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

GABRIELA JACQUES-SILVA

Injeção Distribuída de Falhas para Validação de Dependabilidade de Sistemas Distribuídos de Larga Escala

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

Profa. Dra. Taisy Silva Weber
Orientadora

Porto Alegre, maio de 2005

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Jacques-Silva, Gabriela

Injeção Distribuída de Falhas para Validação de Dependabilidade de Sistemas Distribuídos de Larga Escala / Gabriela Jacques-Silva. – Porto Alegre: PPGC da UFRGS, 2005.

79 f.: il.

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

1. Tolerância a falhas. 2. Injeção de falhas. 3. Avaliação de dependabilidade. 4. Validação experimental. 5. Aplicações Java. I. Weber, Taisy Silva. II. Título.

<“Every machine will eventually fall apart.">
— ANONYMOUS

<“If it’s not on fire, then it’s a software problem.">
— ANONYMOUS

AGRADECIMENTOS

Agradeço primeiramente à minha orientadora Taisy Weber, que deu oportunidade de realizar o mestrado mesmo não sendo dos bancos da graduação da UFRGS. Certamente é um exemplo de dedicação à docência. É uma professora que faz jus ao termo da área que escolheu como trabalho, pois é altamente disponível. Sempre pronta para atender os alunos, sanar dúvidas, dar conselhos e até para contar as últimas novidades dos eventos da computação, tudo sempre com bom humor.

Agradeço também a minha família que me deu suporte emocional e financeiro para concluir esta etapa. Espero que agora, mesmo à distância (seja em Brasília, Primavera do Leste, Santa Maria ou Miami), isso continue. Um agradecimento especial a minha irmã, melhor amiga nem sob encomenda. Infelizmente vais me abandonar mais cedo do que o meu previsto do nosso suposto duplex. Mas obrigada por tudo: desde me aturar filando comida até a compreensão e me apoiar incondicionalmente em todas as situações, por mais estranhas que essas pudessem parecer.

À professora Ingrid Pôrto, que sempre esteve disposta a responder meus questionamentos variados e que agora também é minha orientadora através do projeto DepGriFE. Ao professor Benhur Stein, que foi meu primeiro orientador em atividades de pesquisa e foi fundamental na minha motivação em seguir carreira científica.

Ao CNPq e ao Instituto de Informática, pela concessão da bolsa de mestrado. À HP/Brasil, pela bolsa DTI através do projeto DepGriFE e pelo financiamento da viagem ao NCA.

Aos colegas de laboratório Jeysonn Balbinot, Márcio Bystronski, Pri Kurtz e Felipe Mobus. À Carol Chiao pelo desenvolvimento da interface gráfica do FIONA. Ao Tórgan Siqueira, por sempre deixar os computadores em condições de trabalho e por gerenciar o projeto exemplarmente. Ao Júlio Gerchman, um bolsista de iniciação científica com maturidade de mestrando, que auxiliou sempre com boa vontade nas atividades de implementação, além de questionar e expor novas idéias.

Aos amigos da UFRGS, que garantiram a descontração entre o tempo de trabalho. Entre eles Rodrigo Sanger, Cláris Marquezan, Ricardo Vianna, Guillermo Hess, Dudu Pons Dias, Émerson Barbieri e gurias do Café com Leite Esporte Clube.

Aos meus grandes amigos que fiz aqui e que espero não perder o contato, mesmo que cada um vá para um canto do mundo. Roberto Drebes, perfeccionista ao extremo, pelos debates científicos, auxílio na melhoria da qualidade dos artigos, e por saber me escutar (ou até fingir que escutava e isso já era o suficiente) em vários momentos de estresse e irritação, qualquer fosse o assunto. Michelle Leonhardt, minha ‘miga’ de almoço, lanche ou qualquer intervalo, seja para falarmos de seriados americanos ou reclamarmos das avaliações dos nossos artigos (quanta injustica!). Joana Trindade, a bolsista histriônica que começou tentando me converter para o lado negro da música, mas sempre alegrou o

ambiente de trabalho com suas gargalhadas, falações e cantarolando um repertório não muito variado. Hoje embarca até nos meus programas furados.

Agradecimento especial aos já amigos da UFSM, que me ajudaram bastante desde quando viemos juntos para cá: Lucas Schnorr, Tiago Fioreze, Rodrigo Righi, Hélio Miranda e Daniel Bortolás. Por fim, agradeço também aos meus amigos mais antigos: Kika Martinelli, Raquel Bitencourt e Bruno Menezes.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

BCI	Byte Code Insertion
ComFIRM	Communication Fault Injection through OS Resources Modification
CSFI	Communication Software Fault Injection
DOCTOR	IntegrateD SOftware Fault InjeCtiOn EnviRonment
FIDe	Fault Injection via Debugging
FIONA	Fault Injector Oriented to Network Applications
GOOFI	Generic Object-Oriented Fault Injection
GUI	Graphical User Interface
IP	Internet Protocol
JNI	Java Native Interface
JPDA	Java Platform Debugger Architecture
JVM	Java Virtual Machine
JVMTI	Java Virtual Machine Tool Interface
JVMPI	Java Virtual Machine Profiler Interface
JVMDI	Java Virtual Machine Debug Interface
LWFI	LightWeight Fault Injector
NFTAPE	Network Fault Tolerance and Performance Evaluator
PFI	Protocol Fault Injection
POA	Programação Orientada a Aspectos
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WS-FIT	Web Service - Fault Injection Technology

LISTA DE FIGURAS

RESUMO

Uma etapa fundamental no desenvolvimento de sistemas tolerantes a falhas é a fase de validação, onde é verificado se o sistema está reagindo de maneira correta à ocorrência de falhas. Uma das técnicas usadas para validar experimentalmente um sistema é injeção de falhas. O recente uso de sistemas largamente distribuídos para execução dos mais diversos tipos de aplicações, faz com que novas técnicas para validação de mecanismos de tolerância a falhas sejam desenvolvidas considerando este novo cenário.

