Hoarau and Tixeuil, 2005) é uma linguagem para descrição de cenários de falhas, que trabalha sobre o injetor de falhas FCI (**F**AIL **C**luster **I**mplementation). Este injetor, por sua vez, foi desenvolvido para injeção de falhas em aplicações de *cluster* e *peer-to-peer* (P2P). O principal objetivo de FAIL/FCI é oferecer um mecanismo para uma criação de cenários de falhas complexos, sejam eles probabilísticos ou determinísticos, sem a complexidade na criação destes cenários, inerentes aos injetores já existentes.

A arquitetura da ferramenta, ilustrada na figura 2.10 (Tixeuil et al., 2006) como um exemplo de um nodo em um sistema distribuído, é composta por três componentes: um *compilador*, uma *biblioteca* e um *daemon*. Ambos são descritos nos itens a seguir.

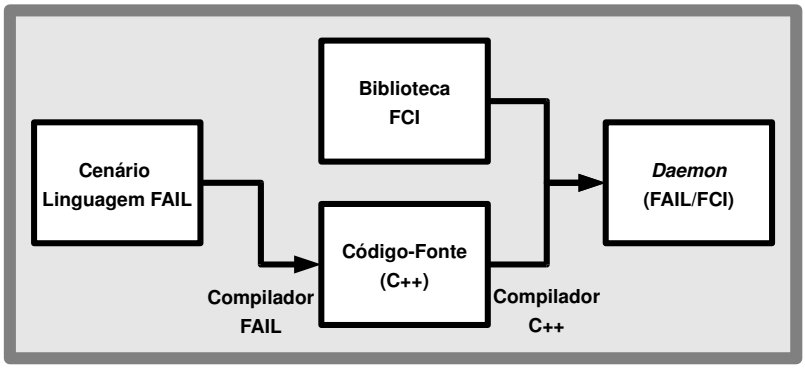

Figura 2.10: Arquitetura do injetor FCI

- **Compilador FCI:** responsável pela pré-compilação dos cenários de falhas escritos em FAIL, gerando códigos-fontes (no caso do FCI, em C++) que serão utilizados pela biblioteca FCI (descrita no próximo item), bem como arquivos de configuração para a realização do experimento.

- **Biblioteca FCI:** a partir dos arquivos gerados pelo compilador, a biblioteca FCI realiza a distribuição destes arquivos pelos nodos do sistema distribuído. Vale ressaltar que esta biblioteca distribui os arquivos como código-fonte *não compilado*, de forma a manter a *heterogeneidade* entre os nodos que formam o *cluster* do experimento.

- **Daemon FCI:** componente presente em cada nodo do *cluster*, o daemon é encarregado de realizar as compilações dos fontes em cada nodo, bem como realizar a instrumentação de código, que trata-se da injeção de falhas em si na aplicação alvo.

De todos os injetores analisados neste trabalho, o FAIL/FCI é o que se apresenta de forma mais adequada nos quesitos de criação de cenários de falhas e de extensibilidade, uma vez que a linguagem FAIL é expressiva e poderosa, permitindo a criação de cenários de falhas complexos com simplicidade, bem como uma extensão facilitada de um determinado modelo de falhas implementado. Entretanto, a linguagem FAIL possui a limitação de funcionar apenas com o injetor FCI, que por sua vez é limitado aos ambientes de *clusters* e redes *peer-to-peer*, conforme já citado anteriormente.

2.10 Comparativos

Esta seção apresenta uma série de comparações sobre os injetores de falhas analisados nas seções anteriores. O foco desta comparação diz respeito a *forma* pelo qual uma determinada *carga de falhas* é descrita no injetor. Neste sentido, os injetores analisados foram comparados a partir dos seguintes critérios: *modificação no código-fonte* (caso a criação

de cargas implique em alguma alteração do código-fonte do injetor analisado), *expressividade* (indicando a facilidade em expressar cargas de falhas no injetor) e *dificuldade* (referente a dificuldade de uso da respectiva interface oferecida pelo injetor). A tabela 2.1 ilustra esta comparação, relatada em detalhes nos parágrafos a seguir.

Tabela 2.1: Comparação quanto à criação de carga de falhas.

	Modificação no Código-Fonte	Expressividade	Dificuldade
FIONA	Não	Média	Média
FIRMAMENT	Não	Alta	Alta
FIRMI	Sim	Alta	Média
DOCTOR	Sim	Baixa	Média
ORCHESTRA	Não	Alta	Alta
NFTAPE	Sim	Alta	Alta
LOKI	Sim	Alta	Média
MENDOSUS	Não	Média	Média
FAIL/FCI	Não	Alta	Média

Primeiramente, a ferramenta **FIONA** não exige a modificação do seu código-fonte para a especificação de cargas de falhas, uma vez que a mesma é realizada a partir de um arquivo de configuração. Sua expressividade é *média*, pois mesmo oferecendo uma interface gráfica para a criação das cargas de falhas, a mesma carece de mecanismos para a especificação de cenários mais detalhados ou complexos. Finalmente, sua dificuldade também é *média*, uma vez que criação de cargas de falhas exige algum tempo de aprendizado, mesmo com a presença da respectiva interface gráfica.

Assim como FIONA, a ferramenta **FIRMAMENT** não exige a modificação do código-fonte, pois as cargas de falhas são realizadas através de *faultlets*, como explicado na seção 2.2. A expressividade da ferramenta é *alta*, visto que a mesma permite descrever uma ampla variedade de cenários de falhas. Ao mesmo tempo, essa liberdade na descrição de cargas de falhas faz com que a dificuldade de FIRMAMENT seja *alta*, pois a criação das cargas é descrita através de uma linguagem do tipo *assembly* (neste caso, a FIRMASM), o que dificulta a criação de cenários mais detalhados.

A ferramenta **FIRMI**, ao contrário das citadas anteriormente, exige modificação no código-fonte, visto que a carga de falhas é descrita através do diagrama de classes do respectivo injetor, utilizando-se do mecanismo de *extensão*. Logo, sua expressividade é *alta*, visto que a interface permite o acesso a praticamente todos os mecanismos do injetor, através da criação de uma nova classe. Sua dificuldade é *média*, uma vez que o diagrama de classes da ferramenta exige algum tempo para familiarização, mesmo possuindo uma construção intuitiva.

DOCTOR, por emular predominantemente falhas de hardware, exige modificação em seu código-fonte para a criação de cargas de falhas. Devido a esta restrição, a expressividade neste quesito torna-se *baixa*, pois não existem mecanismos para a criação de cenários mais detalhados. A dificuldade é *média*, uma vez que a criação de cargas é simples e intuitiva, porém exige que o usuário tenha conhecimentos em sistemas de tempo real, aplicações alvo deste injetor.

O injetor de falhas **ORCHESTRA**, por possuir uma linguagem de *scripts* para a especificação de cargas de falhas, não exige a modificação em seu código-fonte para a realização desta tarefa. A expressividade do injetor é *alta*, pois existe a possibilidade de expressar qualquer tipo de falha, a partir da linguagem existente para especificação de

cargas de falhas. Esta alta expressividade implica em uma dificuldade *alta* de utilização da ferramenta, visto que, para um uso efetivo do injetor, é exigido um conhecimento detalhado do tráfego de mensagens a ser analisado.

Já a ferramenta **NFTAPE**, devido ao conceito de LWFI (*LightWeight Fault Injector*, explicado na seção 2.6), permite a modificação de seu código-fonte para a criação de cargas de falhas. A expressividade do mecanismo é *alta*, uma vez que a abordagem proposta (baseada em LWFI) não restringe o modelo de falhas suportado pelo injetor. Esta liberdade, no entanto, prejudica a dificuldade do mecanismo para criação de cargas, que passa a ser *alta*.

Por utilizar uma visão parcial do estado global do sistema, a ferramenta **LOKI** necessita de modificações em seu código-fonte, visando a elaboração de cargas de falhas. A expressividade do mecanismo para tal finalidade é *alta*, pois uma carga de falhas pode ser descrita através de expressões formadas por variáveis e operadores lógicos. A dificuldade de utilização é *média*, sendo apenas necessário o tempo de aprendizado para a utilização das expressões, assim como ocorre na ferramenta DOCTOR.

A ferramenta **MENDOSUS** não exige modificação no código-fonte, pois a especificação de cargas de falhas é realizada a partir de uma linguagem de especificação. Esta linguagem, por sua vez, torna a expressividade *média*, visto que casos detalhados de falhas são mais difíceis de serem especificados com uma linguagem de scripts. A dificuldade na utilização de MENDOSUS é também *média*, uma vez que a documentação da ferramenta é adequada, mas carece de exemplos mais práticos e detalhados de utilização.

Finalmente, o injetor de falhas **FAIL/FCI** não exige modificações em seu código-fonte, uma vez que a especificação de carga de falhas é realizada através da linguagem **FAIL**, construída para tal finalidade. A expressividade desta ferramenta é *alta*, visto que a linguagem FAIL permite a criação de cenários de falhas detalhados a partir de construções simples. A dificuldade é *média*, referente a familiarização com a respectiva linguagem.

No contexto do trabalho, foram utilizados os injetores **FIRMAMENT**, **MENDOSUS** e **FAIL/FCI**. Esta escolha é justificada pelo fato de que estas ferramentas são disponibilizadas para utilização, inclusive com os códigos-fonte. A propósito, esta é uma dificuldade inerente às ferramentas de injeção de falhas: a indisponibilidade da ferramenta, seja em formato de código-fonte, ou mesmo em formato binário.

Neste sentido, um critério adicional se faz necessário para a análise das ferramentas, relacionado a *flexibilidade*. A flexibilidade é um critério que possui o significado de *multiplicidade*, ou seja, refere-se ao número de casos possíveis de falhas que um engenheiro de testes pode especificar em uma dada ferramenta. Assim, mesmo sendo importante para definir a usabilidade de uma ferramenta, a flexibilidade não foi adotada neste comparativo justamente porque, no contexto dessa análise, a informação não era facilmente disponível, uma vez que as ferramentas não estavam disponíveis.

2.11 Conclusões do Capítulo

Este capítulo apresentou um panorama dos principais injetores de falhas de comunicação existentes na literatura. Ênfase foi dada aos injetores que possuíam algum esforço em facilitar a especificação de cargas de falhas. Ao final, um comparativo foi realizado, onde percebeu-se que a boa expressividade encontrada impacta em uma dificuldade de utilização. Neste sentido, o ambiente proposto neste trabalho possui o intuito de oferecer um alto grau de expressividade com um baixo nível de dificuldade, de forma a ser uma alternativa viável para os mecanismos de cargas de falhas existentes atualmente.

3 MODELO DO AMBIENTE

Um ambiente de cenários de falhas deve refletir todos os casos possíveis de falhas, de acordo com o tipo de aplicação alvo desejado. Neste sentido, o modelo descrito neste trabalho foi elaborado visando a divisão em *elementos*, modularizando as funcionalidades do ambiente, bem como promovendo a extensibilidade do mesmo. A figura 3.1 ilustra uma visão macro de como esse modelo foi concebido no contexto do ambiente.

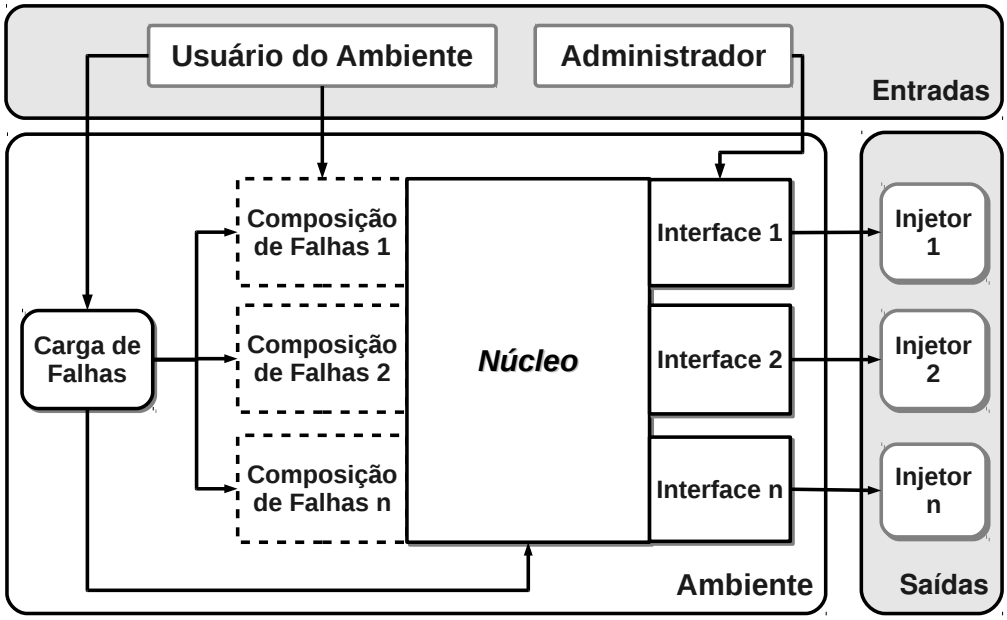

Figura 3.1: Elementos que integram o modelo do ambiente

A partir deste modelo, as seções seguintes mostram como cada um destes elementos é utilizado, considerando o funcionamento global do ambiente. Para isso, uma abordagem *top-down* será utilizada. Ou seja, primeiramente serão descritos os elementos de mais alto nível, acompanhando-se da descrição dos elementos de mais baixo nível.

Neste sentido, a seção 3.1 mostra como as *entradas* do ambiente são especificadas, enquanto que a seção 3.2 relata o mecanismo de *carga de falhas* adotado no ambiente. A seção 3.3, por sua vez, ilustra a utilização do elemento *composição de falhas*, seguindo da explicação do *núcleo* do ambiente na seção 3.4. Finalizando o detalhamento do modelo, as seções 3.5 e 3.6 mostram a *interface* do ambiente e as *saídas* geradas pelo mesmo, respectivamente. A seção 3.7 aborda uma conclusão do capítulo.

3.1 Entradas

As entradas do ambiente são responsáveis pelos dados necessários para o seu funcionamento. Desta forma, a partir das entradas, o ambiente pode ser executado da forma desejada pelo engenheiro de testes. Neste sentido, deve-se ter o cuidado de especificar as entradas com o mais alto grau de fidelidade possível, de acordo com as exigências impostas na especificação do experimento.

Conforme mencionado na seção anterior, as entradas do ambiente podem ser especificadas por dois tipos de usuários: o *usuário do ambiente* propriamente dito e o *administrador*. Cada um destes usuários possui papéis específicos no contexto do jFaultload. O detalhamento de cada um destes tipos de usuário é realizado nos itens que seguem.

- **Usuário do Ambiente:** é o responsável por um experimento de injeção de falhas. Em outras palavras, este usuário é representado pelo próprio *engenheiro de testes*, com o intuito de utilizar efetivamente o ambiente, a partir da especificação de uma *carga de falhas genérica* (no contexto do ambiente, a mesma será realizada em Java) que, por sua vez, será traduzida pelo ambiente para as cargas específicas de cada injetor utilizado. Assim, além desta *Carga de Falhas*, cabe ao usuário especificar também uma *Composição de Falhas* (caso seja necessária ao experimento), além do *Injetor Específico* a ser utilizado. A explicação detalhada de como o usuário do ambiente realiza cada uma destas tarefas será realizada nas seções posteriores.

- **Administrador:** consiste em um usuário *periódico* do ambiente, responsável pela *configuração* do mesmo. Desta forma, o administrador é responsável pela *instanciação do framework*, através da criação e manutenção da *Interface para Injetores Específicos*. Esta interface, como será visto mais adiante, é implementada a partir de um conjunto de *scripts de mapeamento* - neste caso, é criado um script por ferramenta de injeção de falhas, de forma independente aos experimentos de testes.

A partir dos conceitos acima, pode-se afirmar que um *engenheiro de testes*, mesmo englobando as tarefas relacionadas ao *usuário do ambiente*, pode também realizar tarefas de administração na ferramenta jFaultload. No entanto, é possível constatar duas atribuições, nas quais as atividades realizadas pelo engenheiro de testes podem ser divididas: primeiramente, existem as atividades relacionada à *análise dos testes*, envolvendo um conhecimento panorâmico das técnicas de testes, de forma a possibilitar a escolha da melhor técnica para o experimento em questão. A segunda atribuição, por sua vez, consiste na *programação dos testes*, abrangendo as técnicas de mais baixo nível, que fornecem suporte às técnicas de testes previstas na análise. Neste sentido, a tabela 3.1 ilustra, de forma conceitual, as principais diferenças existentes entre os dois usuários do jFaultload.

Tabela 3.1: Diferenças entre Usuário e Administrador do jFaultload

	Usuário do Ambiente	**Administrador**
Definição	Utiliza o ambiente de forma **direta** para a execução de experimentos	Prepara o ambiente para uso efetivo pelo *engenheiro de testes*
Requisitos	Saber “o que testar” e “como testar”	Saber como a técnica de testes pode ser *efetivamente* implementada
Perfil	Usuário conhecedor das técnicas de injeção de falhas, com foco em testes	Usuário conhecedor de ferramentas para injeção de falhas

Neste sentido, considerando as características relacionadas ao aumento de usabilidade (um dos objetivos pelo qual o framework jFaultload foi concebido), a adoção de dois perfis de usuários proporciona um benefício importante na utilização da ferramenta: a denominada *separação de responsabilidades* (também conhecida pelo termo inglês *"separation of concerns"*). Esta importância advém da redução da complexidade, inerente aos experimentos de teste a serem realizados. Assim, enquanto o administrador realiza as configurações e a instanciação do framework, o engenheiro de testes pode trabalhar com o assunto que realmente interessa - ou seja, o próprio experimento de testes, nas suas entranhas. Mesmo em casos que não seja possível a adoção de dois perfis de usuários, o usuário do ambiente pode atuar, de forma simultânea, como administrador, uma vez que tais tarefas de administração são desenvolvidas de forma independente das demais atividades. Além disso, estas respectivas atividades são necessárias apenas quando uma nova ferramenta de injeção de falhas passa a ser adotada para utilização no ambiente.

3.2 Carga de Falhas

Sendo uma peça fundamental para a tarefa de injeção de falhas, o mecanismo de carga de falhas deve permitir que o usuário de um dado injetor especifique as mais variadas modalidades de cargas, de acordo com o experimento a ser conduzido. Neste sentido, a *flexibilidade* e a *expressividade* são duas características desejadas neste mecanismo. Enquanto a flexibilidade possui o significado de *multiplicidade* (ou seja, está relacionada ao número de casos possíveis de falhas que podem ser injetadas), a expressividade refere-se ao nível de detalhamento, no qual esta respectiva carga pode ser especificada.

Considerando o mecanismo de cargas de falhas de forma isolada, é desejado que o mesmo seja genérico o suficiente, de forma a não restringir a possibilidade de falhas permitidas pelos injetores. Ao mesmo tempo, isso não significa que todo e qualquer injetor, que possa vir a ser usado, tenha capacidade de injetar qualquer falha que o ambiente permita descrever. Como exemplo disso, pode ser citado o injetor FAIL/FCI (Hoarau and Tixeuil, 2005): como o mesmo só permite injetar falhas de colapso, mesmo que o engenheiro de testes descreva atraso, não será possível gerar uma carga de falhas para esse injetor. Entretanto, caso sejam disponibilizados vários injetores que permitam atrasos de pacotes, uma só descrição pode servir para ambos, usufruindo assim das vantagens proporcionadas pelo ambiente.

Como já mencionado no capítulo 2, a descrição de cargas de falhas pode ser realizada de várias formas (instrumentação de código, arquivo de configuração, *scripts*, entre outras). Dentre todas essas, a utilização de uma linguagem de programação de alto nível é a modalidade que melhor se encaixa nos dois requisitos mencionados no início desta seção (flexibilidade e expressividade). Isto se justifica pelo fato de que uma linguagem pode ser facilmente estendida, alterada ou mesmo reutilizada (flexibilidade), ao mesmo tempo em que permite descrever, de forma bastante detalhada, como uma carga de falhas deve ser especificada (expressividade).

Assim, esta seção visa descrever a linguagem utilizada no ambiente (Munaretti and Weber, 2008a). Esta linguagem é de fundamental importância para o ambiente, visto que ela possibilita a criação de cargas de falhas de forma direta pelo engenheiro de testes. Além disso, a linguagem proporciona o atendimento aos requisitos de flexibilidade e expressividade, conforme explicado no parágrafo anterior.

Para a especificação destas cargas de falhas, a linguagem escolhida foi baseada em um subconjunto da linguagem de programação Java. As principais decisões que justificaram

a adoção de Java para a composição deste ambiente são explicitadas nos itens que seguem.

- **Popularidade da linguagem:** Java é amplamente utilizada, tanto no meio acadêmico como profissional, para a realização de tarefas nos mais diferentes níveis de complexidade. Isto facilita a curva de aprendizado da descrição de cargas de falhas e, por consequência, permite que o foco do engenheiro de testes esteja voltado às tarefas realmente relevantes, ou seja, relacionadas aos experimentos de testes em si.
- **Orientação a objetos:** a orientação a objetos permite uma maior *modularização* das tarefas referentes às cargas de falhas, possibilitando o *reuso* das mesmas.
- **Atendimento aos requisitos de *flexibilidade* e *expressividade:*** a linguagem Java atende, de forma apropriada e precisa, ao requisito de flexibilidade, por permitir várias formas de criação de cargas de falhas. Da mesma forma, a demanda de expressividade também é preenchida, devido à liberdade e ao nível de detalhe que Java proporciona na criação de uma carga de falhas.

Neste sentido, um exemplo de carga de falhas em Java é ilustrado na figura 3.2. Este exemplo será descrito em detalhes no capítulo 6. Para um melhor entendimento do subconjunto de Java que está modelado no ambiente, a subseção 3.2.1 ilustra quais construções de Java estão presentes, enquanto que a subseção 3.2.2 aborda o suporte adicional a topologias, prevista nesta especificação adotada pelo jFaultload.

```
class Delay {
   void main() {
      Node n = new Node($Topology);
      Packet p;
      Distrib d = new Distrib("3%");
      while(p=n.getHeadPacket()){
         if(p.getRTPProtocol()){
            if(d.sort()) {
               p.delay(0.1);
            }
         } // if(p.getRTPProtocol())
      } // while
   } // main()
}
```

Figura 3.2: Exemplo de carga de falhas em Java

3.2.1 Construções de Java Suportadas

O tempo de aprendizado dispensado em uma linguagem recém-definida é, comumente, elevado. Com o intuito de reduzir ao máximo este tempo, pode-se contar com o reuso de construções provenientes de linguagens amplamente conhecidas. Esta reutilização facilita a aplicação prática da linguagem, aumentando a produtividade do engenheiro de testes. Com isso, a elaboração de experimentos mais complexos é facilitada, trazendo uma economia de tempo na especificação das cargas de falhas.

No contexto do ambiente jFaultload, a linguagem para descrição de cargas de falhas faz uso das principais construções existentes em Java para a codificação de programas. Estas construções podem ser divididas em três partes: *Declaração de Variáveis* (podendo ser *numéricas* ou *alfanuméricas*), *Comandos de Condição* (através da chamada *if*) e *Comandos de Repetição* (utilizando-se das chamadas *while* e *for*). Estas partes, por sua vez,

são esquematizadas na tabela 3.2. O suporte a topologias, referente ao último item desta tabela, é explicado em detalhes na subseção seguinte. Este suporte, na sua concepção, foi construído com base nas ideias modeladas na ferramenta de injeção de falhas simuladas *SimmFI* (Barcellos et al., 2005).

Tabela 3.2: Construções Java previstas no jFaultload

Declaração de Variáveis	Permite a declaração de variáveis de vários tipos (como *numéricas/alfanuméricas*). Sintaxe: `<tipo> <nome> [=<valor>];`
Comandos de Condição	Definido através de um comando *if* (como na linguagem Java). Sintaxe: `if (<expr>) { <comandos> }`
Comandos de Repetição	Utilizados em operações que precisam ter uma execução repetida, de acordo com uma expressão. Sintaxe: `while|for (<expr>) { <comandos> }`
Suporte a Topologias	Suporte a classes adicionais, referentes a topologia anteriormente definida. Sintaxe: `Node|Path|Packet <nome> [=<valor>];`

3.2.2 Suporte a Topologias

Um item central ao escopo de uma linguagem para descrição de cargas de falhas de comunicação refere-se exatamente ao suporte existente para a definição de *topologias*. Por topologia, entende-se toda e qualquer modelagem de uma rede que, uma vez montada, seja a base de um experimento de injeção de falhas. Desta forma, a injeção de falhas visa atuar exatamente nos componentes desta topologia, a fim de verificar o funcionamento dos mesmos sob a presença de falhas.

Considerando o ambiente jFaultload, no contexto da linguagem a ser utilizada para especificação de cargas de falhas, o suporte a definição de topologias é realizada a partir de um conjunto de classes. Estas classes, explicadas nos parágrafos que seguem, são agrupadas em um pacote específico do ambiente, de forma a permitir uma extensibilidade, caso seja necessária. A figura 3.3 ilustra um diagrama UML que aborda as classes citadas, ilustrando como as mesmas estão modeladas.

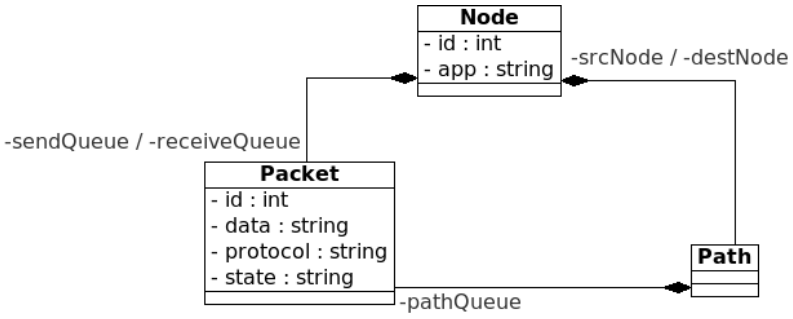

Figura 3.3: Classes do jFaultload que implementam o suporte a topologias

Classe *Packet*: representa um pacote na topologia de rede, sendo assim responsável pelo transporte das informações que trafegam na mesma. Os atributos desta classe são listados a seguir.

- *id:* corresponde ao identificador único do pacote;
- *data:* representam os dados transportados pelo pacote;

- *protocol:* indica o protocolo utilizado;
- *state:* mostra o estado em que se encontra o pacote, podendo ser um dos seguintes: *em envio*, *em transmissão* ou *em recebimento*.

Classe *Node*: modela uma determinada unidade de processamento, que é integrante da rede representada pela topologia. Neste caso, um nodo pode ser tanto um computador como um switch ou roteador, por exemplo. Com relação aos atributos, os mesmos são apresentados nos itens que seguem.

- *id:* identificador único do nodo;
- *app:* indicação da aplicação alvo a ser executada no respectivo nodo (opcional, no caso de switches ou roteadores);
- *sendQueue:* lista de pacotes *em envio* pelo respectivo nodo;
- *receiveQueue:* lista de pacotes *em recebimento* pelo nodo.

Classe *Path*: esta classe tem como objetivo detalhar um canal de transmissão entre dois nodos quaisquer de uma dada topologia. Para isso, são definidos os atributos mostrados a seguir.

- *srcNode:* representa o nodo *origem* do caminho;
- *destNode:* indica o nodo *destino* do caminho;
- *pathQueue:* armazena uma lista de pacotes, correspondente aos pacotes que se encontram *em transmissão* pelo respectivo caminho.

Além dos atributos, cada uma das classes mencionadas possui a definição de um conjunto de métodos. Tais métodos referem-se ao conjunto de falhas que podem ser injetadas em um experimento. Este conjunto, por sua vez, delimita o *modelo de falhas* do ambiente proposto. As definições destes métodos são elaboradas nos itens seguintes, classificados por *níveis*. Na tabela 3.3, pode-se visualizar em quais classes estes métodos estão implementados, indicando-se assim os tipos de falhas que podem ser injetadas em cada componente da topologia.

Tabela 3.3: Relação entre classes e métodos para injeção de falhas

Nível	Método	Classes
Pacote	`drop()`	Packet
	`delay(<x>)`	Packet
Físico	`networkPartitioning()`	Node, Path, Packet
Lógico	`crash()`	Node, Path, Packet
	`omission(<x>)`	Node, Path, Packet
	`timing(<x>)`	Node, Path, Packet
	`byzantine(<x>)`	Node, Path, Packet

Nível de pacote: diz respeito às falhas que afetam, de forma direta, os pacotes que trafegam pela respectiva topologia utilizada no experimento. Logo, apenas as falhas implementadas na classe *Packet* são consideradas neste nível. As falhas possíveis a nível de pacote podem ser classificadas nos tipos descritos a seguir.

- *drop():* realiza o *descarte* do pacote indicado. Assim, a transmissão do respectivo pacote passa a não ser realizada, sem a possibilidade de recuperação posterior do mesmo.

- *delay(<x>):* efetua um *atraso* no pacote sob falhas. Neste caso, a transmissão ainda é realizada, mas após o intervalo especificado.

Nível físico: refere-se às falhas relacionadas a *topologia* de rede utilizada no experimento. Considerando o ambiente proposto, a única falha prevista neste nível está relacionada ao *particionamento de rede*, explicado a seguir.

- *Particionamento de rede:* realiza um particionamento na rede utilizada pelo experimento. Assim, a respectiva rede se transforma em duas ou mais subredes, de forma que uma subrede não possui comunicação alguma com a outra.

Nível lógico: este nível engloba todas as falhas que visam modelar um comportamento específico a um dado componente da rede. Para este nível, são previstos quatro tipos de falhas a serem injetadas, conforme os itens seguintes.

- *Colapso:* refere-se a interrupção *total* do serviço fornecido pelo componente. Assim como acontece com a falha "*drop()*" (descrita anteriormente), não existe a possibilidade de um retorno posterior deste serviço.

- *Omissão:* ocasiona uma interrupção *parcial* do serviço oferecido pelo componente. Neste caso, o nível desta interrupção pode ser configurado pelo usuário do experimento.

- *Temporização:* proporciona um *atraso* no fornecimento do serviço pelo componente. Assim como a falha anterior, o tamanho deste atraso também é configurável pelo usuário.

- *Aleatório (Bizantino):* uma falha aleatória (também conhecida como *bizantina*) apresenta um comportamento imprevisível. Em comparação com os demais tipos de falhas, esta é a considerado a mais complexa, considerando-se os aspectos de injeção de falhas, podendo englobar todas as demais, e ainda a alteração do conteúdo de pacotes, além da duplicação e geração espontânea de novos pacotes.

Finalizando o suporte a topologias existente na linguagem proposta, mecanismos de *ativação de falhas* fazem-se necessários. O principal objetivo da existência destes mecanismos advém da necessidade de delimitar os eventos passíveis de receber, em um dado experimento, a injeção de falhas necessária para a execução dos testes. No contexto do ambiente, foram definidos os mecanismos de ativação relatados nos itens que seguem.

- **Envio/recebimento de mensagens:** consiste em injetar uma falha após a ocorrência de um envio ou recebimento de uma mensagem qualquer, sendo esta mensagem escolhida de forma aleatória ou determinística. Para isso, é utilizado o atributo *state*, existente na classe *Packet*, uma vez que este atributo indica se o respectivo pacote está em envio ou em recebimento.

- **Valor de variáveis:** este é um mecanismo existente de forma natural na linguagem proposta, visto que tal funcionalidade já existe na linguagem Java. Assim, considerando o ambiente, falhas podem ser injetadas a partir de um certo valor que uma determinada variável possa assumir durante a execução de um experimento.

- **Tempo pré-determinado:** neste trabalho, é possivel injetar falhas considerando um tempo determinado do experimento, bem como entre intervalos pré-definidos. Para isso, é fornecida uma chamada de método, presente em todas as classes de topologia mencionadas anteriormente, denominada *at(x[,y],<classe.falha>)*. Esta chamada indica que uma falha (especificada no parâmetro *<classe.falha>*) pode ser injetada a partir do tempo *x* ou, alternativamente, entre os intervalos *x* e *y*.

- **Distribuições aleatórias:** neste mecanismo, distribuições podem ser utilizadas para a realização de injeção de falhas. Neste sentido, como está se trabalhando com a linguagem Java, qualquer biblioteca disponível para a mesma pode ser utilizada neste contexto. Assim, o usuário do ambiente pode ter a liberdade de utilizar a biblioteca de distribuições mais adequada para a sua respectiva realidade.

3.3 Composição de Falhas

Conceitualmente, uma composição de falhas define um conjunto de falhas admitidas por uma determinada aplicação. Assim, cabe ao experimento emular falhas dentro deste conjunto, contendo assim todos os tipos de falhas necessários para a sua execução. Desta forma, é possível saber, no dado experimento, quais tipos de falhas poderão ser injetadas e quais tipos não serão tratados.

No contexto do ambiente, é utilizada uma abordagem simplificada para a especificação de uma composição de falhas, baseada em *script*. Esta abordagem é ilustrada na figura 3.4 e explicada nos parágrafos subsequentes.

```
<Component>.<Fault Label> { <Primitive Fault> [,;] <PrimitiveFault> [,;] ... }
<Component>.<Fault Label> { <Primitive Fault> }
...
```

Figura 3.4: Abordagem *script* para especificação de composições de falhas

A linguagem *script*, utilizada para a especificação de composição de falhas, é formada por três construções. A construção <Component> representa o **componente** para o qual se deseja especificar o tipo de falha, podendo ser Node ou Path. <Fault Label> indica um **rótulo** para o tipo de falha, sendo diretamente especificado pelo usuário do script. Finalmente, <Primitive Fault> especifica uma falha primitiva a ser adicionada no tipo de falha a ser especificado, podendo ser um dos métodos pertencentes a classe **Component**. Esta classe, por sua vez, será vista em detalhes na seção 3.4.

Ademais, esta linguagem *script* é utilizada a partir de um arquivo de configuração, sendo uma linha para cada tipo de falha. O tipo de falha é acoplado ao respectivo componente, através do caractere ".". As falhas primitivas referentes a cada tipo de falha são delimitados pelos caracteres "{" e "}". Neste escopo de falhas primitivas, cada elemento (que representa uma falha primitiva) é separado pelos caracteres "," ou ";", representando os operadores lógicos "E" (execução de *todas* as falhas primitivas relacionadas no escopo) e "OU" (execução de apenas *uma* das falhas primitivas relacionadas no escopo, escolhida de forma aleatória), respectivamente.

Considerando sistemas baseados em troca de mensagens, uma composição está relacionada a um *modelo de falhas*. Neste sentido, existem diversos modelos de falhas disponíveis na literatura, dentre os quais podem ser destacados os modelos definidos por

Cristian (Cristian et al., 1986), Birman (Birman, 1996b) e Jalote (Jalote, 1994). Cristian define as falhas de colapso, omissão, temporização e bizantinas. Birman, por sua vez, define as falhas de colapso, parada segura (*fail-stop*), omissão de envio, omissão de recepção, rede, particionamento de rede, temporização e bizantinas.

Desta forma, mesmo sendo simplificada, a linguagem criada permite a especificação de composições elaboradas de falhas, refletindo de maneira adequada os conceitos indicados nos mesmos. Para efeito de exemplo, a figura 3.5 ilustra um exemplo de composição de falhas especificado a partir desta linguagem - neste caso, é ilustrada a especificação do modelo proposto por Cristian, definindo-se assim as falhas de colapso, omissão, temporização e bizantinas. As falhas *primitivas* (ilustradas, na figura, dentro dos escopos delimitados por "{"e "}") serão abordadas detalhadamente na seção 3.4.

```
Node.Crash { restart() , stop() }
Node.Omission { dropPackets(packet) ; dropPackets(percent) }
Node.Timing { delayPackets(packet) ; delayPackets(percent) }
Node.Byzantine {
    stop() ; dropPackets(packet) ; delayPackets( Uniform(percent) )
}
```

Figura 3.5: Exemplo para especificação de composições de falhas (modelo de Cristian)

3.4 Núcleo

O *núcleo*, por ser o elemento de mais baixo nível do ambiente, define todas as operações básicas suportadas pelo mesmo. Logo, o núcleo será o responsável pela delimitação das falhas a serem implementadas, criando-se assim o *escopo* das falhas que poderão ser injetadas em um determinado experimento. Visando principalmente questões de desempenho, o núcleo implementado por este ambiente adotará uma abordagem simplificada, sendo assim formado por um conjunto pequeno de primitivas.

O foco do ambiente, conforme já mencionado, está voltado a falhas de comunicação. Sistemas baseados em troca de mensagens, por terem a sua execução realizada em uma rede de computadores, são compostos basicamente por dois componentes fundamentais, a saber: **nodos** e **caminhos**. Estes componentes, conceituados nos itens que seguem, serão utilizados no núcleo do ambiente para a especificação das cargas de falhas, necessárias na elaboração de um experimento.

- **Nodos:** modelam as *unidades de execução* do sistema em questão, possuindo estado interno próprio. Assim, dependendo do contexto, um nodo pode representar um **emissor**, um **receptor** ou **ambos**.