Injeção de falhas no sistema de comunicação do nodo é uma técnica tradicional para a validação de aplicações distribuídas, para forçar a ativação dos mecanismos de detecção e recuperação de erros relacionados à troca de mensagens. A condução de experimentos com injetores de comunicação tradicionais é feita pelo uso do injetor em uma máquina do sistema distribuído. Se o cenário desejado é de múltiplas falhas, o injetor deve ser instanciado independentemente nas n máquinas que as falhas serão injetadas. O controle de cada injetor é individual, o que dificulta a realização do experimento. Esta dificuldade aumenta significativamente se o cenário for um sistema distribuído de larga escala. Outro problema a considerar é a ausência de ferramentas apropriadas para a emulação de determinados cenários de falhas. Em aplicações distribuídas de larga escala, um tipo comum de falha é o particionamento de rede. Não há ferramentas que permitam diretamente a validação ou a verificação do processo de defeito de aplicações distribuídas quando ocorre um particionamento de rede.

Este trabalho apresenta o estudo de uma abordagem para injeção de falhas que permita o teste de atributos de dependabilidade de aplicações distribuídas de pequena e larga escala implementadas em Java. A abordagem considera a não obrigatoriedade da alteração do código da aplicação sob teste; a emulação de um cenário de falhas múltiplas que ocorrem em diferentes nodos, permitindo o controle centralizado do experimento; a validação de aplicações que executem em sistemas distribuídos de larga escala e consideram um modelo de falhas realista deste tipo de ambiente, incluindo particionamentos de rede. A viabilidade da abordagem proposta é mostrada através do desenvolvimento do protótipo chamado FIONA (*Fault Injector Oriented to Network Applications*), o qual atualmente injeta falhas em aplicações desenvolvidas sob o protocolo UDP.

Palavras-chave: Tolerância a falhas, injeção de falhas, avaliação de dependabilidade, validação experimental, aplicações Java.

Distributed Fault Injection for Dependability Evaluation of Large-Scale Distributed Systems

ABSTRACT

A fundamental step on the development of fault-tolerant systems is the validation phase, where the system is verified to assure its correct behavior in the occurrence of faults. One of the techniques used to experimentally validate systems is fault injection. The recent use of large-scale distributed systems to execute a great variety of applications requires the development of new validation approaches of the fault tolerance mechanisms considering these new environments.

Fault injection located at a node's communication system is a traditional technique used for the validation of distributed applications, forcing the activation of the error detection and recovery mechanisms related to message exchange. The conduction of experiments with traditional communication fault injectors is done with a single injector in each machine of the distributed system. If the scenario to be tested comprises multiple faults, the injector should be instantiated independently on the various machines where the faults should be injected. Each injector is controlled individually, making the experiment execution more complex. The difficulty increases significantly if the environment is a large-scale distributed system. Another problem to consider is the lack of appropriate tools to emulate some fault scenarios. In large-scale distributed applications, the occurrence of network partitioning faults, for example, are quite common. There are no tools that allow the direct validation or verification of the failure process of distributed applications when a network partitioning occurs.

This work presents the study of an approach for fault injection that allows the test of dependability attributes of small and large-scale distributed applications developed in Java. The approach considers that the alteration of the target application's source code should not be required. Also, it allows emulating a multiple fault scenario occurring in different nodes and the centralized control of the experiment. The approach permits the validation of applications that execute in large-scale distributed systems and considers an adequate fault model for this kind of environment, such as network partitioning. Its viability is shown by the development of a prototype tool named FIONA (*Fault Injector Oriented to Network Applications*), which currently inject faults in applications implemented over UDP.

Keywords: fault tolerance, fault injection, dependability assessment, experimental validation, Java applications.

1 INTRODUÇÃO

Há muitos anos, é evidente a dependência da sociedade quanto ao uso de sistemas de computação. Devido à incorporação destes sistemas no fornecimento dos mais variados serviços, cada sistema possui diferentes requisitos de funcionamento, pois podem estar realizando uma atividade comum, como a visualização de um *site* da internet, ou mesmo uma atividade crítica, como o controle de uma usina de energia nuclear. A ocorrência de uma falha em cada um destes sistemas tem impactos diferentes, pois enquanto o primeiro em seu pior caso implica uma reinicialização do sistema, no segundo o pior caso pode resultar em perdas humanas e monetárias de grandes proporções. Devido a estas diferenças, cada aplicação tem seus requisitos específicos quanto aos tipos de falhas a que deve tolerar. Baseado nestes requisitos, são desenvolvidos mecanismos de tolerância a falhas, os quais são responsáveis pelo tratamento quando da ocorrência de falhas, seja através de técnicas de mascaramento ou de detecção e posterior recuperação.

1.1 Motivação

Uma etapa fundamental no desenvolvimento de sistemas tolerantes a falhas é a fase de validação, onde é verificado se o sistema está reagindo de maneira correta à ocorrência de falhas. Delegar esta verificação para uma situação de uso efetivo do software (uma falha real) pode gerar conseqüências bastante desastrosas. A validação de sistemas pode ser tanto analítica quanto experimental, sendo estas duas formas complementares. Uma das técnicas usadas para validar experimentalmente um sistema é através da injeção de falhas. Através desta técnica, são introduzidas falhas no sistema de maneira controlada e é monitorada a resposta do sistema nessas condições. Seu objetivo é testar a eficiência dos mecanismos de tolerância a falhas e avaliar a segurança de funcionamento dos sistemas, provendo uma realimentação no processo de desenvolvimento (IYER, 1995).