- **Caminhos:** correspondem as *unidades de comunicação* entre dois nodos quaisquer do sistema. Desta forma, a troca de dados entre os nodos do sistema só pode ser realizada através de caminhos. Além disso, caminhos possuem uma velocidade de propagação específica, que indica o tempo no qual um determinado dado leva para ser transportado de um nodo do sistema para outro.

Integrado aos nodos e caminhos, dois conceitos complementares são necessários. Estes conceitos estão relacionados aos *pacotes* e à *rede* como um todo, que são explicados

nos itens seguintes. A figura 3.6 ilustra uma visão completa deste conceito de *Rede*, envolvendo assim uma possível topologia que pode ser modelada no núcleo, a partir dos componentes *Nodo* e *Caminho*, juntamente com os *pacotes* de comunicação.

- **Pacotes:** formam as *unidades de dados* existentes no sistema. Portanto, quando um determinado nodo do sistema deseja transmitir ou receber conteúdos de algum outro nodo, estes conteúdos são armazenados em um ou mais pacotes, para assim serem transportados a partir do caminho existente. Logo, um determinado pacote, durante a execução do sistema, pode estar no contexto do nodo, do caminho ou em ambos (neste caso, representando um pacote *em envio* ou *em recebimento*).

- **Rede:** o componente rede engloba uma união entre todos os componentes acima, sendo os nodos interligados através dos caminhos, com os respectivos pacotes fluindo entre eles. Desta forma, o componente em questão modela o sistema como um todo.

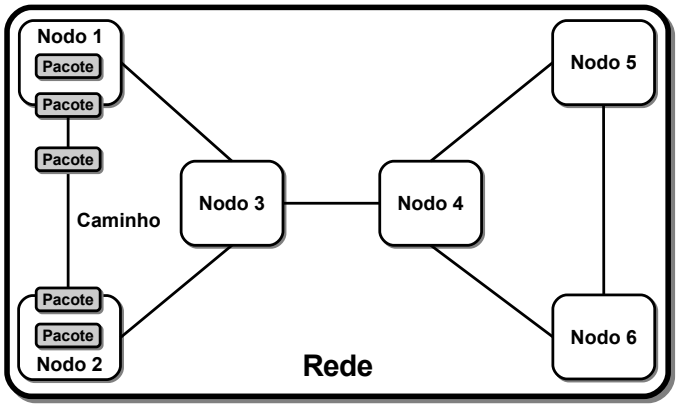

Figura 3.6: Modelo de rede do núcleo, com os componentes *Nodo* e *Caminho*

Além disso, o núcleo do ambiente define também as *falhas primitivas* suportadas pelo ambiente. Por falha primitiva, entende-se que é um tipo de falha comumente encontrada em sistemas baseados em troca de mensagens, podendo levar o respectivo sistema a um estado *inconsistente*. Desta forma, o elemento *Composição de Falhas* poderá fazer uso destas falhas primitivas, deixando assim o ambiente *independente* de um modelo de falhas específico.

No contexto do ambiente aqui descrito, as falhas primitivas serão aplicadas sobre os **pacotes** do sistema em questão, afetando, consequentemente, os **nodos** e **caminhos**. Isso ocorre porque o ambiente é voltado a falhas de **comunicação**. Vale lembrar que estes pacotes podem estar em contexto de nodo, caminho ou ambos, o que determinará sobre qual dos dois componentes (neste caso, nodo ou caminho) a respectiva falha primitiva será aplicada. Assim, foram adotadas as falhas primitivas apresentadas nos itens a seguir.

- **Reinício:** o componente em questão é *reiniciado*, tendo assim sua configuração retornada ao seu respectivo estado inicial. Nos contextos de nodo e caminho, isto significa que todos os pacotes serão descartados, sem possibilidade de recuperação posterior dos mesmos.

- **Parada:** o componente tem a sua execução *suspensa*, podendo ser *retomada* em algum momento posterior do experimento. Em nodos, esta falha primitiva corresponde à parada no envio e recebimento de pacotes, enquanto que em caminhos, indica a parada na propagação de pacotes. Ao retomar a execução, os pacotes que estavam presentes nos respectivos componentes **não** são perdidos, seguindo-se assim os destinos já definidos anteriormente para os mesmos.

- **Perda de pacotes:** como o nome indica, o componente sofrerá o descarte de um conjunto dos pacotes existentes no contexto do mesmo. Este descarte pode ocorrer de forma *determinística* ou *probabilística*. O descarte determinístico é baseado em um evento fixo/previsível do sistema, enquanto que o probabilístico é baseado em um percentual de pacotes que devem ser descartados durante o experimento, independente do conteúdo dos mesmos.

- **Atraso:** nesta falha primitiva, todos os pacotes presentes em um dado componente sofrem um atraso, parametrizável pelo usuário do ambiente. Assim como na perda de pacotes, este atraso pode ser *determinístico* (a partir de um valor fixo de atraso) ou *probabilístico* (a partir de uma distribuição aleatória).

Considerando a topologia e as falhas primitivas descritas, ambas são definidas no núcleo através de uma *hierarquia de classes*. Esta técnica é adequada para a construção do núcleo, uma vez que a mesma permite delimitar, a partir de um componente genérico, todas as funcionalidades necessárias - neste caso, as falhas primitivas. Desta forma, fica a cargo do componente específico a implementação de tais funcionalidades, que devem seguir as diretrizes definidas no respectivo componente genérico.

No caso do ambiente, é definida uma interface **Component**, que define os métodos correspondentes às falhas primitivas definidas nos itens anteriores, a saber: **restart()**, **stop()**, **dropPackets()** e **delayPackets()**, respectivamente. No contexto do núcleo, a classe **Network** engloba um conjunto de objetos da classe **Component**. Além disso, existem as classes específicas **Node** e **Path**, que implementam os métodos de **Component**, além da classe **Packet**, abrangendo assim todos os componentes fundamentais abordados. O diagrama UML desta estrutura de classes pode ser visualizado na figura 3.7.

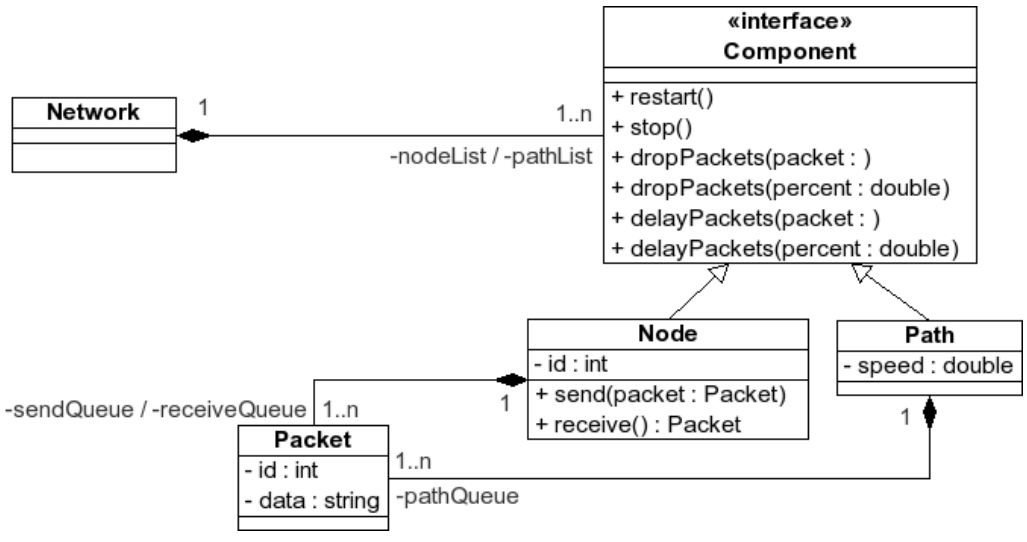

Figura 3.7: Diagrama UML do núcleo do ambiente

3.5 Interface

Este elemento visa mapear um cenário de falhas, previamente criado no ambiente, para um formato de um determinado injetor. Ao atingir tal formato, o cenário é chamado *cenário específico*, correspondendo assim à carga efetiva de falhas do injetor específico, servindo como dados de entrada ao mesmo. Desta forma, pode ser oferecida toda a compatibilidade necessária aos injetores de falhas existentes. A arquitetura deste elemento, juntamente com os componentes que integram o mesmo, é ilustrada na figura 3.8 e explicada nos itens seguintes.

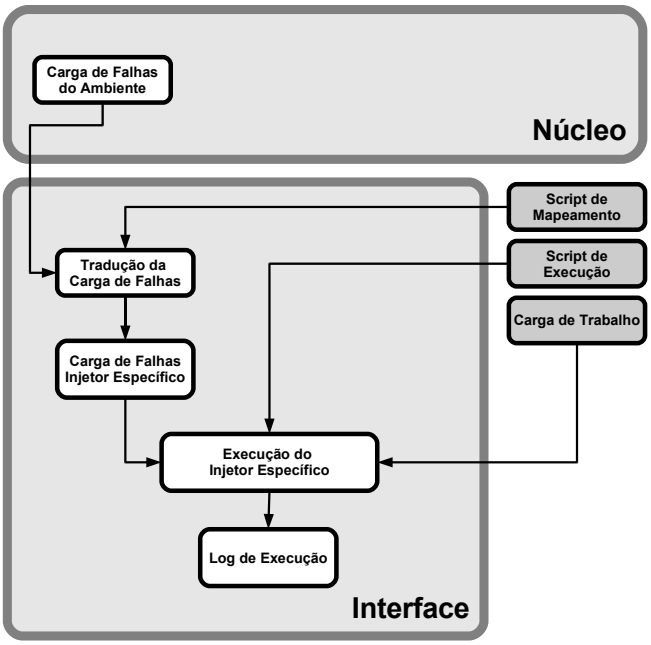

Figura 3.8: Arquitetura da Interface para Injetores Específicos

- *Carga de Falhas do Ambiente:* corresponde às falhas a serem injetadas em um dado experimento. Para isso, as respectivas falhas são especificadas através de um *script*, utilizando-se da linguagem Java (como visto anteriormente).
- *Script de Mapeamento:* componente responsável pelo *relacionamento* entre os comandos do ambiente e os comandos do injetor. Este script é formado pela respectiva chamada de um tipo de falha do ambiente, acompanhada do comando corresponde a um injetor. Este mapeamento pode ser classificado em dois tipos, conforme descrito a seguir.
  - **Gerais:** neste tipo de script, todos os comandos do injetor são mapeados para comandos do ambiente. Assim, pode-se cobrir todas as funcionalidades do injetor no ambiente. Entretanto, a efetividade do script é proporcional ao número de comandos mapeados, tornando esta etapa complexa.
  - **Específicos:** em um mapeamento específico, *somente* os comandos utilizados no dado experimento são mapeados para comandos do ambiente. Em comparação ao mapeamento geral, a construção deste script torna-se mais simplificada. No entanto, várias readaptações podem ser necessárias, de acordo com a natureza do experimento.

- *Script de Execução:* corresponde aos comandos e parâmetros necessários à execução de um injetor específico. Como esta especificação é utilizada diretamente na execução do injetor, a mesma não possui uma estrutura definida. Logo, o script é informado ao ambiente de forma direta.

- *Carga de Trabalho:* componente que implementa a carga de trabalho da aplicação alvo sob teste. Assim como o script de execução, este componente também é utilizado de forma direta pelo injetor específico, para fins de coleta de dados, como geração de *logs* (descrito logo em seguida).

- *Tradução da Carga de Falhas:* responsável pela conversão da carga de falhas do ambiente em uma carga de falhas específica do injetor utilizado. Para isso, é utilizado o *script de mapeamento*, que também será descrito em seguida.

- *Carga de Falhas do Injetor Específico:* principal saída do ambiente, corresponde à carga de falhas real, a ser inserida no respectivo injetor específico para execução e posterior análise de resultados.

- *Execução do Injetor Específico:* consiste na realização do experimento propriamente dito, a partir da carga específica gerada no item anterior, utilizando também o **Script de Execução** e a **Carga de Trabalho** passados como entrada.

- *Geração de Log de Execução:* efetua uma coleta de dados, referente ao experimento realizado no injetor, considerando possíveis erros/problemas na execução.

- *Log de Execução:* permite visualizar os resultados obtidos na injeção realizada, com o intuito de gerar dados estatísticos, obter indicadores de uso, comparações de execução/desempenho, entre outros itens relevantes.

Assim como na composição de falhas, este elemento é descrito pelo usuário seguindo-se a abordagem de *plugin*. Em outras palavras, o usuário do ambiente especifica a *forma* pelo qual a especificação do injetor específico é **mapeada** ao modelo de falhas já definido no ambiente. Para isso, é utilizado um *script de mapeamento*, cuja sintaxe pode ser visualizada na figura 3.9.

```
<Fault Label 1> : <Command>
<Fault Label 2> : <Command>
...
<Fault Label n> : <Command>
```

Figura 3.9: Script de mapeamento - modelo de falhas/especificação do injetor

Neste script, cada linha representa um mapeamento. Este mapeamento, separado pelo caractere ":", corresponde a um modelo de falhas do ambiente (representado por "<Fault Label>" pelo respectivo comando a ser chamado no injetor específico (representado por "<Command>". Desta forma, é possível realizar a ligação entre o ambiente e o injetor de forma simples, facilitando assim a execução dos experimentos.

Vale ressaltar que o ambiente permite a inclusão de vários injetores de falhas simultaneamente a uma execução de um dado experimento. Por este motivo, são ilustrados dois exemplos de scripts de mapeamento na figura 3.10. A figura 3.10(a) representa um

mapeamento para o injetor FIONA (Jacques-Silva et al., 2004) (um injetor de falhas para aplicações de rede), enquanto que a figura 3.10(b) ilustra um mapeamento para o injetor FAIL/FCI (Hoarau and Tixeuil, 2005) (um injetor para aplicações do tipo *grid*). Em ambos os casos, o modelo de falhas utilizado refere-se ao proposto por **Cristian** e, logo, são utilizados aqui os rótulos já definidos na figura 3.5, como visto na seção 3.3.

(a) FIONA (b) FAIL/FCI

Figura 3.10: Exemplos de scripts de mapeamento

3.6 Saídas

Ao final do processamento da *interface* do ambiente (descrita na seção anterior), as *saídas* do jFaultload referem-se ao objetivo principal no estudo de um experimento de testes. Neste caso, tais saídas correspondem às cargas de falhas, referentes a cada injetor escolhido para execução no respectivo experimento.

Consequentemente, cada saída gerada pelo ambiente corresponde à *entrada* da respectiva ferramenta de injeção de falhas escolhida. Esta entrada, por sua vez, pode ser diretamente utilizada em cada um dos injetores, sem que, para isso, o engenheiro de testes tenha um conhecimento profundo sobre as particularidades das ferramentas.

A tabela 3.4 ilustra um exemplo de saídas do jFaultload, envolvendo três ferramentas de injeção de falhas - MENDOSUS, FIRMAMENT e FAIL/FCI. Este exemplo corresponde à execução de uma carga de falhas em Java, previamente especificada pelo engenheiro de testes. Uma explicação detalhada, referente à transformação de uma carga de falhas em Java para cargas de falhas aplicáveis aos injetores, será realizada no capítulo 6.

Tabela 3.4: Exemplo de saídas - jFaultload

	Delay

FAIL/FCI	`Daemon Logica_Nodo1 {`
	`  node 1:`
	`  int distribution = FAIL_RANDOM(1, 100);`
	`  distribution <= 3  -> stop, goto 2;`
	`  node 2:`
	`}`
	
	`Computer Nodo1 {`
	`  program = "/opt/Prog1";`
	`  daemon = "Logica_Nodo1";`
	`}`

MENDOSUS	`// Configuração da topologia:`
	`set_host Nodo1 10.0.0.1`
	
	`// Comandos para executar a aplicação alvo:`
	`set_command 0 "/opt/Prog1" EXEC 0 Nodo1`
	
	`// Falhas a serem injetadas:`
	`set_fault Nodo1 ft_host_freeze_rec TRANSIENT poisson 0.03`

FIRMAMENT	`SET 100 R0 ; R0 = 100; // 100%`
	`RND R0 R1 ; R1 = RND(R0); // Obtém número (semente 100)`
	`SET 95 R0 ; R0 = 97; // 3% de probabilidade`
	`SUB R1 R0 ; R0 = R0 - R1;`
	`JMPN R0 DROP ; se R0 é negativo, descarta o pacote`
	
	`DROP:`
	`  DRP`

3.7 Conclusões do Capítulo

Este capítulo apresentou uma modelagem completa, referente ao ambiente de especificação de cargas de falhas jFaultload. Foi ressaltado que o modelo é dividido em *elementos*, sendo cada elemento descrito de uma forma minuciosa e detalhada. Mesmo com uma descrição extensa, o modelo proposto mostrou-se com um baixo nível de complexidade e altos níveis de flexibilidade e expressividade, requisitos obtidos principalmente através da abordagem Java, utilizada para a especificação de cargas de falhas. Além disso, a extensibilidade também possui um alto nível de aderência no contexto deste modelo, devido aos elementos que permitem tal característica, tais como os dois perfis de usuários, bem como a possibilidade de realizar composições de falhas.

4 ARQUITETURA DO AMBIENTE

Este capítulo descreve a arquitetura do framework jFaultload. O principal objetivo desta arquitetura é fornecer os subsídios necessários para a implementação de um protótipo. Além disso, a arquitetura proposta visa refletir, de forma prática, os conceitos abordados na modelagem do ambiente (conforme visto no capítulo 3).

Neste sentido, a arquitetura do jFaultload é, primeiramente, abordada através de uma divisão em *camadas*, como será descrito na seção 4.1. Logo após, a arquitetura do ambiente terá a sua visualização realizada a partir de *blocos* (na seção 4.2). Em seguida, a seção 4.3 descreve como o modelo do ambiente se relaciona com cada componente da arquitetura. Ao final, o funcionamento da arquitetura proposta é explicado em detalhes, na seção 4.4. As conclusões do capítulo são relatadas na seção 4.5.

4.1 Visão em Camadas

Além de subsidiar uma futura implementação, a arquitetura de uma ferramenta é primordial para a *divisão* das funcionalidades, permitindo uma melhor *modularização* dos componentes, além de tornar mais *organizada* a elaboração de sua concepção interna. Considerando um nível alto de granularidade, a arquitetura do jFaultload é dividida em níveis de operação, aqui chamados de *camadas*. Estas camadas, enumeradas do mais alto para o mais baixo nível (figura 4.1), são descritas detalhadamente em cada um dos parágrafos subsequentes.

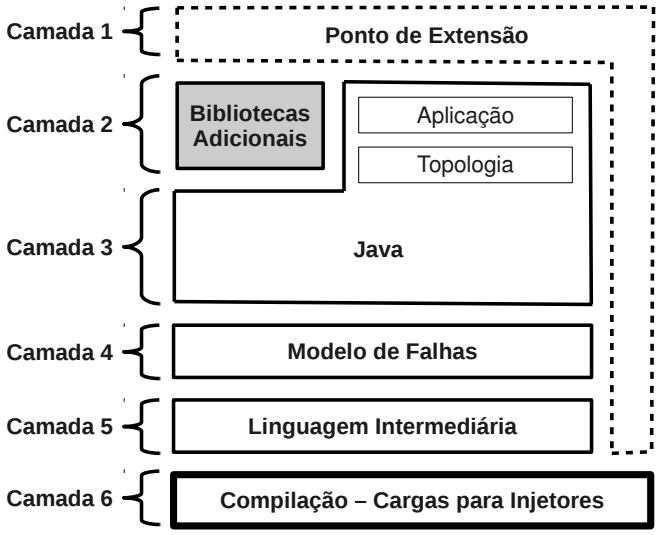

Figura 4.1: Camadas do jFaultload

A **camada 1** engloba o *ponto de extensão* do framework jFaultload. Em outras palavras, esta é a camada responsável pela *instanciação* deste framework, de forma a torná-lo *completo* e *pronto para execução*, considerando o contexto de uma ou mais ferramentas de injeção de falhas. Devido a sua importância para a funcionalidade efetiva do ambiente, o ponto de extensão também se faz presente no contexto das próximas 4 camadas adjacentes. Entretanto, a *especificação* desta extensão ocorre apenas na camada 1, sendo que as camadas seguintes fazem apenas uso do que foi definido, sem a realização de modificações posteriores.

Na **camada 2**, fazem parte o conjunto de entradas denominadas como *entradas complementares*. Elas possuem este nome pelo fato de fornecerem informações adicionais (mas não menos importantes) sobre o experimento a ser realizado. Assim, estas entradas correspondem, primeiramente, às *bibliotecas adicionais* que, porventura, podem ser necessárias durante a realização de algum experimento. Vale ressaltar que estas bibliotecas são de uso *opcional*, não sendo ativadas caso o experimento não as utilize. Adicionalmente, fazem parte das *entradas complementares* as informações referentes à *topologia* e à *aplicação alvo*. Ao contrário das bibliotecas, estas informações são essenciais no contexto do ambiente, uma vez que elas referenciam *o que* vai ser testado (caso da aplicação alvo), bem como *onde* será testado (caso da topologia).

Seguindo a descrição, a **camada 3** envolve a *entrada fundamental* do ambiente jFaultload. Esta entrada, por sua vez, é referente a carga de falhas em *Java*, especificada pelo usuário do ambiente (neste caso, o engenheiro de testes). Considerando a arquitetura como um todo, esta é a camada de maior interação com o engenheiro de testes (considerado como usuário direto e efetivo da ferramenta). Isto ocorre porque é nesta camada que a especificação de cargas detalhadas de falhas pode ser realizada, o que é o objetivo principal do ambiente proposto. Além disso, esta é a última camada (considerando as duas camadas descritas nos parágrafos anteriores) no qual existe interação externa com algum usuário do ambiente (seja engenheiro de testes ou administrador).

A **camada 4** é a primeira camada de *uso interno* do jFaultload. Neste sentido, a respectiva camada delimita o *modelo de falhas* suportado pelo ambiente, no qual a camada imediatamente superior (referente à carga de falhas em Java) faz uso para a especificação de cargas de falhas. Por consequência, esta camada fornece todos os subsídios necessários para que a carga de falhas possa ser especificada e, posteriormente, injetada em um dado experimento, da maneira desejada pelo engenheiro de testes.

A **camada 5**, também de uso interno no ambiente, aborda a especificação de uma *linguagem intermediária*. Esta especificação consiste em uma linguagem de baixo nível, cujo principal objetivo é facilitar a transformação de uma carga de falhas descrita em Java (foco deste trabalho) para uma carga de falhas correspondente ao injetor utilizado. Para isso, esta linguagem possui uma construção simplificada, utilizando apenas uma *tabela de símbolos* (para armazenamento de dados) e sete comandos. Estes comandos são descritos nos itens que seguem.

- INSERT: adiciona um valor na tabela de símbolos, para posterior uso e avaliação.
  Sintaxe: INSERT “<variável>” “<valor>”

- RUN: executa uma aplicação alvo, a ser avaliada no contexto de um experimento de testes. Opcionalmente, podem ser informados um ou mais parâmetros de inicialização para esta respectiva aplicação.
  Sintaxe: RUN “<aplicação>” [“<parâmetros>”]

- RAND: gera um número aleatório, a partir de um valor previamente configurado.
  Sintaxe: RAND “<semente>”

- EVAL: avalia uma expressão, direcionando a execução para a linha da linguagem intermediária representada por <linha_expr_verd> (caso expressão seja verdadeira) ou <linha_expr_falsa> (caso seja falsa).
  Sintaxe: EVAL “<expressão>” ? <linha_expr_verd> ! <linha_expr_falsa>

- GOTO: direciona a execução para alguma linha da linguagem intermediária.
  Sintaxe: GOTO “<linha>”

- DROP: descarta o pacote especificado pelo comando.
  Sintaxe: DROP “<pacote_a_ser_descartado>”

- DELAY: aplica um atraso ao pacote especificado pelo comando.
  Sintaxe DELAY “<pacote_a_ser_atrasado>” “<atraso>”

Finalmente, a **camada 6** abrange a camada de mais baixo nível da arquitetura. Nesta camada, é realizada a *compilação* da carga de falhas gerada, fazendo uso das abordagens definidas nas camadas superiores. Estas abordagens referem-se à *linguagem intermediária* (gerada a partir da carga e modelo de falhas) e ao *ponto de extensão* (que fornece subsídios para a realização efetiva da compilação). Como produto desta camada, tem-se a carga de falhas para injetores, que pode ser utilizada diretamente nas respectivas ferramentas de teste, formando assim as saídas fundamentais do ambiente.

4.2 Visão em Blocos

A partir das camadas que integram a arquitetura do jFaultload, é possível constatar algumas funcionalidades em comum entre as mesmas. Neste contexto, a seção atual mostra a arquitetura sob uma visão diferente, através da divisão em *blocos*, baseando-se assim em um nível menor de granularidade, se comparado com a divisão em camadas, mencionada na seção anterior. A figura 4.2 ilustra essa divisão por blocos, sendo cada bloco explicado logo a seguir.

Inicialmente, o **bloco 1** envolve todas as *entradas* necessárias para o pleno funcionamento do ambiente. Logo, este bloco contempla todos os componentes das três primeiras camadas. Estes componentes abrangem o *Ponto de Extensão* (camada 1), as *Bibliotecas Adicionais*, a *Aplicação* e a *Topologia* (referentes a camada 2), bem como a carga de falhas em *Java* (da camada 3).

O **bloco 2**, por sua vez, agrupa as tarefas pertinentes a todo o *processamento* realizado, de forma que o framework jFaultload consiga realizar a execução dos experimentos de testes. Neste sentido, o respectivo bloco abrange as camadas 4 e 5 da arquitetura, envolvendo os componentes *Modelo de Falhas* e *Linguagem Intermediária*. Vale ressaltar que o componente *Ponto de Extensão*, mesmo sendo da camada 1, possui uma participação indireta neste bloco, através do utilização de suas primitivas na especificação e criação dos modelos de falhas, bem como na linguagem intermediária em questão.

Ao final, o **bloco 3** representa as *saídas* do ambiente. Em outras palavras, este bloco é o responsável pela geração final do produto resultante da execução do jFaultload. Por conseguinte, este bloco é formado pela única e última camada restante da arquitetura,

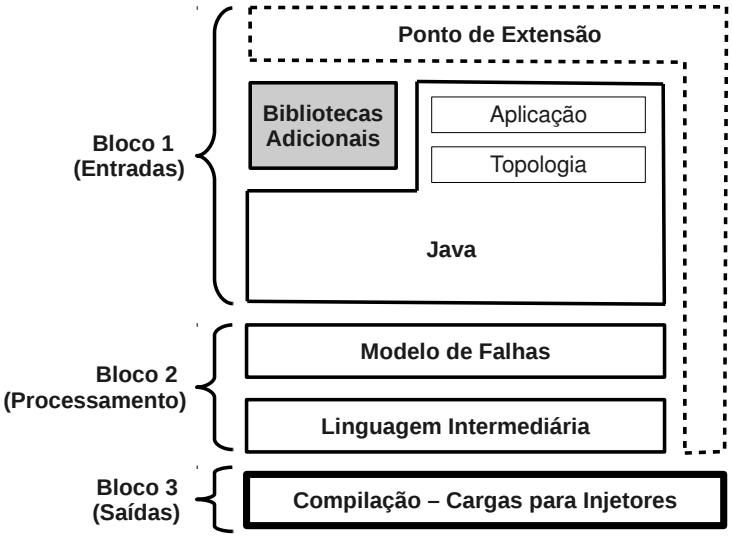

Figura 4.2: Camadas do jFaultload, agrupadas por *blocos*

referente à camada 6, sendo assim encarregado da compilação da linguagem intermediária para, posteriormente, realizar a geração de cargas de falhas envolvendo um ou mais injetores sob experimentos de testes.

4.3 Relacionamento com o Modelo do Ambiente

A arquitetura do jFaultload, como já ilustrada e explicada, possui características que a tornam *modular*, *enxuta* e *extensível*. No entanto, para um melhor entendimento dos conceitos que a envolvem, especialmente considerando a sua modelagem, uma *associação* entre as definições realizadas no modelo do ambiente (visto no capítulo 3) faz-se necessária. Neste sentido, a seção atual ilustra as *equivalências* existentes entre a *arquitetura* e o *modelo* do framework, realizando assim uma *união* entre ambas as abordagens. A figura 4.3 ilustra estas equivalências, que são detalhadas nos parágrafos seguintes, agrupados por camadas, componentes da arquitetura e elementos do modelo.

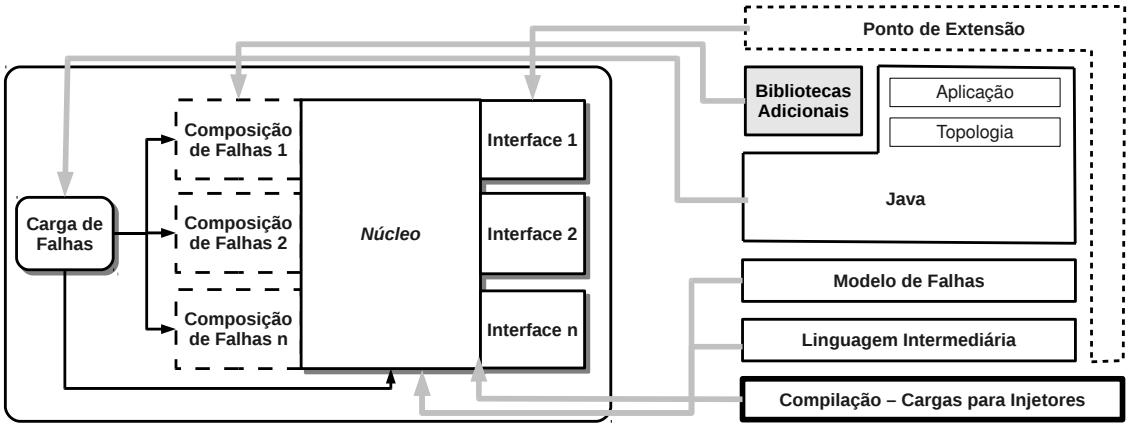

Figura 4.3: Equivalências entre o *modelo* e as *camadas* do jFaultload

A **camada 1**, referente ao *Ponto de Extensão*, está diretamente associada com a **Interface do Ambiente**. Isto se justifica pelo fato de que este ponto de extensão realiza a

instância do framework, tornando-o *completo* para execução, o que é o mesmo propósito da interface no contexto do modelo. Ainda neste caminho, ambos necessitam da especificação por um *administrador*, de forma a tornar esta tarefa mais fácil e acessível para um engenheiro de testes.

O componente *Bibliotecas Adicionais* (referente à **camada 2**), por sua vez, associa-se diretamente com o elemento **Composição de Falhas**. Esta associação ocorre porque ambos são de uso *opcional* em seus respectivos contextos. Além disso, os mesmos foram criados com o mesmo intuito, relacionado a *complementar* a descrição da carga de falhas do ambiente, através de funcionalidades adicionais e personalizadas, especificadas pelo próprio usuário do ambiente.

Já o componente *Java*, juntamente com os componentes *Aplicação* e *Topologia* (todos estes envolvidos nas **camadas 2 e 3**) estão associados ao elemento **Carga de Falhas** do modelo. Neste caso, os componentes citados, bem como o respectivo elemento do modelo, referem-se à *entrada principal* do ambiente, diretamente utilizada pelo engenheiro de testes. Uma pequena diferença entre o modelo e a arquitetura neste quesito diz respeito aos componentes *Aplicação* e *Topologia*: enquanto que, no modelo, ambos são especificados de forma "externa", na arquitetura eles estão embutidos, de forma a permitir uma melhor modularização. No entanto, o significado destes componentes em ambas as abordagens é análogo, sem prejuízo ao entendimento dos conceitos envolvidos.

Os componentes *Modelo de Falhas* e *Linguagem Intermediária* (pertencentes às **camadas 4 e 5**) estão ambos associados ao elemento do modelo denominado **Núcleo**. Isto é necessário porque ambos os componentes correspondem a *funcionalidades internas* do framework que, por sua vez, fornecem suporte às funcionalidades *externas*. Ao mesmo tempo, estes componentes são os responsáveis pelo *processamento* do ambiente, a partir dos dados já fornecidos.

Finalmente, a **camada 6** (representada pelo componente de *compilação*) é associada a dois elementos do modelo: *Núcleo* e *Interface*. Isto acontece devido ao fato de que o processo de compilação exige informações referentes a estes dois elementos. No caso da arquitetura, estas informações são fornecidas tanto pela *Linguagem Intermediária* (da camada 5 e associada ao Núcleo), como também pelo *Ponto de Extensão* (da camada 1 e associada à Interface).

4.4 Funcionamento

Com a arquitetura definida e esquematizada, seu respectivo funcionamento pode ser detalhado de forma clara e precisa. Por este motivo, a figura 4.4 esboça, de forma completa, o funcionamento do framework jFaultload. Como é possível constatar, as especificações da *topologia* e *aplicação alvo* são fornecidas como **entradas** do ambiente, assim como a própria *carga de falhas* (especificada na linguagem Java). Estas entradas são submetidas a uma série de *passos de compilação* que, por sua vez, abrangem cada componente da arquitetura descrita anteriormente, bem como cada elemento do modelo definido no capítulo anterior.

Como resultado desse processo de funcionamento, espera-se que uma ou mais cargas de falhas sejam geradas. Neste caso, é gerada uma carga para cada injetor a ser utilizado pelo engenheiro de testes, considerando a condução de um dado experimento de injeção de falhas. Neste sentido, as subseções seguintes (4.4.1, 4.4.2 e 4.4.3, respectivamente) explicam em detalhes todas as operações realizadas em cada um dos passos de compilação existentes.

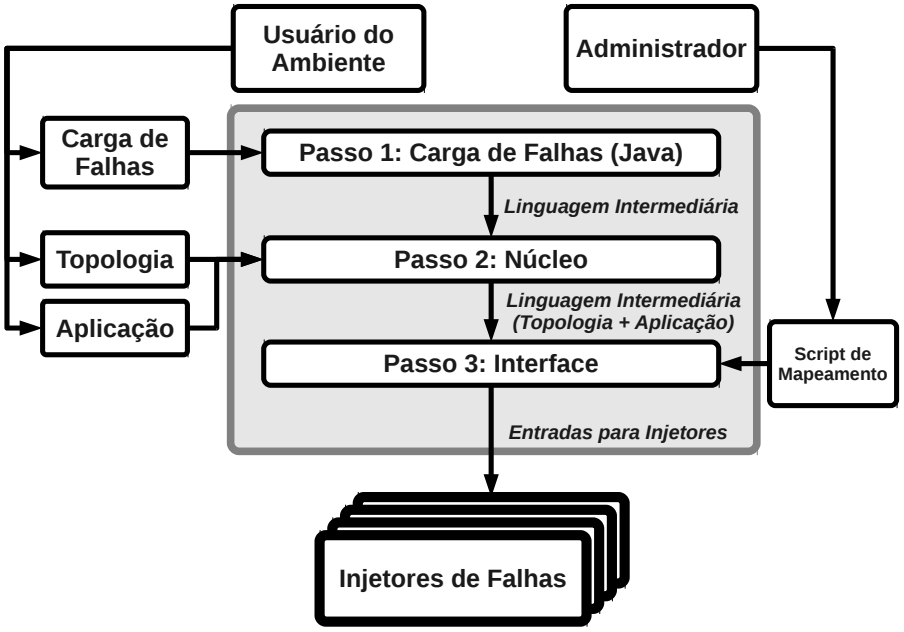

Figura 4.4: Passos de compilação do jFaultload

4.4.1 Passo 1

No *passo 1* de compilação, uma **carga de falhas** em Java deve ser especificada. Nesta carga, realizada pelo usuário direto do ambiente (representado aqui pelo engenheiro de testes), é especificada todo o cenário de falhas do experimento a ser executado, bem como as falhas a serem injetadas. Para isso, o respectivo usuário deve possuir o conhecimento da linguagem Java a ser utilizada no ambiente (que não corresponde, necessariamente, a todas as funcionalidades existentes na linguagem Java original), além de outras características e particularidades, referentes ao caso de teste em questão.

Para efeito de demonstração sobre como executar este passo de compilação, a tabela 4.1 ilustra a definição de três cargas de falhas. Cada uma destas cargas ilustra um determinado caso de teste. Os parágrafos seguintes explicam em detalhes como estas cargas podem ser descritas e geradas, de forma efetiva, no contexto do jFaultload.