O desenvolvimento de mecanismos de tolerância a falhas é essencial ao projeto de sistemas computacionais de missão crítica. Portanto, há vários anos são conhecidas técnicas tanto para o desenvolvimento como para a validação deste tipo de sistema.

O recente uso de sistemas largamente distribuídos, como *grids* (FOSTER; KESSELMAN; TUECKE, 2001), para execução dos mais diversos tipos aplicações, faz com que novos mecanismos de tolerância a falhas sejam desenvolvidos considerando este novo cenário. Diferentemente de sistemas de missão crítica, esses sistemas são construídos com base em uma infra-estrutura pré-existente e sem previsão inicial de mecanismos de tolerância a falhas. Tais sistemas são bastante usados para a execução de aplicações que demandam alto desempenho. Se não houver nenhuma previsão para o tratamento de falhas, em aplicações de longa duração, por exemplo, a falha de um componente durante sua execução pode comprometer toda a computação. Em casos mais severos, onde a ocor-

rência de falhas é freqüente, pode ser impossível finalizar a computação. Considerando este cenário de sistemas largamente distribuídos, é necessário tanto o desenvolvimento de novas técnicas de tolerância a falhas quanto novos meios para sua validação, o qual é o foco deste trabalho.

Uma das técnicas tradicionais para a validação de aplicações distribuídas é a injeção de falhas no sistema de comunicação do nodo em execução, forçando a ativação dos mecanismos de detecção e recuperação de erros relacionados à troca de mensagens. As falhas são injetadas em um nodo do sistema distribuído e os mecanismos serão ativados nos outros nodos que estavam em comunicação com o nodo em condição de falha. Uma técnica possível é a injeção de falhas na memória ou nos registradores da CPU do nodo, supondo que estes erros irão se propagar até alcançar o sistema de comunicação. Esta é uma abordagem indireta, que pode ter uma latência para a manifestação da falha bastante alta, afetando o tempo de duração do experimento de teste. Para acelerar essa manifestação, as falhas podem ser injetadas diretamente no sistema de troca de mensagens. Ferramentas como DOCTOR (HAN; SHIN; ROSENBERG, 1995), ORCHESTRA (DAWSON; JAHANIAN; MITTON, 1996), CSFI (CARREIRA; MADEIRA; SILVA, 1995) e ComFIRM (LEITE, 2000) usam esta abordagem.

A condução de experimentos com estas ferramentas é dada pelo uso do injetor em uma máquina do sistema distribuído. Se o cenário desejado é de múltiplas falhas, o injetor deve ser instanciado independentemente nas n máquinas nas quais as falhas serão injetadas. O controle de cada injetor é individual, o que dificulta a realização do experimento. Esta dificuldade aumenta significativamente se o cenário for um sistema distribuído de larga escala.

Outro problema a considerar no uso de injeção de falhas, é a falta de ferramentas apropriadas para a emulação de determinados cenários de falhas. Apesar de existirem vários injetores de falhas, ferramentas apropriadas para um alvo específico são geralmente difíceis de encontrar e usar, levando muitos desenvolvedores a construir seus próprios injetores quando necessário. Uma ferramenta de injeção de falhas adequada para uma determinada aplicação deve possibilitar a construção de cenários de falhas de acordo com o modelo de falhas implementado pela aplicação sob teste. Em aplicações distribuídas de larga escala, um tipo comum de falha é o particionamento de rede. Não há ferramentas que permitam diretamente a validação ou a verificação do processo de defeito de aplicações distribuídas quando ocorre um particionamento de rede.

1.2 Objetivo

Considerando o cenário descrito, o presente trabalho tem como objetivo o estudo de uma abordagem para injeção de falhas que permita o teste de atributos de dependabilidade de aplicações distribuídas implementadas em Java. A técnica deve considerar a não obrigatoriedade da alteração do código da aplicação sob teste, tendo uma baixa intrusividade espacial. Para facilitar a portabilidade da ferramenta, recursos do próprio ambiente Java devem ser usados. A arquitetura proposta deve compreender não apenas a injeção de falhas em um nodo do sistema, mas também permitir a emulação de um cenário de falhas múltiplas que se manifestam em diferentes nodos, permitindo o controle centralizado do experimento. A arquitetura também deve possibilitar a validação de aplicações que executem em sistemas distribuídos de larga escala e considerar um modelo de falhas realista deste tipo de ambiente, como particionamentos de rede.

Para demonstrar a factibilidade da abordagem, foi desenvolvido FIONA (*Fault Injec-*

tor Oriented to Network Applications), um injetor distribuído de falhas de comunicação que objetiva validar aplicações distribuídas implementadas em Java. FIONA se diferencia das ferramentas existentes por ser distribuída e escalável, facilitando a configuração de cenários com múltiplas falhas, e por permitir a emulação direta de falhas de particionamento de rede. A ferramenta atualmente opera injetando falhas em aplicações implementadas sob o protocolo UDP. Apesar da ferramenta atualmente estar restrita ao protocolo UDP, a abordagem de injeção de falhas e a arquitetura proposta é mais geral e permite a expansão da ferramenta a outros protocolos de comunicação suportados por Java.

1.3 Resultados Alcançados

O presente trabalho alcançou os seguintes resultados:

- demonstração da possibilidade do uso de uma interface nativa de Java (JVMTI), disponibilizada originalmente para o desenvolvimento de ferramentas de monitoramento e depuração, como abordagem para a técnica de injeção de falhas por software;
- desenvolvimento de uma arquitetura para a implementação de um injetor distribuído de falhas, combinando a arquitetura e os protocolos de ferramentas pré-existentes de monitoramento de sistemas de larga escala;
- emulação direta de um modelo de falhas de sistemas distribuídos de larga escala;
- facilitação da criação de cenários de falhas para sistemas distribuídos, permitindo a configuração centralizada de falhas injetadas em múltiplos nodos;
- desenvolvimento de um protótipo chamado FIONA pronto para uso, que permite a validação de sistemas distribuídos de pequena e larga escala;
- condução de experimentos, que mostram a viabilidade da arquitetura e da abordagem de injeção de falhas proposta, além de evidenciar a operacionalidade do protótipo desenvolvido.

1.4 Organização do Trabalho

Esta dissertação está organizada da seguinte forma:

- O capítulo dois apresenta conceitos de injeção de falhas, falhas de comunicação e modelos de falhas de sistemas distribuídos. São apresentadas também algumas ferramentas existentes de injeção de falhas, sendo estas divididas em duas categorias: ferramentas de injeção de falhas de comunicação e ferramentas distribuídas de injeção de falhas. É abordada ainda a condução de experimentos de injeção de falhas, enfatizando a criação de cenários de falhas e a obtenção de medidas de dependabilidade. O capítulo é finalizado apresentando algumas abordagens conhecidas para teste de injeção de falhas em programas Java, sendo elas: reflexão computacional, programação orientada a aspectos, injetores de baixo nível e uso de interfaces de monitoramento de depuração.

- O capítulo três apresenta a ferramenta FIONA, mostrando seus objetivos e premissas de construção. É apresentado o modelo de falhas considerado, sua arquitetura (local e distribuída) e como seu modelo de falhas pode ser expandido. Considerações sobre a implementação da ferramenta também são abordados, enfatizando o processo de injeção de falhas e as diferenças de implementação da arquitetura local para a distribuída.

- O capítulo quatro apresenta como deve ser feita a construção de cenários de falhas para FIONA e como esta ferramenta deve ser executada, tanto localmente como de forma distribuída. O capítulo ainda apresenta quatro exemplos de testes, utilizando aplicações que usam o protocolo UDP para comunicação. As aplicações foram testadas com falhas de omissão, temporização e particionamento de rede.

- O quinto e último capítulo mostra as contribuições do trabalho e perspectivas futuras.

2 INJEÇÃO DE FALHAS

A utilização de mecanismos de tolerância a falhas é um meio de aumentar a dependabilidade de um sistema, que pode ser definida como a confiabilidade de um sistema computacional. O conceito de dependabilidade inclui vários atributos, sendo eles: confiabilidade, disponibilidade, integridade, *safety*, manutenabilidade e confidencialidade (AVIZIENIS et al., 2004). Para avaliar a dependabilidade de um sistema computacional devemos conhecer seu comportamento não somente em condições normais de operação, como também em condições de falhas. A fim de verificar este comportamento sem esperar que uma falha real ocorra, são realizados experimentos de injeção de falhas. Conhecendo-se o modelo de falhas suportado pelo sistema sob validação, podem ser criadas falhas que são injetadas neste durante sua execução. Observa-se, então, o comportamento do sistema durante a injeção de falhas e os efeitos causados por estas. Através deste tipo de experimento podem ser determinadas medidas de dependabilidade, como a cobertura da detecção de erros, a eficiência e o impacto no desempenho dos mecanismos de tolerância a falhas. Experimentos de injeção de falhas também podem revelar problemas de software que não são encontrados com técnicas tradicionais de teste e métodos formais (VOAS, 1997).

Este capítulo expõe brevemente as técnicas de injeção de falhas existentes, como são classificadas falhas de comunicação, modelos de falhas de sistemas distribuídos, alguns exemplos de ferramentas de injeção de falhas, condução de experimentos de validação, técnicas para injeção de falhas em programas Java e uma conclusão sobre o capítulo.

2.1 Técnicas de Injeção de Falhas

Experimentos de injeção podem ser conduzidos em várias fases do desenvolvimento de um sistema. A técnica a ser utilizada depende da fase que os experimentos serão executados. Na fase de projeto de um sistema podemos usar a técnica de injeção de falhas por *simulação*. Já na fase de protótipo, as técnicas existentes são as de injeção de falhas por *hardware*, por *software* e *híbrida*, que utiliza conjuntamente injeção por hardware e software.

A injeção de falhas por simulação é utilizada logo no início da fase de projeto do sistema. Uma das vantagens do uso de simulação é que ela é capaz de modelar sistemas complexos com um alto grau de fidelidade. Esta técnica de injeção se caracteriza também por ser altamente controlável e flexível em relação ao modelo de falhas, ao tempo/evento ativador da falha e a coleta de dados. Sua desvantagem é que são necessários parâmetros de entrada precisos, além de validação dos resultados de saída (IYER; TANG, 1996). Resultados de experimentos também mostraram que injeção de falhas por simulação pode ser imprecisa na modelagem de comportamentos de sistemas em falha, já que as especificações para os estados falhos são geralmente vagos ou indefinidos (STOTT et al.,

1998). Alguns exemplos de ferramentas de injeção de falhas por simulação são DEPEND (GOSWAMI; IYER; YOUNG, 1997), MEFISTO (JENN et al., 1994) e FOCUS (CHOI; IYER, 1992).