Tabela 4.1: Exemplos de cargas de falhas, geradas no Passo 1

Omissão

```
import env.Node;

public class Omissao {
  void main(String[] args) {
    Node n=new Node($Topology);
    n.setApp($Application);
    n.omission(0.05);
  }
}
```

Temporização

```
import env.Node;
import env.Path;

public class Temporizacao {
  void main(String[] args) {
    Node c=new Node($T1);
    Node s=new Node($T2);

    Path p1 = new Path(c, s);
    Path p2 = new Path(s, c);

    s.setApp($Application);

    p1.timing(0.1);
  }
}
```

Crash/Drop

```
import env.*;

public class Crash_Drop {
  void main(String[] args) {
    Node c=new Node($T1);
    Node s=new Node($T2);

    Path p1 = new Path(c, s);
    Path p2 = new Path(s, c);

    s.setApp($Application);

    Packet pkt_p2=p2.head();
    while (pkt_p2.data()=="ACK")
    {
      Packet pkt_p1=p1.head();
      if (pkt_p1.data()!="SEND")
      {
        pkt_p1.drop();
      }
    }

    c.crash();
  }
}
```

Na coluna **Omissão**, é ilustrada uma carga de falhas que visa injetar uma falha de omissão. Neste caso, o componente alvo desta injeção é um nodo, representado pela variável "n". Além disso, o método "setApp" permite especificar a aplicação alvo que será executada durante o experimento de injeção de falhas. Finalmente, a chamada à falha de omissão possui um parâmetro de inicialização, indicando a taxa de omissão de pacotes em trânsito neste nodo (configurada como 5%).

Como exemplo de injeção de falhas em um caminho, a coluna **Temporização** mostra a injeção de uma falha de temporização em um caminho. Neste exemplo, é ilustrada a criação de uma topologia completa, porém simples, envolvendo apenas um cliente e um servidor. A conexão entre estes dois nodos é realizada por um canal bidirecional, especificado como dois caminhos unidirecionais (representados por "p1" e "p2", respectivamente). No contexto da falha injetada, vale ressaltar que a mesma foi aplicada apenas ao caminho "p1": assim, apenas os pacotes que trafegam do cliente para o servidor são afetados. Quanto a inicialização da respectiva falha, foi configurado um atraso de 0.1 segundos.

Em seguida, a coluna **Crash/Drop** exemplifica a injeção de duas falhas para um dado experimento. Primeiramente, é injetado um descarte em todos os pacotes em tráfego no canal "p2" (que liga o servidor ao cliente). Vale ressaltar que esta falha só é injetada sob certas condições: neste caso, enquanto existir a transmissão de pacotes com o dado "ACK" do servidor para o cliente (caminho "p2") e ocorrer a transmissão de um pacote "SEND" do cliente para o servidor (caminho "p1"). Logo após, ou seja, quando for encerrada a transmissão de pacotes "ACK" anteriormente descrita, é injetada uma falha de colapso no nodo cliente.

Então, considerando o funcionamento do ambiente a partir destas cargas de falhas, na execução do passo 1 de compilação, temos a geração da *linguagem intermediária*. Neste contexto, a tabela 4.2 apresenta esta primeira fase da tradução, onde cada classe mencionada foi submetida ao compilador Java existente no ambiente, tendo como resultado a criação destes scripts. Vale ressaltar que este script não necessita de intervenção humana - neste caso, a única necessidade de contato com estas chamadas ocorre na elaboração dos scripts de mapeamento, como já mencionado anteriormente.

Tabela 4.2: Scripts intermediários, gerados a partir das cargas de falhas do passo 1

Omissão

```
1 INSERT n "$Topology"
2 INSERT app "$Application"
3 INSERT omissao "95"
4 INSERT total "100"
5 RUN app
6 RAND omissao total
7 EVAL "omissao-total<0"? 8 ! 6
8 DROP "n1"
9 GOTO 6
```

Temporização

```
 1 INSERT c "$T1"
 2 INSERT s "$T2"
 3 INSERT p1 "P1($T1,$T2)"
 4 INSERT p2 "P2($T2,$T1)"
 5 INSERT app "$Application"
 6 INSERT temp "99"
 7 INSERT total "100"
 8 RUN app
 9 RAND temp total
10 EVAL "temp-total<0" ? 11 ! 9
11 DELAY "p1"
12 GOTO 9
```

Crash/Drop

```
 1 INSERT c "$T1"
 2 INSERT s "$T2"
 3 INSERT p1 "P1($T1,$T2)"
 4 INSERT p2 "P2($T2,$T1)"
 5 INSERT app "$Application"
 6 RUN app
 7 INSERT msg1 "ACK"
 8 INSERT msg2 "SEND"
 9 EVAL "p2 == msg1" ? 10 ! 12
10 EVAL "p1 == msg2" ? 11 ! 9
11 DROP "p1"
12 DROP "c"
```

É possível constatar, tanto na carga de falhas quanto nos scripts intermediários mostrados, alguns rótulos iniciados por "$" (como **`$Topology`** e **`$Application`**). O objetivo destes rótulos é o de, justamente, *separar* as especificações da *topologia* e *aplicação alvo* da especificação da carga de falhas, de forma a facilitar o trabalho do engenheiro de testes durante a especificação dos cenários, bem como aumentar o reuso das respectivas carga de falhas. Assim, a substituição destes rótulos por dados concretos é realizada posteriormente, através do passo 2 de compilação, que será descrito logo em seguida.

4.4.2 Passo 2

No contexto do segundo passo de compilação, é envolvido o elemento *Núcleo* (referente ao modelo do ambiente), assim como os componentes *Aplicação*, *Topologia* e *Linguagem Intermediária* (referentes à arquitetura). Assim, o principal objetivo desta fase de compilação consiste em *complementar* a linguagem intermediária anteriormente gerada no passo 1. Neste complemento, as informações de *aplicação* e *topologia* são adicionadas, deixando assim o script pronto para uma execução posterior.

Considerando as informações de *aplicação* e *topologia*, ambas são informadas na especificação das *entradas*, diretamente pelo engenheiro de testes (o usuário direto do ambiente). Neste sentido, estes dados podem ser informados pelo usuário através de uma *associação*, onde as variáveis que iniciam por "$" podem ser substituídas por um valor correspondente, através do sinal "->" (que, no ambiente, tem o significado de "associação"). A tabela 4.3 mostra a especificação destas entradas, considerando os scripts intermediários gerados no passo anterior.

Tabela 4.3: Especificações de *aplicação* e *topologia*

Omissão	Temporização	Crash/Drop
$Topology ->** "n1" **$Application -> "/opt/Prog1"	**$T1 ->** "C" **$T2 ->** "S" **$Application ->** "/opt/Prog2"	**$T1 ->** "C" **$T2 ->** "S" **$Application ->** "/opt/Prog2"

Desta forma, a partir das entradas referentes a aplicação e topologia, juntamente com o script intermediário obtido através do passo 1 de compilação, o script intermediário completo pode ser gerado. Neste caso, as informações referentes a aplicação e topologia já estarão nas suas respectivas posições, deixando assim o script pronto para a execução da próxima etapa de compilação. A tabela 4.4 exibe estes scripts intermediários completos.

Tabela 4.4: Scripts intermediários completos, finalizando o passo 2

Omissão	Temporização	Crash/Drop
1 INSERT n "n1" 2 INSERT app "/opt/Prog1" 3 INSERT omissao "95" 4 INSERT total "100" 5 RUN app 6 RAND omissao total 7 EVAL "omissao-total<0"? 8 ! 6 8 DROP "n1" 9 GOTO 6	1 INSERT c "C" 2 INSERT s "S" 3 INSERT p1 "P1(C,S)" 4 INSERT p2 "P2(S,C)" 5 INSERT app "/opt/Prog2" 6 INSERT temp "99" 7 INSERT total "100" 8 RUN app 9 RAND temp total 10 EVAL "temp-total<0" ? 11 ! 9 11 DELAY "p1" 12 GOTO 9	1 INSERT c "C" 2 INSERT s "S" 3 INSERT p1 "P1(C,S)" 4 INSERT p2 "P2(S,C)" 5 INSERT app "/opt/Prog2" 6 RUN app 7 INSERT msg1 "ACK" 8 INSERT msg2 "SEND" 9 EVAL "p2 == msg1" ? 10 ! 12 10 EVAL "p1 == msg2" ? 11 ! 9 11 DROP "p1" 12 DROP "c"

4.4.3 Passo 3

Após dois passos de compilação, o terceiro e último passo têm por objetivo realizar a geração de uma ou mais cargas de falhas, referentes aos injetores de falhas utilizados em um experimento de testes. Por este motivo, a tarefa de *delimitação dos injetores* faz-se necessária nesta etapa, de forma a descobrir quais os injetores que estão sendo utilizados no respectivo experimento, etapa não necessária nas fases anteriores de compilação. Com tudo isso, o resultado final do processamento realizado pelo ambiente pode ser apresentado de forma simples e direta.

Entretanto, antes da geração das cargas referentes aos injetores, o ambiente necessita de um *script de mapeamento*. Como já mencionado anteriormente, este script é necessário para dar prosseguimento à tarefa de tradução de cargas de falhas, com o intuito de fornecer subsídios referentes às respectivas ferramentas de injeção de falhas utilizadas. O script é criado pelo administrador do ambiente, demandando assim pouca ou nenhuma manutenção futura, além de ser *orientado a injetores*, não tendo assim dependência direta de cada carga de falhas elaborada no jFaultload. Para efeito de exemplo, a tabela 4.5 ilustra os scripts de mapeamento utilizados para os injetores FAIL/FCI, MENDOSUS e FIRMAMENT.

Tabela 4.5: Exemplos de scripts de mapeamento, para o passo 3

FAIL/FCI	`INSERT(x,y) : "int $x $y"`
	`RUN(x) : "Computer $x { program = '$x'; daemon = '$xLogic'; }"`
	`RAND(x,y) : "FAIL_RANDOM($x, $y)"`
	`EVAL(x?t!f) : "$x -> $t"`
	`DROP(x) : "stop"`

MENDOSUS	`INSERT(x,y) : ""`
	`RUN(x) : "set_command $i '$x' EXEC $i $x"`
	`RAND(x,y) : "poisson $x"`
	`EVAL(x?t!f) : ""`
	`DROP(x) : "set_fault $x fail_host_power_off TRANSIENT"`

FIRMAMENT	`INSERT(x,y) : "SET $y $x"`
	`RUN(x) : ""`
	`RAND(x,y) : "RND $y $x"`
	`EVAL(x?t!f) : "JMP $x $t"`
	`DROP(x) : "DRP"`

Assim, com a obtenção dos scripts de mapeamento, a última etapa de processamento desta fase de compilação refere-se, exatamente, à geração da carga *real* de falhas, que pode assim ser utilizada diretamente nos respectivos injetores escolhidos para a realização do experimento. Logo após a geração desta carga, o experimento de injeção de falhas estará pronto para ser realizado, considerando a carga de falhas e os respectivos procedimentos de configuração e inicialização dos injetores. A tabela 4.6 ilustra três cargas de falhas, geradas para os injetores FAIL/FCI, MENDOSUS e FIRMAMENT, referentes ao tipo de falha relacionado à *omissão*.

4.5 Conclusões do Capítulo

O capítulo abordou uma proposta de arquitetura para o ambiente de especificação de cargas de falhas jFaultload. Inicialmente, esta arquitetura foi mostrada sob duas perspectivas: a primeira, relacionada a uma visão em *camadas*, enquanto que a segunda abordava uma visão em *blocos*. A principal diferença entre essas duas abordagens, além do nível de granularidade, está relacionada diretamente ao agrupamento de funcionalidades semelhantes (algo presente na visão em blocos), bem como na visualização por níveis de complexidade (o que é marcante na visão em camadas).

Em seguida, foram abordadas as formas pelas quais os conceitos vistos na arquitetura se relacionam com a respectiva modelagem do ambiente. Isto foi importante para, principalmente, realizar uma unificação de conceitos. Entretanto, tal associação proporcionou também um melhor entendimento da abordagem que foi adotada pelo ambiente jFaultload, facilitando assim um possível aprendizado a ser realizado pelo engenheiro de testes.

Por fim, o funcionamento completo da arquitetura do ambiente foi mostrado. Este funcionamento, denominado como *compilação*, foi dividido em três passos. Cada passo foi explicado em detalhes, sendo o *produto* de cada passo mostrado de forma clara e explícita,

Tabela 4.6: Exemplo de entradas para injetores, referentes à falha de *omissão*

	Omissão

FAIL/FCI	`Daemon Logica_Nodo1 {`
	`  node 1:`
	`  int distribution = FAIL_RANDOM(1, 100);`
	`  distribution <= 5  -> stop, goto 2;`
	`  node 2:`
	`}`
	
	`Computer Nodo1 {`
	`  program = "/opt/Prog1";`
	`  daemon = "Logica_Nodo1";`
	`}`

MENDOSUS	`// Configuração da topologia:`
	`set_host Nodo1 10.0.0.1`
	
	`// Comandos para executar a aplicação alvo:`
	`set_command 0 "/opt/Prog1" EXEC 0 Nodo1`
	
	`// Falhas a serem injetadas:`
	`set_fault Nodo1 ft_host_freeze_rec TRANSIENT poisson 0.05`

FIRMAMENT	`SET  100 R0  ; R0 = 100; // 100%`
	`RND  R0  R1  ; R1 = RND(R0); // Obtém número (semente 100)`
	`SET  95  R0  ; R0 = 95; // 5% de probabilidade`
	`SUB  R1  R0  ; R0 = R0 - R1;`
	`JMPN R0  DROP  ; se R0 é negativo, descarta o pacote`
	
	`DROP:`
	`  DRP`

indicando a sua utilidade para a etapa seguinte, bem como as vantagens proporcionadas pela sua utilização, ressaltando os benefícios gerados no resultado final do processamento do ambiente.

5 PROTÓTIPO DO AMBIENTE

Considerando a arquitetura do ambiente previamente abordada, o capítulo atual visa a construção de um protótipo. Este protótipo, por sua vez, tem o intuito de validar as ideias expostas neste trabalho, servindo como prova de conceito para o uso da abordagem mencionada. Com isso, o protótipo em questão visa demonstrar a facilidade em especificar cargas de falhas.

Será ilustrada, neste capítulo, toda a construção interna que envolve o protótipo implementado. Primeiramente, a seção 5.1 diz respeito às especificações iniciais, que foram utilizadas para a concepção do protótipo. A seção 5.2 mostrará a forma pelo qual os componentes fundamentais, vistos no modelo do ambiente, foram considerados. Ao final, a seção 5.3 abordará algumas conclusões, referentes ao protótipo construído.

5.1 Especificações Iniciais

Esta seção relata as especificações iniciais, necessárias para a construção do protótipo do jFaultload, abrangendo seus principais requisitos. Neste sentido, a seção 5.1.1 descreve em detalhes quais as ferramentas utilizadas para a sua elaboração, enquanto que a seção 5.1.2 mostra o escopo, com ênfase nas funcionalidades que, mesmo previstas no modelo e na arquitetura, não foram tratadas.

5.1.1 Ferramentas Utilizadas

Para sua concepção, o jFaultload utilizou uma série de ferramentas. A principal delas trata-se do **JavaCC** (um acrônimo de *Java Compiler Compiler*). Esta ferramenta se fez necessária, visto que a mesma é adequada para a interpretação de linguagens complexas, ou seja, que envolvem uma sintaxe detalhada. Dentro do contexto de JavaCC, foi utilizada a biblioteca *JJTree*, com o intuito de habilitar as *árvores de sintaxe*, uma vez que esta foi uma demanda que mostrou-se necessária durante a elaboração do respectivo protótipo.

A ferramenta JavaCC, por sua vez, trouxe também a necessidade de utilização de várias outras ferramentas adicionais. Dentre elas, pode ser destaca a própria linguagem *Java* (através da máquina virtual Java, fornecida pela Sun), além do ambiente de desenvolvimento *Eclipse* (que foi fundamental para todas as etapas de desenvolvimento e concepção do protótipo do jFaultload).

Finalmente, para a execução do ambiente, foram utilizadas *máquinas virtuais*, com o intuito de facilitar a configuração dos experimentos, bem como permitir uma análise *heterogênea* dos respectivos injetores de falhas (neste caso, tal análise faz referência a sistemas operacionais de diferentes versões, executando assim diferentes injetores). Neste contexto, foram utilizados as máquinas virtuais *VMWare* e *Virtual Box*, sendo que nelas

foram executados os sistemas operacionais *Linux Slackware 9.1*, com kernel versão 2.4 (para a execução dos injetores FIRMAMENT e FAIL/FCI), bem como o sistema *Red Hat 6.2*, com kernel versão 2.2 (para a execução do injetor MENDOSUS, que exigia uma plataforma mais antiga).

5.1.2 Escopo

Referente ao *escopo* do protótipo do jFaultload, é importante mencionar que o mesmo foi voltado a duas frentes: *carga de falhas* e *script de mapeamento*. Com relação a carga de falhas, ressalta-se que, no protótipo construído para a validação das ideias deste trabalho, duas classes de falhas foram previstas: neste caso, as falhas relacionadas a *colapso* e as falhas relacionadas a *atraso*. As falhas de colapso e atraso de mensagem são suficientemente primitivas para permitir modelar outras classes de falhas, como colapso em nodos e caminhos, bem como falhas de rajadas em caminhos. As falhas de atraso, por sua vez, permitem modelar falhas relacionadas a desempenho. Logo, apenas falhas aleatórias (bizantinas) não podem ser totalmente modeladas com esses tipos primitivos de falhas. Sobre o script de mapeamento, o mesmo também foi implementado *parcialmente* no ambiente, visando o atendimento das necessidades fundamentais, referentes aos experimentos realizados. No entanto, como o ambiente é concebido sob a forma de um *framework*, as funcionalidades faltantes podem ser facilmente implementadas via mecanismo de *extensão*, sem prejuízo às funcionalidades já implementadas.

5.2 Componentes Fundamentais

Com o objetivo de modularizar a implementação do protótipo, os componentes fundamentais do protótipo foram implementados a partir de um conjunto de *classes*. Esse conjunto de classes, por sua vez, segue a abordagem adotada na concepção da arquitetura do ambiente. Nos itens seguintes, serão apresentadas as implementações realizadas no protótipo, separadas por cada componente fundamental, definido anteriormente na arquitetura do ambiente: *File* (seção 5.2.1), *Plugin* (seção 5.2.2), *Table* (seção 5.2.3), *Type* (seção 5.2.4) e *Inicialização do Ambiente* (seção 5.2.5). Em seguida, a seção 5.2.6 apresenta um fluxo de execução referente aos componentes deste protótipo, seguindo a abordagem de *pipeline*.

5.2.1 File

O componente *File* inclui a interface FileParser. Esta interface contém a chamada call(), responsável pela leitura do arquivo de entrada relativo a carga de falhas do experimento. Adicionalmente, uma classe abstrata, denominada como GenericFileParser, implementa as funcionalidades necessárias (com o uso da interface FileParser), além de definir um método específico para a leitura de cada linha do arquivo correspondente a carga de falhas, chamado de getFileLines(). Finalmente, a classe FaultLoadParser efetua a leitura, propriamente dita, do arquivo de carga de falhas, utilizando-se da abordagem de *script* já mencionada anteriormente (através da implementação do método parse()).