A injeção de falhas por hardware é a técnica que injeta falhas mais próximas daquelas que se manifestam na realidade. A injeção é realizada por um equipamento adicional que pode ter ou não contato com o sistema alvo. Um exemplo de uso do método com contato é o uso de injeção em nível de pinos (ARLAT et al., 1990). Radiação de íons pesados (KARLSSON et al., 1994) e injeção baseada em raio laser (J. R. SAMSON; MORENO; FALQUEZ, 1998) são exemplos de uso do método sem contato. As principais desvantagens do uso de injeção de falhas via hardware é o alto custo de desenvolvimento do injetor, a possível danificação do sistema sob teste e também a baixa portabilidade. No entanto, esta técnica é muito adequada para o estudo de características de dependabilidade de protótipos que exigem alta precisão temporal para ativação do hardware e monitoramento (injetor com baixa intrusividade) ou necessitam acesso a locais que não são facilmente alcançáveis por outros métodos de injeção (HSUEH; TSAI; IYER, 1997).

A terceira técnica existente é a de injeção de falhas por software. Uma ferramenta que utiliza esta técnica geralmente é um trecho de software que usa todos os ganchos (*hooks*) possíveis do processador e do sistema para criar um comportamento incorreto de maneira controlada (CARREIRA; SILVA, 1998). As vantagens desta abordagem são portabilidade, simplicidade de implementação, expansibilidade para novos modelos de falhas, não danificação do sistema sob teste, e injeção de falhas em um nível mais alto de abstração, como o sistema operacional ou então uma aplicação específica. Sua principal desvantagem é a alta intrusividade, que pode ser tanto temporal quanto espacial. Desta maneira, o comportamento do sistema alvo pode ser alterado e, em alguns casos, invalidar os resultados dos experimentos de injeção de falhas. Uma implementação cuidadosa do injetor pode atenuar este tipo de problema. Outro problema dos injetores deste tipo é que alguns recursos de hardware não são acessíveis via software. Com isso surge espaço para o desenvolvimento de uma abordagem híbrida, onde as falhas que não podem ser injetadas por software são injetadas por hardware (KANAWATI; KANAWATI; ABRAHAM, 1995).

Neste trabalho será considerada a técnica de injeção de falhas por software, por ser conveniente para injetar falhas no nível de abstração desejado (sistema de troca de mensagens) e por ter as vantagens descritas acima.

2.2 Falhas de Comunicação

Um sistema distribuído é um conjunto de computadores independentes que parecem ao usuário um sistema único e coerente (TANENBAUM; STEEN, 2002). Para computadores independentes trabalharem conjuntamente, estes trocam informações via troca de mensagens. Quem fornece o serviço de troca de mensagens para as aplicações é o sistema operacional, que usa do meio físico disponível para realizar a transmissão de dados. O sistema operacional oferece chamadas de sistema `send` e `receive`, que são o ponto de entrada para as aplicações realizarem trocas de mensagem. Este ponto de entrada pode ser usado diretamente pelas aplicações ou ainda por uma biblioteca de comunicação que forneça uma abstração de mais alto nível para o programador, como por exemplo a Invocação de Métodos Remotos (*Remote Method Invocation*) de Java (SUN MICROSYSTEMS, 2004a).

Quando se trata de falhas, uma aplicação não consegue distinguir se a falha se manifestou no nível físico, no sistema operacional ou na própria camada de abstração usada

pela aplicação para realizar a troca de mensagens. Portanto, a injeção de falhas de comunicação pode se dar em qualquer um destes níveis.

Para se injetar falhas no sistema de troca de mensagens é necessário que a própria mensagem seja manipulada, podendo ser tratada tanto no envio como na recepção. Com isto, o processo de injeção de falhas de comunicação é dividido em duas fases: a seleção da mensagem e a manipulação da mensagem em si. Esta classificação foi usada primeiramente no desenvolvimento do injetor de falhas ORCHESTRA (DAWSON; JAHANIAN; MITTON, 1996) e influenciou também o desenvolvimento da ferramenta ComFIRM (LEITE, 2000).

A seleção de mensagens pode ocorrer de três formas básicas: (i) *baseada em conteúdo*, (ii) *baseada em fluxo* e (iii) *baseada em elementos externos*. A primeira forma considera que a seleção é baseada no conteúdo da mensagem, ou seja, em algum padrão existente em sua seqüência de *bytes*. Exemplos incluem a seleção de mensagens de confirmação de um determinado protocolo ou de início de uma conexão. A seleção baseada em fluxo não depende do conteúdo individual de uma mensagem, mas apenas do fluxo de mensagens. Podem ser selecionadas aleatoriamente 10% das mensagens transmitidas ou ainda a *n*-ésima mensagem de uma comunicação entre dois processos. O terceiro tipo é aquele em que a seleção não é baseada na mensagem em si, mas em elementos externos, como temporizadores, variáveis controladas pelo usuário ou ainda medições de algum fenômeno físico.

Após a seleção da mensagem, é necessário tomar uma ação sobre ela. A fase de manipulação de mensagens é o momento em que a falha será efetivamente injetada. Existem três classes de ação possíveis: (i) *ação interna*, (ii) *ação sobre a mensagem* e (iii) *ação sobre elementos externos*. A primeira classe inclui todas as ações em que não resultam na impossibilidade de transmissão ou entrega (*delivery*) da mensagem. Neste caso a mensagem tem algum de seus campos alterados, como, por exemplo, o endereço de origem. Ações sobre a própria mensagem são as consideradas mais comuns. Exemplos incluem perda, atraso e duplicação da mensagem. A última classe representa aquelas ações que, embora decorram do fato de alguma mensagem ter sido selecionada, não atuam diretamente sobre a mensagem. Exemplos incluem a atuação sobre contadores, temporizadores e alteração de variáveis.