Referente a *extensibilidade* deste componente, a mesma é realizada a partir da classe GenericFileParser. Ao mesmo tempo, a classe FaultLoadParser também pode ser estendida, através da adição de novas construções. A figura 5.1 ilustra as classes acima mencionadas, utilizando um diagrama UML.

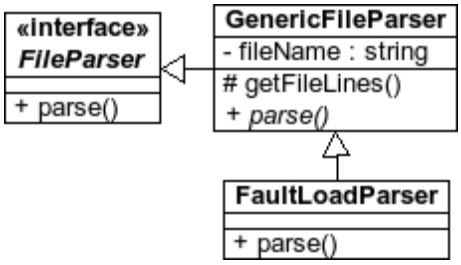

Figura 5.1: Componente *File*

5.2.2 Plugin

Sendo um componente acoplado diretamente a cada injetor de falhas utilizado em um experimento de testes, o ambiente pode fazer uso de uma ou mais instâncias de *plugins*. Ao mesmo tempo, uma vez que um determinado *plugin* é desenvolvido, o mesmo pode ser utilizado em qualquer experimento que seja criado no futuro, desde que envolva o respectivo injetor de falhas associado.

Na implementação do plugin, uma interface é definida (FaultInjectionPlugin). Esta interface inclui um método, denominado generateFaultLoad(). Este método, por sua vez, realiza a geração das cargas de falhas para um determinado injetor de falhas. A figura 5.2 ilustra a modelagem desta classe.

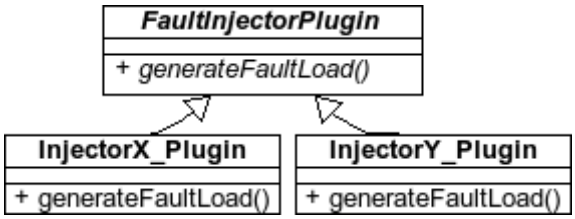

Figura 5.2: Componente *Plugin*

5.2.3 Table

O componente *Table* é implementado a partir de duas classes: FaultLoadLine e FaultLoadTable. A classe FaultLoadLine implementa uma falha a ser injetada em um experimento. Logo, cada *linha* modelada por esta classe é representada através de uma *falha primitiva* (através da classe PrimitiveFault, abordada na próxima seção), seguida dos parâmetros necessários para a execução desta falha. Já a classe FaultLoadTable cria a instância da tabela em si, formada essencialmente por uma lista de objetos dotipo FaultLoadLine.

No contexto do ambiente, a tabela é implementada através do padrão de projeto *singleton* (ou seja, existe apenas uma instância desta tabela por execução). Finalmente, a classe que modela a tabela inclui um método chamado loadTable(), responsável pela transmissão desta tabela ao leitor das cargas de falhas (neste caso, a classe FaultLoadParser). A estrutura das classes mencionadas é mostrada na figura 5.3.

5.2.4 Type

O componente Type está relacionado à definição do *modelo* das cargas de falhas que podem ser configuradas em uma determinada instância de execução do ambiente. Assim,

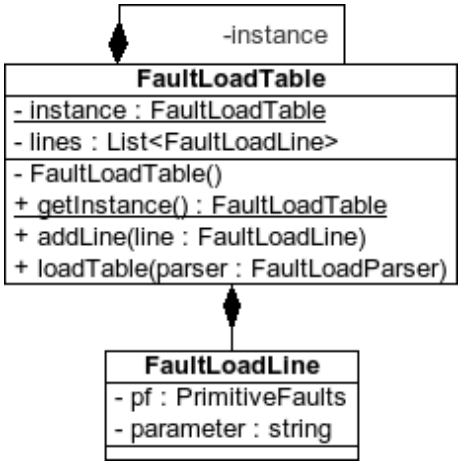

Figura 5.3: Componente *Table*

as falhas primitivas a serem utilizadas em um experimento devem, primeiramente, ser definidas neste componente. Ao mesmo tempo, este componente prevê também a definição das falhas compostas, de forma que as mesmas possam ser utilizadas nos experimentos de forma efetiva.

Considerando a implementação do protótipo do ambiente, o componente Type é introduzido a partir da classe EnvironmentTypes. Nesta classe, é definida uma enumeração denominada PrimitiveFaults, que abrange o conjunto de todas as falhas primitivas atualmente suportadas pelo ambiente. Desta maneira, caso exista a necessidade de se adicionar mais falhas primitivas, assim como falhas compostas que sejam baseadas nas falhas primitivas já criadas, novas enumerações podem ser criadas nesta classe. A figura 5.4 ilustra em detalhes esta classe.

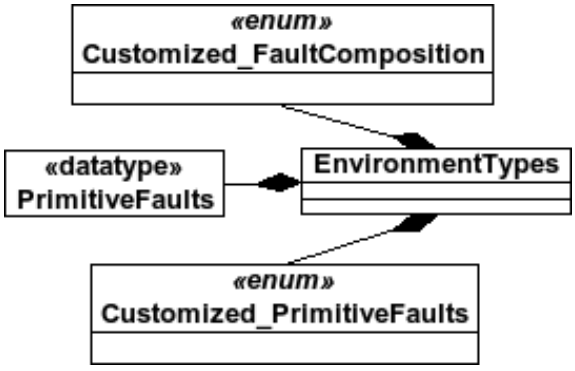

Figura 5.4: Componente *Type*

5.2.5 Inicialização do Ambiente

Como o nome indica, este componente representa o ponto inicial de execução do ambiente. Neste sentido, a classe responsável pela inicialização do ambiente implementa o método main(), responsável por todas as configurações iniciais de um dado experimento. Dentre estas configurações, podem ser destacados os arquivos de entrada (referentes às carga das falhas a serem injetadas), bem como as informações adicionais referentes à aplicação alvo sob teste.

Além disso, a classe referente à inicialização do ambiente visa agrupar todos os de-

mais componentes integrantes do ambiente, bem como ordenar a execução dos mesmos, em cada caso. Neste sentido, por ser uma classe que difere de forma acentuada a cada experimento, a mesma é definida diretamente pelo usuário do respectivo ambiente. Por este motivo, esta classe de inicialização não é implementada pelo ambiente de forma direta.

5.2.6 *Pipeline* de Execução

Nas seções anteriores, as funcionalidades de cada um dos componentes do ambiente são explicadas, abrangendo as principais classes envolvidas. Com estes conceitos em mente, a presente seção descreve as interações que ocorrem entre estes componentes, assim como o funcionamento do ambiente como um todo. Com o intuito de ilustrar este comportamento, um *pipeline* de execução é mostrado na figura 5.5.

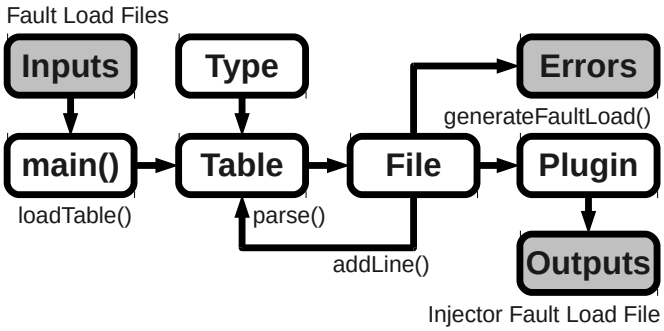

Figura 5.5: *Pipeline* de execução do ambiente

Primeiramente, o componente de *inicialização* (main()) recebe como entrada um ou mais arquivos (Inputs), referentes às cargas de falhas que serão utilizadas em um experimento. Em seguida, já na execução do método main(), o ambiente invoca a chamada do método loadTable(), a fim de que seja carregada a tabela de falhas a serem injetadas. Desta forma, a execução é passada para o componente Table.

O componente Table, por sua vez, executa o método parse() da classe File. Neste ponto, existe uma interação direta entre os componentes Table e File: para cada linha obtida pelo componente File, a respectiva linha será adicionada ao componente Table, através da chamada addLine(). Toda esta etapa, referente a leitura do arquivo de carga de falhas, é baseada no componente Type, que define os tipos de falhas que serão injetadas durante um experimento.

Nesta fase de leitura da carga de falhas, a execução pode ser desviada para um local específico, caso ocorram *erros* nesta leitura. Assim, quando erros acontecerem, mensagens sobre os respectivos erros são mostradas, de forma a orientar o usuário do ambiente, informando os passos necessários para a correção dos respectivos erros. Nestas condições, o ambiente finaliza a sua execução.

Finalmente, caso não ocorram erros durante a leitura da carga de falhas, tanto as tabelas como as demais estruturas do ambiente estarão prontas para a próxima fase, envolvendo a chamada ao método generateFaultLoad(). Depois disso, o controle é passado ao componente Plugin, que inicia a geração de uma carga de falhas apropriada ao respectivo injetor de falhas utilizado. Esta fase também está sujeita a erros, tendo assim o mesmo comportamento descrito anteriormente. Como saída ("Outputs"), o ambiente realiza a geração das cargas de falhas para cada injetor de falhas utilizado no experimento de testes.

5.3 Conclusões do Capítulo

Este capítulo relatou a implementação de um protótipo do jFaultload. Inicialmente, as especificações iniciais deste protótipo foram abordadas, através do relato das principais ferramentas utilizadas, bem como do escopo do referido protótipo. Seguindo o detalhamento do protótipo, cada componente fundamental integrante deste protótipo foi descrito de forma exaustiva, com um diagrama para cada componente, tendo ao final a elaboração de um *pipeline* completo de execução. A partir disso, percebe-se que o protótipo contemplou boa parte do que foi proposto no modelo e na arquitetura do ambiente, estando assim preparado para a execução de experimentos, cujo tema será abordado no capítulo seguinte.

6 VALIDAÇÃO DO AMBIENTE

A partir do protótipo implementado, o capítulo atual visa demonstrar alguns exemplos de utilização do ambiente proposto. Neste sentido, a seção 6.1 mostra como as falhas, a serem injetadas em um experimento, podem ser *ativadas* no contexto do ambiente. Logo após, a seção 6.2 ilustra um exemplo de criação de plugins, realizada pelo administrador da ferramenta. Em seguida, a seção 6.3 aborda a criação de cargas completas de falhas, utilizando-se da linguagem Java adotada neste contexto. Finalmente, a seção 6.4 ilustra um experimento com o jFaultload, envolvendo o injetor FIRMAMENT. Conclusões sobre o capítulo são abordadas na seção 6.5.

6.1 Ativação de Falhas

Considerando o modelo do ambiente proposto, juntamente com o auxílio da arquitetura estendida, a execução do ambiente deve ser realizada a partir de entradas simples, de forma que o nível de complexidade seja o mais baixo possível. Para isso, a ativação das falhas a serem injetadas em um experimento devem seguir essa diretriz, ou seja, devem ser de simples utilização e configuração. Neste sentido, o principal objetivo está voltado a usabilidade da interface, principal premissa do ambiente.

Figura 6.1: Exemplo de carga de falhas genérica.

Para efeito de exemplo, nas figuras 6.1 e 6.2 são ilustradas as ativações de uma falha de *colapso*, seguida de uma falha de *omissão* com descarte de 10% dos pacotes. Para isso, é necessária apenas a criação de duas entradas: a **carga de falhas** e o **mapeamento de falhas**. Referente ao mapeamento de falhas, vale ressaltar a realização de um mapeamento para cada injetor específico - no contexto deste exemplo, para os injetores FIONA (figura 6.2(a)) e FAIL/FCI (figura 6.2(b)).

6.2 Criação de Plugins

Esta seção descreve a criação de plugins para o ambiente, bem como a execução dos mesmos no contexto de um experimento, envolvendo injetores de falhas. Assim, considerando esta execução, serão geradas cargas de falhas para três diferentes ferramentas: FAIL/FCI, MENDOSUS e FIRMAMENT. A escolha destas ferramentas é justificada pelo

(a) FIONA (b) FAIL/FCI

Figura 6.2: Exemplos de scripts de mapeamento.

argumento de que cada uma delas possui uma abordagem diferente para a descrição de cargas de falhas.

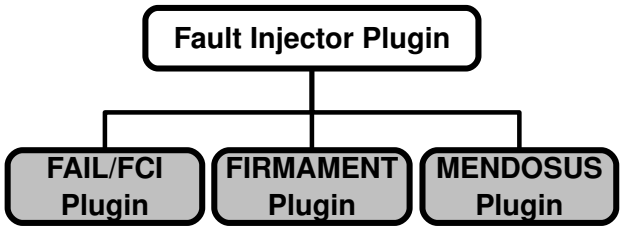

Figura 6.3: Classes referentes aos *plugins* do ambiente

Considerando a criação, propriamente dita, das cargas de falhas, como um primeiro passo desta etapa, devem ser criados os *plugins* associados a cada um dos três injetores de falhas que serão utilizados. Como já mencionado anteriormente, este passo é realizado pelo administrador do ambiente. Assim, para a execução desta tarefa, uma classe é criada para cada injetor, sendo que todas estas classes devem implementar a interface FaultInjectorPlugin. A figura 6.3 ilustra como estas classes são criadas, considerando a hierarquia das mesmas.

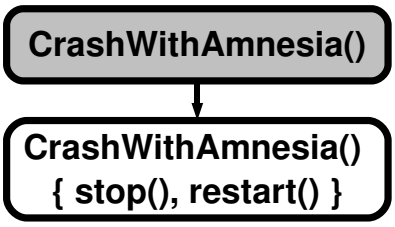

Figura 6.4: Criação de uma falha composta

No passo seguinte, o usuário do ambiente possui subsídios para, então, criar uma *falha composta* que, por sua vez, é acompanhada de uma *falha genérica*. Para um melhor entendimento desta abordagem, foi modelado neste exemplo a criação de uma falha composta, denominada como *CrashWithAmnesia*. Esta falha composta é formada por duas falhas simples, correspondentes às falhas *stop* e *restart*, respectivamente. A figura 6.4 ilustra a criação desta falha composta.

Ao final, a criação dos plugins, juntamente com a definição de uma falha composta, demonstram que as cargas de falhas, relacionadas a cada injetor de falhas, podem ser criadas de uma maneira simples e direta. Desta forma, os usuários do ambiente não necessitam aprender as especificações de cargas de falhas relacionadas a cada injetor de falhas utilizado - apenas o administrador desempenha esta tarefa, e em momentos específicos e periódicos. A tabela 6.1 esquematiza as cargas de falhas criadas pelo ambiente, considerando o exemplo descrito nos parágrafos anteriores.

Tabela 6.1: Cargas de falhas geradas pelo ambiente

Fault Injector	Faultload
FAIL/FCI	Daemon CrashWithAmnesia { ?ok -> halt, restart; }
FIRMAMENT	CrashWithAmnesia: DRP JMP CrashWithAmnesia
MENDOSUS	set_fault Client ft_app_crash STICKY set_fault Server ft_app_crash STICKY

6.3 Carga de Falhas em Java

Esta seção descreve alguns exemplos de cargas de falhas que podem ser descritas e geradas no ambiente mencionado, através da linguagem definida na seção anterior. Para isso, foram definidas três cargas de falhas, cada uma em uma classe diferente, onde cada carga ilustra um determinado caso de teste. Os parágrafos seguintes explicam em detalhes como estas cargas podem ser descritas e geradas no ambiente.

Na classe relacionada a **Omissão** (figura 6.5), é ilustrada uma carga de falhas que visa injetar uma falha de omissão. Neste caso, o componente alvo desta injeção é um nodo, representado pela variável "n". Além disso, o método "setApp" permite especificar a aplicação alvo que será executada durante o experimento de injeção de falhas (no contexto do exemplo, o programa "/opt/Prog1"). Finalmente, a chamada à falha de omissão possui um parâmetro de inicialização, indicando a taxa de omissão de pacotes em trânsito neste nodo (configurada como 5%).

```
import env.Node;

public class Omissao {
  void main(String[] args){
    Node n = new Node("n1");
    n.setApp("/opt/Prog1");
    n.omission(0.05);
  }
}
```

Figura 6.5: Carga de falhas referente a *omissão*

Como exemplo de injeção de falhas em um caminho, a classe **Temporização** (figura 6.6) mostra a injeção de uma falha de temporização em um caminho. Neste exemplo, é ilustrada a criação de uma topologia completa, porém simples, envolvendo apenas um cliente e um servidor. A conexão entre estes dois nodos é realizada por um canal bidirecional, especificado como dois caminhos unidirecionais (representados por "p1" e "p2", respectivamente). No contexto da falha injetada, vale ressaltar que a mesma foi aplicada apenas ao caminho "p1": assim, apenas os pacotes que trafegam do cliente para o servidor são afetados. Quanto a inicialização da respectiva falha, foi configurado um atraso de 0.1 segundos.

Em seguida, a classe **Crash/Drop** (figura 6.7) exemplifica a injeção de duas falhas simultâneas para um dado experimento. Primeiramente, é injetado um descarte em todos os pacotes em tráfego no canal "p2" (que liga o servidor ao cliente). Vale ressaltar que esta falha só é injetada sob certas condições: neste caso, enquanto existir a transmissão

```
import env.Node;
import env.Path;

public class Temporizacao {
  void main(String[] args) {
    Node c = new Node("C");
    Node s = new Node("S");

    Path p1 = new Path(c,s);
    Path p2 = new Path(s,c);

    s.setApp("/opt/Prog2");

    p1.timing(0.1);
  }
}
```

Figura 6.6: Carga de falhas referente a *temporização*

de pacotes com o dado “ACK” do servidor para o cliente (caminho “p2”) e ocorrer a transmissão de um pacote “SEND” do cliente para o servidor (caminho “p1”). Logo após, ou seja, quando for encerrada a transmissão de pacotes “ACK” anteriormente descrita, é injetada uma falha de colapso no nodo cliente.

```
import env.*;

public class Crash_Drop {
  void main(String[] args) {
   Node c = new Node("C");
   Node s = new Node("S");
   Path p1 = new Path(c, s);
   Path p2 = new Path(s, c);

   s.setApp("/opt/Prog2");

   Packet pkt_p2=p2.head();
   while(pkP2.data()=="ACK") {
      Packet pkP1=p1.head();
      if (pkP1.data()!="SND") {
        pkP1.drop();
      }
    }
    c.crash();
  }
}
```

Figura 6.7: Carga de falhas referente a *crash* e *drop*

Considerando o funcionamento do ambiente a partir destas cargas de falhas, temos como primeiro passo a geração da *linguagem intermediária*. Neste contexto, a tabela 6.2 apresenta esta primeira fase da tradução, onde cada classe mencionada foi submetida ao compilador Java existente no ambiente, gerando os respectivos scripts de baixo nível. Vale ressaltar que este script não necessita de intervenção humana - neste caso, a única necessidade de contato com estas chamadas ocorre na elaboração dos scripts de mapeamento, discutidos logo em seguida.