Para uma ferramenta de injeção de falhas de comunicação ser flexível é indispensável a possibilidade de combinação das condições de seleção e de manipulação de mensagens. Dessa forma, é dada ao usuário a capacidade de construção de cenários de testes bastante variados para a realizar os experimentos na aplicação alvo.

2.3 Modelo de Falhas

Falhas são fenômenos aleatórios e imprevisíveis, que podem levar o sistema a um estado errôneo. Se erros não são tratados, o sistema pode apresentar um defeito. Um defeito se manifesta sempre que um serviço não é prestado de acordo com a sua especificação. Quando um componente falha, este apresenta um comportamento consistente com um modelo de falhas previamente assumido (SCHNEIDER, 1993). Para componentes diferentes, o modelo de falhas considerado também deve ser diferente. A ocorrência de falhas em sistemas distribuídos, o qual é o foco deste trabalho, já foi modelada por vários autores, entre eles Cristian (1991), Schneider (1993) e Birman (1996).

Este trabalho foi baseado no modelo sugerido por Birman (1996), por descrever em seu modelo falhas de particionamento de rede, que são comuns em sistemas distribuídos

de larga escala. Este modelo descreve as seguintes falhas:

- *Parada* ou *colapso* (*halting* ou *crash*) - o componente pára sua execução e entra em colapso sem tomar ações incorretas.
- *Fail-stop* - similares a falhas de colapso, com a diferença de essa falha ser precisamente detectável pelos outros componentes do sistema.
- *Omissão de envio* - manifesta quando a mensagem foi enviada pela aplicação, porém foi perdida antes de esta mensagem efetivamente ser enviada para a rede. Possíveis causas para esse tipo de falha incluem a falta de espaços nos *buffers* do sistema operacional ou mesmo da interface de rede.
- *Omissão de recepção* - similares a falhas de omissão de envio, porém manifestam no lado do destinatário da mensagem, antes desta chegar na aplicação.
- *Rede* - manifesta quando uma mensagem é perdida na própria rede, após seu envio e antes da sua recepção.
- *Particionamento de rede* - forma mais grave de falha na rede, onde a rede se divide em subredes que estão desconectadas entre si. Máquinas dentro de uma mesma subrede conseguem se comunicar, porém todas as mensagem destinadas a máquinas fora desta subrede são perdidas.
- *Temporização* - manifestam quando uma propriedade temporal do sistema é violada, como por exemplo o envio de mensagens fora do tempo especificado (antes ou depois do tempo).
- *Bizantinas* - incluem uma grande variedade de comportamentos falhos, como o corrompimento de dados ou até um comportamento malicioso da aplicação.

Observa-se que para a construção de ferramentas de injeção de falhas, a causa específica de cada falha não interessa ao modelo adotado. É importante apenas que a ferramenta consiga emular as possíveis manifestações de cada falha. No caso de sistemas distribuídos, a maioria das falhas descritas no modelo acima podem ser emuladas afetando o sistema de troca de mensagens. A ação direta do injetor nesse sistema também acelera o disparo dos mecanismos de tolerância a falhas, pois é através dele que as aplicações distribuídas fazem a detecção destes tipos de falhas.

Falhas de colapso, omissão, temporização e bizantinas podem ser emuladas através da injeção de falhas em apenas um nodo, pois são falhas que afetam diretamente as mensagens enviadas por cada nodo. Entretanto, falhas de particionamento de rede, freqüentes em ambientes de redes de larga escala, caracterizam-se por afetarem mais de um nodo por vez. Sua emulação é dificultada quando feita através de um injetor de falhas que opere em um nodo único. Para emular este tipo de falha nestas ferramentas seria necessário gerenciar diversas instâncias independentes da ferramenta em cada nodo afetado pela falha. A emulação destas falhas é facilitada com o uso de uma ferramenta que opere de forma distribuída.

Em sistemas distribuídos, o modelo de falhas considerado também pode variar de acordo com o protocolo de comunicação usado e qual o alvo a ser testado. Uma das características do protocolo UDP é a não confiabilidade, ou seja, não há garantias quanto à recepção da mensagem. Os pacotes podem ser entregues fora de ordem ou até mesmo

perdidos. Este protocolo portanto não tolera falhas como omissão e temporização, sendo necessário o tratamento direto pela aplicação que a usa ou eventualmente por camadas intermediárias. Por esta razão, a injeção deste tipo de falha nestas aplicações se torna necessária a fim de testar o funcionamento correto dos mecanismos desenvolvidos na própria aplicação. Para aplicações que usam comunicação *multicast* (um-para-muitos) o problema é similar, pois também não há garantias de entrega da mensagem. Neste caso há diferença entre a mensagem ser perdida no envio ou na recepção, pois quando a mensagem é perdida no envio nenhum nodo recebe a mensagem, enquanto que se a mensagem é perdida na recepção apenas um ou mais nodos não a recebe.

Diferentemente, aplicações desenvolvidas sob o protocolo TCP não requerem ser testadas em condições de falhas de omissão e temporização, por exemplo. Por este protocolo ter como característica sua confiabilidade, este já trata falhas como perda de pacotes, ordenamento e duplicação de mensagens. Neste caso, ao não ser que o alvo do teste seja o próprio protocolo, o modelo de falhas a ser considerado é mais simplificado. As falhas a serem consideradas são, neste caso, colapso (que pode manifestar em vários momentos da conexão), particionamento de rede e bizantinas. O protocolo TCP, apesar de ser bastante usado, por ser orientado a fluxo (*stream*) não é adequado para construção de sistemas de comunicação de grupo, os quais tendem a ser baseados em mensagens. O TCP também impõe uma semântica FIFO (*First In First Out*), que freqüentemente não é necessária para determinadas aplicações (WIESMANN; DÉFAGO; SCHIPER, 2003).