Após obter o script intermediário, correspondente à carga de falhas em Java, o ambiente necessita de um script de mapeamento para dar prosseguimento à tarefa de tradução da carga de falhas. Este script é criado pelo instalador do ambiente, demandando assim pouca ou nenhuma manutenção futura. Além disso, este script é orientado *a injetores*, não tendo assim dependência direta de cada carga de falhas elaborada no ambiente. A tabelas

Tabela 6.2: Scripts intermediários, gerados a partir das cargas de falhas

Omissão	1 INSERT n "n1" 2 INSERT app "/opt/Prog1" 3 INSERT omissao "95" 4 INSERT total "100" 5 RUN app 6 RAND omissao total 7 EVAL "omissao-total<0"? 8 ! 6 8 DROP "n1" 9 GOTO 6
Temporização	1 INSERT c "C" 2 INSERT s "S" 3 INSERT p1 "P1(C,S)" 4 INSERT p2 "P2(S,C)" 5 INSERT app "/opt/Prog2" 6 INSERT temp "99" 7 INSERT total "100" 8 RUN app 9 RAND temp total 10 EVAL "temp-total<0" ? 11 ! 9 11 DELAY "p1" 12 GOTO 9
Crash/Drop	1 INSERT c "C" 2 INSERT s "S" 3 INSERT p1 "P1(C,S)" 4 INSERT p2 "P2(S,C)" 5 INSERT app "/opt/Prog2" 6 RUN app 7 INSERT msg1 "ACK" 8 INSERT msg2 "SEND" 9 EVAL "p2 == msg1" ? 10 ! 12 10 EVAL "p1 == msg2" ? 11 ! 9 11 DROP "p1" 12 DROP "c"

6.3 e 6.4 ilustram os scripts de mapeamento e os *triggers* (referentes aos casos não tratados pelo ambiente), respectivamente, utilizados para os injetores FAIL/FCI, Mendosus e FIRMAMENT.

Tabela 6.3: Scripts de mapeamento aplicados aos injetores utilizados

FAIL/FCI	INSERT(x,y) : "int **$x $y**" RUN(x) : "Computer **$x** { program = '**$x**'; daemon = '**$x**Logic'; }" RAND(x,y) : "FAIL_RANDOM(**$x, $y**)" EVAL(x?t!f) : "**$x -> $t**" DROP(x) : "stop"
MENDOSUS	INSERT(x,y) : "" RUN(x) : "set_command **$i** '**$x**' EXEC **$i $x**" RAND(x,y) : "poisson **$x**" EVAL(x?t!f) : "" DROP(x) : "set_fault **$x** fail_host_power_off TRANSIENT"
FIRMAMENT	INSERT(x,y) : "SET **$y $x**" RUN(x) : "" RAND(x,y) : "RND **$y $x**" EVAL(x?t!f) : "JMP **$x $t**" DROP(x) : "DRP"

Finalmente, a última etapa do processamento do ambiente refere-se, exatamente, à geração da carga *real* de falhas, que pode assim ser utilizada diretamente nos respectivos injetores escolhidos para a realização do experimento. Logo, após a geração desta carga, o experimento de injeção de falhas estará pronto para ser realizado, considerando-se a carga de falhas, bem como os respectivos procedimentos de configuração e inicialização dos injetores. A tabela 6.5 ilustra a geração de cargas de falhas para os injetores FAIL/FCI,

Mendosus e FIRMAMENT, considerando a carga de falhas referente à **Omissão**, mencionada no início desta seção.

Tabela 6.4: *Triggers* aplicados aos injetores utilizados

FAIL/FCI	BeforeFaultload : "Daemon **$x**Logic {" BeforeLine : "node **$x**" AfterLine : ", goto **$x;**" AfterFaultload : "}"
MENDOSUS	BeforeFaultload : "set_host **$x $y**" BeforeLine : "" AfterLine : "" AfterFaultload : ""
FIRMAMENT	BeforeFaultload : "MAIN:" BeforeLine : "" AfterLine : "" AfterFaultload : "END:"

6.4 Experimento

Após a geração das cargas de falhas pelo jFaultload, a campanha de testes passa a estar pronta para execução. Esta campanha de testes, por sua vez, é formada por vários experimentos, sendo cada experimento referente à execução de uma dada carga de falhas no injetor de falhas. Para efeito de exemplo, a figura 6.8 ilustra a carga de falhas em Java, utilizada neste experimento. A figura 6.9, por sua vez, ilustra uma carga de falhas criada pelo jFaultload, a partir da carga de falhas anteriormente especificada, para o injetor de falhas FIRMAMENT.

```
class Delay {
   void main() {
      Node n = new Node($Topology);
      Packet p;
      Distrib d = new Distrib("3%");
      while(p=n.getHeadPacket()){
         if(p.getRTPProtocol()){
            if(d.sort()) {
               p.delay(0.1);
            }
         } // if(p.getRTPProtocol())
      } // while
   } // main()
}
```

Figura 6.8: Exemplo de carga de falhas em Java

Tabela 6.5: Carga de falhas gerada para cada injetor

FAIL/FCI

```
Daemon ClientLogic {
   node 1: int prob = FAIL_RANDOM(1,100);
           prob <= 5 -> stop, goto 2;
   node 2:
}
Computer Client {
   program = "/opt/ClientProgram";
   daemon = "ClientLogic";
}
```

MENDOSUS

```
    // Configuration of topology:
    set_host Client 10.0.0.1
    // Target system commands:
    set_command 0 "/opt/ClientProgram"
                EXEC 0 Client
    // Faults that will be injected:
    set_fault Client ft_host_power_off
              TRANSIENT poisson 0.05
```

FIRMAMENT

```
; R0 = 100; // 100%
SET  100 R0
; R1 = RND(R0)
; (sorts a number, with seed 100)
RND  R0  R1
; R0 = 95
; (looking for 5% of probability)
SET  95  R0
; R0 = R0 - R1;
SUB  R1  R0
; if R0 is negative, drops packet (crash)
JMPN R0  DROP
DROP: DRP
```

Assim, referente à campanha propriamente dita, vale ressaltar que o injetor de falhas FIRMAMENT, utilizado neste experimento de testes, executa sua carga de falhas sob cada pacote em trânsito, na pilha do protocolo RTP. Neste tempo, um servidor de *streaming* de vídeo é configurado, tendo neste servidor a execução de um arquivo referente a um vídeo. Assim, este vídeo corresponde à *carga de trabalho* do respectivo experimento de testes sob análise. O servidor de *streaming*, por sua vez, representa a aplicação alvo, que terá o seu comportamento analisado sob a presença de falhas. No momento em que o vídeo encontra-se sob execução, o mesmo pode ser visualizado por um ou mais clientes. Logo, esta visualização faz com que se inicie um *streaming* assim que uma conexão entre os nodos de cliente e servidor seja estabelecida.

Após um curto período de tempo, o injetor FIRMAMENT recebe um comando, indicando que a execução de carga de falhas, traduzida anteriormente pelo jFaultload, pode ser efetivamente executada. A partir deste momento, o experimento de testes é efetivamente iniciado.

No experimento que envolveu a campanha de testes (Munaretti et al., 2010), foram injetadas falhas de *atraso* em 3% dos pacotes em trânsito no servidor de *streaming* de vídeo. O resultado desta injeção pode ser visualizado a partir da captura de uma amostra do vídeo, ilustrada na figura 6.10. Uma captura do monitor de *streaming* (para este experimento, foi utilizado o aplicativo *QuickTime*) é ilustrada na figura 6.11. Neste caso, as perdas apresentadas pelo monitor indicam a existência de uma quantidade de pacotes que foram realmente atrasados pelos injetores de falhas utilizados. O número de pacotes analisados a cada momento não é especificado, entretanto pode ser assumido um número baixo de pacotes, uma vez que o percentual de perda de pacotes é modificado de forma acentuada de uma amostra para outra. Neste sentido, os dados coletados indicam que

```
 1      START:                          23      PROTOCOL_6970:
 2                                      24        SET 500 R0
 3      LOOP_1:                         25        RND R0 R1
 4        SET  9 R0                     26        SET 470 R0
 5        READB R0 R1                   27        ADD R0 R1
 6        SET  17 R0                    28        JMPN R1 DISTRIB_30_OK
 7        SUB  R0 R1                    29      JMP START
 8        JMPZ R1 PROTOCOL_17           30
 9        JMP  START                    31      DISTRIB_30_OK:
10                                      32        SET 0.1 R0
11      PROTOCOL_17:                    33        DLY R0
12        SET  0 R0                     34
13        READB R0 R1                   35      JMP LOOP_1
14        SET 0x0f R0
15        AND R0 R1
16        SET 4 R0
17        MUL R1 R0
18        READS R0 R1
19        SET  6970 R0
20        SUB  R0 R1
21        JMPZ R1 PROTOCOL_6970
22    ACP
```

Figura 6.9: Carga de falhas para o injetor FIRMAMENT

existe, pelo menos, um pacote sob atraso dentro do conjunto de pacotes analisados.

Considerando a amostra de vídeo exibida, o melhor indicador de qualidade refere-se ao critério *subjetivo*, ou seja, relacionado à opinião do espectador do respectivo vídeo. Para obter tal indicador, o melhor método consiste exatamente na exibição do resultado final, referente ao *streaming* de vídeo, para o engenheiro de testes, da mesma forma que é mostrado na figura 6.10. De posse desta visualização, o engenheiro de testes pode, então, julgar se o respectivo comportamento sob falhas é ou não aceitável.

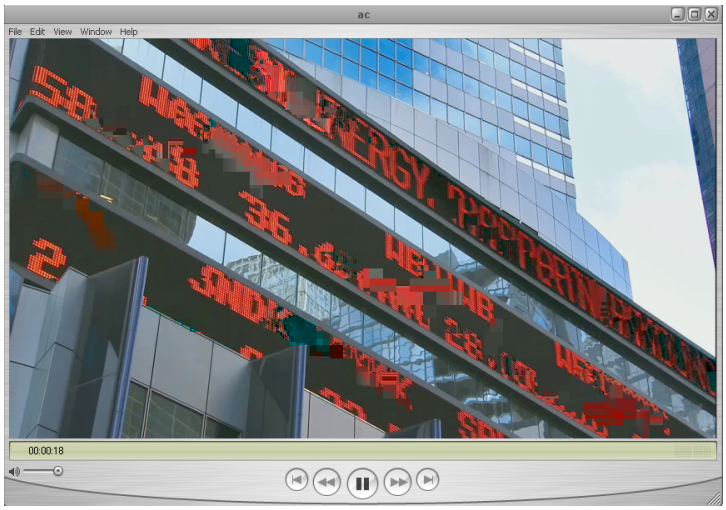

Figura 6.10: Captura de um vídeo com falhas de atraso

Assim, este experimento de testes mostra que uma carga de falhas genérica, escrita em Java e sem referências relacionadas a um injetor de falhas em particular, pode ser automaticamente traduzida em um formato específico de um dado injetor. Este formato, por sua vez, pode ser perfeitamente executado no FIRMAMENT, injetor utilizado na campanha de testes.

Além disso, percebe-se claramente que os passos para a execução dos experimentos, utilizando o jFaultload e o injetor de falhas, são amplamente parametrizáveis. Esta parametrização ocorre na aplicação alvo, bem como nos mecanismos de tolerância a falhas existentes e na própria carga de trabalho em si. Para testar uma aplicação alvo qualquer, é preciso apenas construir um script em Java, onde seja definida uma carga de falhas relevante, de acordo com as configurações desejadas pelo engenheiro de testes. Adicionalmente, uma configuração de hardware/software é necessária (visando representar situações relevantes para a aplicação alvo). Finalmente, a ferramenta de tradução de carga de falhas (referente ao ambiente jFaultload) é executada, juntamente com o injetor de falhas, assim como com os monitores de experimento, que coletam os resultados do respectivo experimento em execução.

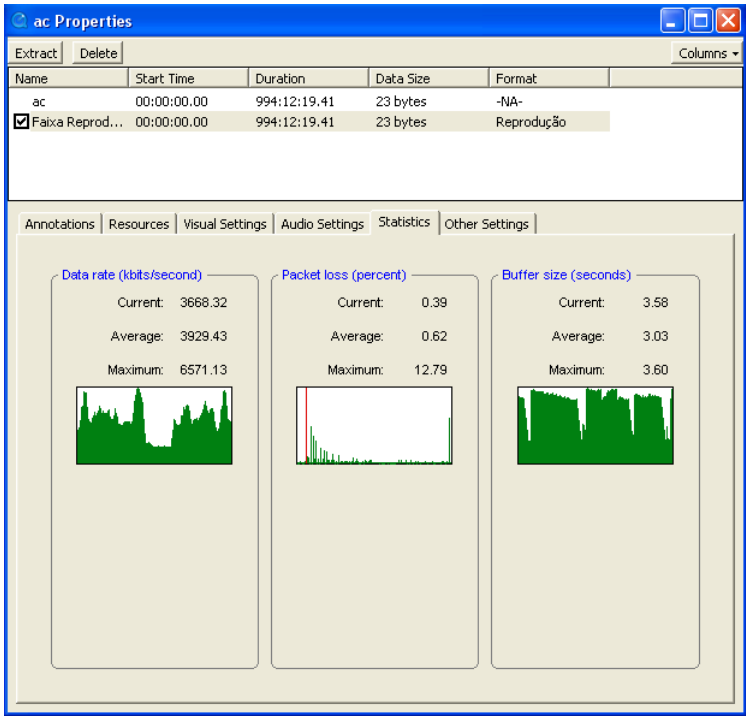

Figura 6.11: Monitor de pacotes do experimento de testes

Logo, o principal esforço existente na realização de um experimento de injeção de falhas está relacionado à criação de uma carga de falhas. Usando o ambiente jFaultload, economiza-se um tempo precioso, relacionado à configuração e tradução de uma descrição de carga de falhas para um dado injetor de falhas. A partir desta economia, as habilidades do engenheiro de testes podem ser aprimoradas, sendo assim voltadas à criação de testes relevantes, visto que este engenheiro passa a se preocupar apenas em como construir descrições de carga de falhas em um nível mais alto, sem depender de como elas são descritas em cada uma das ferramentas de injeção de falhas existentes. Mais ainda, o engenheiro de testes poderia utilizar um outro injetor de falhas, a partir da mesma carga de falhas utilizada na campanha de testes, visando assim aumentar a confiabilidade dos respectivos experimentos, ajudando os desenvolvedores a encontrar, com mais facilidade, erros de software, bem como outras não conformidades.

6.5 Conclusões do Capítulo

Este capítulo abordou uma validação do protótipo do ambiente jFaultload. Neste sentido, primeiramente foi mostrado como as cargas de falhas podem ser especificadas de uma maneira prática. Logo após, a criação de plugins ganhou destaque, abrangendo a forma pelo qual os injetores se encaixam no contexto do ambiente. Além disso, um exemplo de criação de falhas compostas também foi abordado, visando uma facilitação nesta especificação. Em seguida, foi mostrado um conjunto de exemplos práticos que envolveram a especificação de carga de falhas em Java, ilustrando o processo completo de tradução para as cargas de três injetores (FIRMAMENT, FAIL/FCI e MENDOSUS). Finalmente, foi ilustrado um experimento, que envolveu a injeção de falhas, a partir da ferramenta FIRMAMENT, na execução de um *streaming* de vídeo sob o protocolo RTP. Neste experimento, foram injetadas falhas de atraso, com a especificação de uma carga de falhas em Java diretamente no jFaultload. O jFaultload, por sua vez, encarregou-se da tradução desta carga para o injetor de falhas FIRMAMENT, bem como pela execução do experimento em si neste injetor.

7 CONSIDERAÇÕES FINAIS

O trabalho apresentou uma investigação de diversos injetores de falhas conhecidos na literatura. Primeiramente, estes injetores foram descritos visando a funcionalidade dos mesmos: neste caso, foram abordadas as características, a forma de descrição dos cenários de falhas e os pontos de extensibilidade de cada injetor. Em seguida, um estudo comparativo foi apresentado, onde foram relatados os pontos marcantes encontrados nos injetores, tanto na descrição de cenários como nos pontos de extensibilidade dos mesmos.

No contexto dos injetores de falhas, a partir do levantamento realizado, percebe-se que a forma pelo qual uma determinada **carga de falhas** é descrita torna-se *fortemente dependente* do injetor utilizado. Desta forma, o desenvolvedor/projetista da aplicação deve dedicar um esforço adicional na criação destes cenários, transcendendo assim suas tarefas habituais (tais como: definição do modelo de falhas, descoberta dos pontos em que as falhas devem ser injetadas, entre outras), que já possuem complexidade razoável. Por este motivo, a criação de cenários de falhas *consistentes* e *representativos* é um fator que **dificulta** a curva de aprendizado de um injetor de falhas, diminuindo assim a sua usabilidade.

Além disso, ao consideramos o quesito de **extensibilidade** dos injetores atuais, é possível constatar que, apesar dos injetores pesquisados possuírem pontos bem definidos de extensão, a *forma* empregada para a realização desta extensão também é bastante diferente entre cada injetor, assim como ocorre na descrição de cenários. Outro ponto que merece ser destacado diz respeito a *complexidade* em realizar tal extensão, uma tarefa geralmente destinada a desenvolvedores de ferramentas para validação. Neste sentido, uma abordagem mais simplificada para a extensão de injetores poderia ser desejada, caso um determinado experimento exija tal modificação.

A partir da pesquisa realizada, é possível perceber que cada injetor de falhas é implementado para uma realidade de experimentos bastante específica. Desta forma, observa-se a adoção de diferentes critérios para a definição de cenários de falhas, assim como acontece na identificação dos pontos de extensão. Por este motivo, a usabilidade dos injetores como um todo passa a ser prejudicada, devido a curva de aprendizado difícil inerente aos mesmos.

Logo após, foi apresentado um modelo de ambiente para descrição de cenários detalhados de falhas, com ênfase em falhas de comunicação. Foram abordados os elementos principais deste ambiente, com uma descrição detalhada de cada um deles. Além disso, nos elementos de mais alto nível (ou seja, onde ocorre interação direta com o usuário do ambiente), foram ilustrados exemplos de caso de utilização, envolvendo alguns injetores de falhas existentes na literatura.

A partir do modelo do ambiente, foi mostrado que o mesmo apresenta uma arquitetura adequada para a realidade de cenários de falhas, com pontos de extensibilidade bem

definidos. Vale ressaltar também a facilidade de uso do ambiente, a partir de linguagens *script* simplificadas. Com isso, o uso do ambiente torna-se adequado para experimentos que envolvam sistemas baseados em troca de mensagens, independente da complexidade dos mesmos.