2.4 Ferramentas de Injeção de Falhas

Um ambiente para realizar testes de dependabilidade via injeção de falhas é geralmente formado por um sistema alvo, um injetor de falhas, uma biblioteca de falhas, um gerador de carga de trabalho (*workload*), uma biblioteca de carga de trabalho, um controlador, um monitor e um coletor e analisador de dados. O funcionamento básico destes ambientes parte da injeção de falhas no sistema alvo com o injetor de falhas. O sistema alvo é alimentado com o gerador de carga de trabalho. A execução do sistema é monitorada pelo monitor que comunica eventos para o coletor de dados. Este coletor rastreia a execução, que pode ser posteriormente analisada pelo analisador de dados. A interação entre os componentes deste ambiente pode ser melhor visualizada na Figura 2.1, proposta por Hsueh, Tsai e Iyer (1997).

Nesta seção são apresentadas ferramentas desenvolvidas em *software* para injeção de falhas de comunicação. Todas essas ferramentas englobam pelo menos parte dos componentes apresentados na figura 2.1. A seção está dividida em ferramentas de injeção local de falhas e ferramentas que utilizam uma abordagem distribuída, tal qual FIONA, o injetor apresentado neste trabalho.

2.4.1 Injetores de Falhas de Comunicação

Nesta subseção são apresentados brevemente alguns dos injetores de falhas de comunicação existentes. São mostradas também ferramentas que, por possuírem uma arquitetura modular, permitem a extensão de seu modelo de falhas. Os aspectos enfatizados são: o método de injeção de falhas, os tipos de falhas que podem ser injetadas e o sistema alvo para qual foram desenvolvidas.

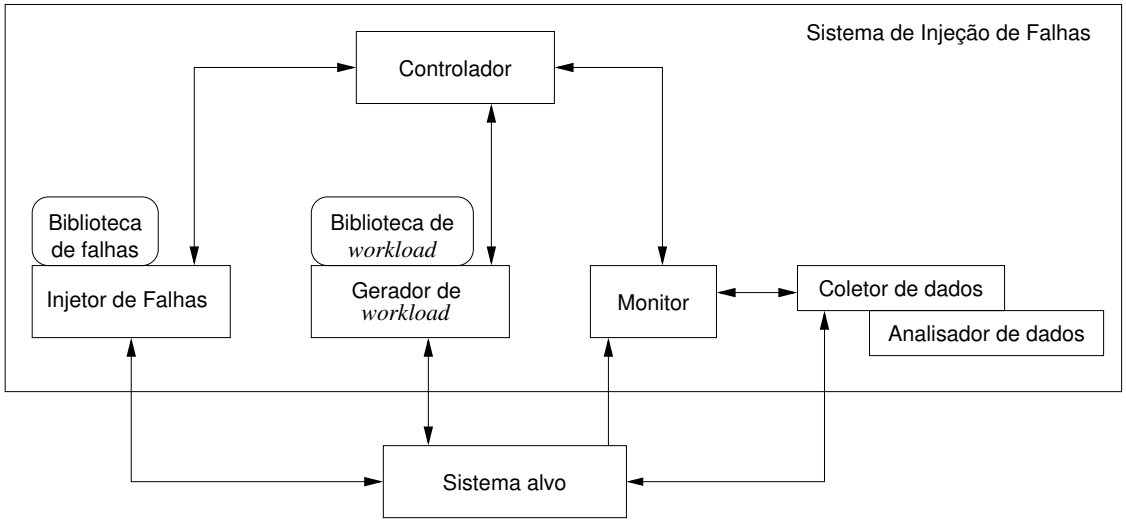

Figura 2.1: Componentes básicos de um ambiente de injeção de falhas

ORCHESTRA

ORCHESTRA (DAWSON; JAHANIAN; MITTON, 1996) é um ambiente de injeção de falhas desenvolvido na Universidade de Michigan para teste de dependabilidade de protocolos distribuídos. Não há distinção entre protocolos de aplicação, protocolos de comunicação ou até mesmo protocolos de dispositivos. O mecanismo de injeção de falhas é através da inserção de uma camada na pilha de protocolos, chamada de PFI (*Protocol Fault Injection* - Injeção de Falhas de Protocolo). Ela é inserida entre duas camadas da pilha de protocolos, sendo logo abaixo do protocolo sob teste e logo acima da camada que o protocolo sob teste trabalha. O PFI tem a função de filtrar e manipular as mensagens trocadas entre os participantes. A PFI é implementada como uma camada de protocolo *x-kernel* (HUTCHINSON; PETERSON, 1991), que é um *framework* para implementação de protocolos que pode ser embutido em qualquer sistema operacional. Entretanto, o *x-kernel* foi implementado inicialmente para o *microkernel* do sistema operacional Mach. Mais tarde houve implementações para o Linux e Solaris, porém estas versões já tiveram seu desenvolvimento descontinuado. As operações possíveis sobre mensagens incluem atraso, perda, reordenação, duplicação, modificação de mensagens e inserção de mensagens espúrias.

DOCTOR

Diferentemente de ORCHESTRA, DOCTOR (*Integrated Software Fault Injection Environment*) (HAN; SHIN; ROSENBERG, 1995) pode não só injetar falhas de comunicação, mas também falhas de memória e de processador. A ferramenta tem como principal alvo o teste de sistemas distribuídos. O disparo para o início da injeção de falhas no sistema pode acontecer de três formas: via *time-out*, interrupções ou então modificação de código. As opções para injeção de falhas de comunicação são a perda de mensagens, duplicação, alteração, atraso ou ainda alguma definida pelo usuário. As falhas definidas pelo usuário podem ser combinações de falhas pré-definidas, podendo ser baseadas no conteúdo ou no histórico das mensagens. Esta ferramenta foi desenvolvida na Universidade de Michigan e foi usada para verificar o comportamento de Harts, um sistema distribuído de tempo real, quando mensagens eram perdidas.