Em seguida, foi apresentada uma nova abordagem para a descrição de cargas de falhas, baseada em uma linguagem de programação de alto nível. Para isso, foi utilizado como base o modelo e a arquitetura do ambiente, previamente definidos. Para isso, foi apresentada uma descrição detalhada desta linguagem, acompanhada de exemplos práticos para uma aplicação direta da mesma.

Primeiramente, pode-se dizer que a linguagem Java mostrou-se apropriada para a tarefa de descrição de cargas de falhas, devido às diversas possibilidades que a mesma proporciona. Dentre elas, podem se destacar o seu uso difundido (diminuindo, assim, a curva de aprendizado do usuário do ambiente), bem como a alta possibilidade de reuso de códigos existentes (facilitando a criação, pelo usuário do ambiente, de cargas de falhas para cenários diferentes, mas que envolvam aspectos semelhantes).

Considerando os requisitos de *flexibilidade* e *expressividade*, pode-se dizer que a linguagem adotada neste trabalho atendeu de forma adequada ambos os requisitos. Neste sentido, Java é **flexível**, por permitir a criação de diferentes modalidades de cargas de falhas com possibilidades de expansões em seu modelo, ao mesmo tempo em que é **expressiva**, devido ao alto nível da linguagem, amplamente conhecida nos dias de hoje.

Além disso, outra característica interessante incluída no trabalho atual diz respeito ao suporte a *topologias*. Graças a este suporte, é possível criar cargas de falhas que realizem a injeção diretamente no componente desejado, de uma forma simples e facilitada. Finalmente, este suporte a topologias também possui a possibilidade de expansões futuras, facilitando a criação de casos de teste ainda não previstos.

Referente ao nível de usabilidade desejado, a abordagem proposta consegue atender, de maneira adequada, aos principais requisitos de usabilidade definidos na literatura desta área (X. Ferré and Constantine, 2001; Nielsen, 1993). Neste sentido, a linguagem Java é de *fácil aprendizado*, como já mencionado anteriormente. Ao mesmo tempo, esta abordagem permite a descrição de cargas de falhas de uma forma *eficiente* e *relativamente livre de erros*. Se considerarmos o público-alvo do ambiente (como desenvolvedores e engenheiros de teste, por exemplo), temos que a abordagem é de *uso agradável*, uma vez que a linguagem Java está muito próxima da realidade deste público.

Ao realizar uma comparação com as ferramentas existentes de injeção de falhas, temos que a abordagem descrita neste trabalho emula um modelo de falhas mais detalhado que os injetores MENDOSUS e FAIL/FCI. Além disso, outra característica presente está relacionada à independência de protocolo e aplicação alvo, o que não ocorre nos injetores DOCTOR (focado em sistemas distribuídos de tempo real) e FIONA (com ênfase no protocolo UDP). Finalmente, a abordagem proposta utiliza uma linguagem de alto nível, facilitando o seu aprendizado, o que é uma dificuldade inerente ao injetor FIRMAMENT, uma vez que o mesmo utiliza-se de uma linguagem de baixo nível para a descrição de cargas de falhas.

APÊNDICE A HISTÓRICO DA PESQUISA

A primeira fonte de inspiração que levou a realização deste trabalho é datada, primeiramente, do ano de 2003, através da participação no *Projeto Simmcast* (Muhammad and Barcellos, 2002) (um framework para avaliação de redes, protocolos e sistemas distribuídos, através da técnica de simulação), como bolsista de iniciação científica. A partir desta participação, o framework Simmcast foi estendido, visando o suporte a experimentos envolvendo injeção de falhas, sendo este trabalho realizado até o ano de 2005 (Barcellos et al., 2005). A arquitetura e um protótipo deste framework estendido, posteriormente denominado como *SimmFI*, foram desenvolvidos como tema de Trabalho de Conclusão, a nível de graduação, no ano de 2006 (Munaretti and Barcellos, 2006).

No contexto do mestrado, o trabalho de pesquisa teve início, de forma efetiva, no primeiro ano do curso de mestrado, durante a execução das disciplinas, durante o ano de 2007. Neste caso, a elaboração do Trabalho Individual (TI) proporcionou um subsídio importante para o início efetivo da pesquisa. Isto se justifica pelo fato de que praticamente todo o levantamento bibliográfico, referente ao estudo das ferramentas de injeção de falhas existentes, foi executado durante a confecção do TI. Por consequência, o capítulo 2 desta dissertação teve o seu conteúdo totalmente baseado no que foi elaborado a partir do Trabalho Individual.

A partir deste levantamento, e já considerando o período do trabalho de pesquisa propriamente dito (ou seja, o primeiro semestre do ano de 2008), foi iniciada a elaboração do *modelo* de um possível ambiente a ser construído. O principal objetivo deste modelo era o de, exatamente, definir uma metodologia que proporcionasse uma utilização efetiva das ferramentas de injeção de falhas, sem a necessidade de aprender as nuances de cada uma destas ferramentas. Este modelo, juntamente com exemplos de utilização do mesmo, foi compilado em um artigo e submetido ao Workshop de Testes e Tolerância a Falhas (WTF) daquele ano. O artigo foi aceito para publicação e apresentação (Munaretti and Weber, 2008a), o que foi realizado juntamente com o Simpósio Brasileiro de Redes de Computadores (SBRC), no Rio de Janeiro.

Com este modelo definido, o foco do trabalho passou a estar voltado para as características internas de cada componente integrante deste modelo. Desta forma, o modelo anteriormente definido foi ampliado, possibilitando uma melhor compreensão dos componentes previamente construídos. Os resultados desta etapa do trabalho foram contemplados em um novo artigo, submetido desta vez para a Escola Regional de Redes de Computadores (ERRC), também em 2008. O artigo submetido foi aceito para publicação e apresentação (Munaretti and Weber, 2008b), sendo realizado na Universidade Ritter dos Reis (UniRitter), em Porto Alegre.

Assim, os dois artigos submetidos até então retrataram praticamente todos os assuntos referentes ao modelo do ambiente proposto. Desta forma, o capítulo 3 da presente disser-

tação inclui tudo o que foi abordado em ambos os artigos. Com este modelo em mente, a arquitetura do ambiente de carga de falhas começou a ser vislumbrada. Juntamente com esta arquitetura, uma versão alfa de um protótipo foi elaborada, visando a especificação de cargas simples de falhas (que eram compostas de, basicamente, apenas *chamadas*, que envolviam a especificação de um ou mais tipos de falhas). Este protótipo, por sua vez, foi denominado inicialmente como *FLE* (um acrônimo de *Fault Load Environment*). O resultado de todo esse trabalho, já no fim do ano de 2008, foi submetido, em forma de artigo, ao Latin-American Test Workshop (LATW), referente ao ano de 2009 (cujo deadline de submissão estava relacionado ao fim do ano de 2008). Este artigo não foi aceito para publicação/apresentação. No entanto, os conteúdos abordados nesta etapa foram primordiais para a execução dos próximos passos do trabalho, bem como para complementar os capítulos referentes ao modelo, arquitetura e protótipo do ambiente, nesta dissertação.

Iniciado o ano de 2009, o trabalho de pesquisa entra em uma nova fase. Isto se justifica pelo fato de que, a partir das observações realizadas pelo professor Sérgio Luis Cechin, de que era realmente necessária uma abordagem para facilitar a especificação de cargas de falhas. Entretanto, foi frisada pelo professor a ideia de que o tempo de aprendizado dispensado nesta nova abordagem proposta deveria ser o menor possível, a fim de que tal abordagem não torne-se complexa, da mesma forma que as abordagens convencionais já o são. Por este motivo, como um primeiro passo, o modelo e a arquitetura do ambiente foram estendidos e reformulados, de forma a tornar o ambiente mais simplificando, bem como prevendo um menor aprendizado na sua utilização. Todo este esforço na continuidade do trabalho foi formatado em um artigo, submetido à Conferência Latino-Americana de Informática (CLEI), referente ao ano de 2009. O artigo foi aceito para publicação (Munaretti et al., 2009b), sendo apresentado na Universidade Federal de Pelotas.

Concomitantemente ao esforço descrito no parágrafo anterior, foi elaborada e definida a metodologia a ser utilizada para a especificação de cargas de falhas, no contexto do ambiente proposto. Para isso, foi adotado um sub-conjunto da linguagem Java, justamente para minimizar o tempo de aprendizado dispensado para uma utilização efetiva do ambiente, como já mencionado anteriormente. A especificação desta nova abordagem para a elaboração de cargas de falhas, juntamente com melhorias no modelo e na arquitetura recém reformulados, foram abordados em um novo artigo, submetido novamente ao Workshop de Testes e Tolerância a Falhas, referente agora à edição de 2009 do respectivo evento. Este artigo foi aceito para submissão e apresentação (Munaretti et al., 2009a), sendo a mesma realizada em João Pessoa, simultamente ao Latin-American Symposium on Dependable Computing (LADC).

Considerando uma arquitetura melhor definida, bem como uma abordagem mais precisa para a especificação de cargas de falhas (sendo a mesma realizada em Java), a próxima etapa do trabalho foi voltada a construção de um protótipo mais elaborado deste ambiente. Para isso, foi necessário um estudo e, posteriormente, a utilização de ferramentas voltadas a geração de compiladores, uma vez que era preciso interpretar uma linguagem de alto nível (neste caso, Java). Neste contexto, foi utilizada a ferramenta *JavaCC*, pela proximidade existente, desta ferramenta, com a linguagem de programação Java. Este protótipo construído foi denominado como *jFaultload*, também pela sua ênfase na utilização de Java.

No estado atual do trabalho, considerando o fim do ano de 2009, foi elaborado um conjunto de experimentos, envolvendo inicialmente o uso do injetor de falhas FIRMAMENT (que é utilizado extensivamente dentro do Grupo de Tolerância a Falhas da UFRGS, bem como em grupos de pesquisa de outras universidades brasileiras). O resultado deste tra-

balho foi submetido em um novo artigo, voltado novamente para o Latin-American Test Workshop (LATW), referente a edição de 2010 do respectivo evento. O artigo submetido foi aceito para apresentação e publicação (Munaretti et al., 2010), sendo a mesma realizada no mês de março de 2010, em Punta del Este (Uruguai).

O último artigo submetido complementou de forma plena o trabalho de pesquisa, se considerarmos a elaboração da dissertação como um todo. Ao visualizarmos os experimentos que envolvem o ambiente, os mesmos foram complementados logo após a submissão do último artigo, com o intuito de considerar o uso de outras ferramentas de injeção de falhas - no contexto do trabalho, foi utilizado o injetor FIRMAMENT para a realização de um experimento prático. Assim, a evolução cronológica da pesquisa realizada, ilustrando de forma esquemática tudo o que foi abordado neste apêndice, pode ser visualizada na tabela A.1.

Tabela A.1: Evolução cronológica da pesquisa realizada

Ano	Eventos
2007	- Elaboração do Trabalho Individual – levantamento de ferramentas
2008	- Construção do modelo – Publicação/apresentação: WTF 2008 - Extensão do modelo – Publicação/apresentação: ERRC 2008 - Construção de arquitetura/protótipo – Submissão ao LATW 2009
2009	- Extensão da arquitetura – Publicação/apresentação: CLEI 2009 - Abordagem para carga de falhas (Java) – Publicação/apresentação: WTF 2009 - Elaboração de protótipo e experimentos – Submissão ao LATW 2010
2010	- Publicação/apresentação: LATW 2010 - Finalização dos experimentos, envolvendo outras ferramentas.

REFERÊNCIAS

Arlat, J., Crouzet, Y., Karlsson, J., Folkesson, P., Fuchs, E., and Leber, G. H. (2003). Comparison of Physical and Software-Implemented Fault Injection Techniques. *IEEE Transactions on Computers*, 52:1115–1133.

Avizienis, A., Laprie, J., Randell, B., and Landwehr, C. (2004). *Basic Concepts and Taxonomy of Dependable and Secure Computing*. In *IEEE Transactions on Dependable and Secure Computing*, volume 1, pages 11–33.

Barcellos, M. P., Woszezenki, C., and Munaretti, R. S. (2005). *Framework de Injeção de Falhas Simulada para Avaliação de Sistemas Distribuídos*. In SBC, editor, *XXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2005)*, pages 799–812, Fortaleza.

Birman, K. (1996a). *Building Secure and Reliable Network Applications*. Prentice Hall. 500 p.

Birman, K. (1996b). *Building Secure and Reliable Network Applications*. Manning Publications, Co, Greenwich.

Chandra, R., Lefever, R., Cukier, M., and Sanders, W. (2000). *Loki: A state-driven fault injector for distributed systems*. In *Proceedings of the International Conference on Dependable Systems and Networks (FTCS-30)*, pages 237–242, New York City, NY, USA.

Cristian, F., Aghili, H., and Strong, R. (1986). *Clock Synchronization in the Presence of Omissions and Performance Faults, and Processor Joins*. In *16th International Symposium on Fault Tolerant Computing Systems*.

Dawson, S., Jahanian, F., and Mitton, T. (1996a). *Fault Injection Experiments on Real-Time Protocols using ORCHESTRA*. In *High-Assurance Systems Engineering Workshop, IEEE Proceedings*, pages 142–149.

Dawson, S., Jahanian, F., Mitton, T., and Tung, T.-L. (1996b). *Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection*. In *Symposium on Fault-Tolerant Computing*, pages 404–414.

Drebes, R. J. (2005). FIRMAMENT: Um Módulo de Injeção de Falhas de Comunicação para Linux. Master's thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre.

Gerchman, J. and Weber, T. S. (2006). *Emulando o Comportamento de TCP/IP em um Ambiente com Falhas para Teste de Aplicações de Rede*. In SBC, editor, *VII Workshop de Testes e Tolerância a Falhas - WTF2006*, pages 41–52, Curitiba, PR.

Han, S., Shin, K., and Rosenberg, H. (1995). *DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems*. In *Int. Computer Performance and Dependability Symposium. (IPDS'95)*, pages 204–213, Erlangen, Germany. IEEE Computer Society Press.

Hoarau, W. and Tixeuil, S. (2005). *A Language-Driven Tool for Fault Injection in Distributed Systems*. In *Proc. of the 6th IEEE/ACM Intl. Workshop on Grid Computing*, pages 194–201, Grand Large, França.

Hsueh, M.-C., Tsai, T. K., and Iyer, R. K. (1997). Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82.

Jacques-Silva, G. (2005). Injeção Distribuída de Falhas para Validação de Dependabilidade de Sistemas Distribuídos de Larga Escala. Master's thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre.

Jacques-Silva, G., Drebes, R., Gerchman, J., and Weber, T. (2004). *FIONA: A Fault Injector for Dependability Evaluation of Java-Based Networks*. In *Proc. of the 3rd IEEE Intl. Symposium on Network Computing and Applications*, pages 303–308, Cambridge, MA.

Jalote, P. (1994). *Fault Tolerance in Distributed Systems*. Prentice-Hall.

Leite, F. O. (2000). ComFIRM: Injeção de Falhas de Comunicação através da Alteração de Recursos do Sistema Operacional. Master's thesis, Universidade Federal do Rio Grande do Sul, Porto Alegre.

Li, X., Martin, R., Nagaraja, K., Nguyen, T. D., and Zhang, B. (2002). *Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services*. In *Proceedings of the 1st Workshop on Novel Uses of System Area Networks (SAN-1)*.

Muhammad, H. H. and Barcellos, M. P. (2002). *Simulating Group Communication Protocols Through an Object-Oriented Framework*. In IEEE, editor, *35th Annual Simulation Symposium*, New York.

Munaretti, R. S. and Barcellos, M. P. (2006). *Construção de uma Ferramenta de Injeção de Falhas Simuladas para Avaliação de Sistemas Distribuídos*. In SBC, editor, *VII Workshop de Testes e Tolerância a Falhas (WTF2006)*, pages 183–194, Curitiba.

Munaretti, R. S., Fiss, B., Weber, T. S., and Cechin, S. L. (2010). *Experimental Dependability Assessment using a Faultload Specification Tool*. In IEEE, editor, *11th Latin-American Test Workshop - LATW*, Punta del Este.

Munaretti, R. S. and Weber, T. S. (2008a). *Modelo de um Ambiente para Descrição de Cenários Detalhados de Falhas*. In SBC, editor, *IX Workshop de Testes e Tolerância a Falhas - WTF2008*, pages 71–84, Rio de Janeiro.

Munaretti, R. S. and Weber, T. S. (2008b). *Suporte a Injetores de Falhas em um Ambiente para Descrição de Cargas de Falhas*. In SBC, editor, *6ª ERRC - Escola Regional de Redes de Computadores*, Porto Alegre.

Munaretti, R. S., Weber, T. S., and Cechin, S. L. (2009a). *Aumentando a Expressividade da Descrição de Cargas de Falhas de Comunicação para Testes com Injetores de Falhas*. In SBC, editor, *X Workshop de Testes e Tolerância a Falhas - WTF2009*, volume 1, pages 115–128, João Pessoa.

Munaretti, R. S., Weber, T. S., and Cechin, S. L. (2009b). *Detailed Description of Communication Faultloads to Improve the Usability of Fault Injection Testing Tools*. In UFPEL, editor, *XXXV Latin American Informatics Conference (CLEI 2009)*, volume 1, pages 1–10, Pelotas.

Nielsen, J. (1993). Iterative User-Interface Design. *IEEE Computer*, pages 32–41.

Shin, K. (1991). *HARTS: A Distributed Real-Time Architecture*. *IEEE Computer*, 24(5):25–35.

Stott, D., Floering, B., Burke, D., Kalbarczyk, Z., and Iyer, R. K. (2000). *NFTAPE: a Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors*. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 91–100.

Tixeuil, S., Hoarau, W., and Silva, L. (2006). *An Overview of Existing Tools for Fault-Injection and Dependability Benchmarking in Grids*. Technical Report TR-0041, CoreGRID (http://www.coregrid.net).

Vacaro, J. C. and Weber, T. S. (2006). *FIRMI: Um Injetor de Falhas para a Avaliação de Aplicações Distribuídas baseadas em RMI*. In SBC, editor, *VII Workshop de Testes e Tolerância a Falhas - WTF2006*, pages 159–170, Curitiba, PR.

Voas, J. M. and McGraw, G. (1998). *Software Fault Injection: Innoculating Programs Against Errors*. Wiley.

X. Ferré, N. Juristo, H. W. and Constantine, L. (2001). Usability Basics for Software Developers. *IEEE Software*, pages 22–29.