CSFI

CSFI (*Communication Software Fault Injector*) (CARREIRA; MADEIRA; SILVA, 1995) é uma ferramenta implementada na Universidade de Coimbra para realizar injeção de falhas de comunicação. Seu objetivo é avaliar o impacto de falhas em sistemas paralelos. CSFI funciona baseado na inserção de um porção de *software* na camada de comunicação de um determinado nodo. Dessa forma, o fluxo de pacotes é controlado e o conteúdo destes é corrompido de acordo com a demanda. Esta adulteração pode afetar diferentes tipos de pacotes e diferentes partes dentro de um pacote. A duração das falhas podem variar entre transientes e permanentes. A versão existente de CSFI foi implementada para um sistema *transputer* T805. Porém, devido a sua estrutura modular, CSFI pode ser portado para outros sistemas com pequeno esforço, segundo os autores.

WS-FIT

WS-FIT (*Web Service - Fault Injection Technology*) (LOOKER; XU, 2003) é um injetor de falhas de comunicação para teste de *web services* baseados no protocolo SOAP (*Simple Object Access Protocol*). A injeção de falhas acontece através da inserção de ganchos na implementação da API (*Application Programming Interface*) SOAP em uso pela aplicação. Neste caso, quando uma mensagem SOAP é enviada, esta é redirecionada para o injetor de falhas, que faz o tratamento da mensagem (dependendo se uma falha deve ser injetada ou não) e a devolve a mensagem a API SOAP, que posteriormente envia a mensagem para o seu destino correto. O modelo de falhas considerado por WS-FIT inclui deleção, duplicação, reordenação e corrompimento de mensagens.

ComFIRM

A ferramenta ComFIRM (*Communication Fault Injection through OS Resources Modification*) (LEITE, 2000), desenvolvida no Instituto de Informática da Universidade Federal do Rio Grande do Sul, propõe-se a injetar falhas apenas de comunicação. ComFIRM se situa no núcleo do sistema operacional Linux, no nível mais baixo do tratamento de mensagens pelo subsistema de rede. Em sua primeira versão, o código da ferramenta era inserido diretamente no código do núcleo do sistema operacional. Atualmente, ComFIRM é instalado no sistema através de um módulo, dispensando a recompilação do núcleo do sistema. A falha é injetada quando uma função do sistema operacional relacionada ao sistema de comunicação é chamada. Desta forma, uma função de tratamento associada com essa atividade realiza as atividades de injeção de falhas. Assim, a ferramenta decide, baseada em regras de configuração, se a mensagem será adulterada ou não. ComFIRM permite modificar o conteúdo das mensagens ou simplesmente descartá-las.

GOOFI

GOOFI (*Generic Object-Oriented Fault Injection*) (AIDEMARK et al., 2001) é uma ferramenta recentemente desenvolvida na Universidade de Tecnologia de Chalmers. Esta ferramenta é genérica, já que não é presa a nenhuma técnica específica de injeção de falhas. Desta maneira, segundo os autores, é construído um ambiente de fácil adaptação para injeção de falhas necessárias para um determinado sistema alvo. Também segundo os autores, esta ferramenta é altamente portável por ser desenvolvida em Java e usar um banco de dados compatível com a linguagem SQL. Na prática não foi possível obter a ferramenta para avaliação ou extensão. A arquitetura desta ferramenta pode ser vista na figura 2.2.

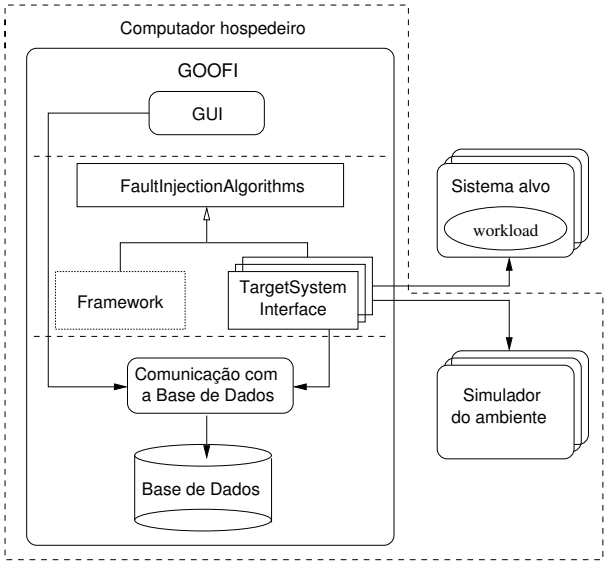

Figura 2.2: Arquitetura da ferramenta GOOFI

A arquitetura da ferramenta pode ser divida em três camadas. A GUI (*Graphical User Interface*) é usada para configurar e iniciar os experimentos de injeção de falhas. A camada central inclui as classes Java `FaultInjectionAlgorithms`, `Framework`, `TargetSystemInterface`. Estas classes definem métodos abstratos que devem ser especificados de acordo com o algoritmo de injeção de falhas e o sistema alvo do experimento. A última camada da arquitetura é a interface com o banco de dados. Esta é usada para armazenar informações sobre o sistema alvo, os experimentos de injeção de falhas e os dados capturados de cada experimento.

NFTAPE

A ferramenta NFTAPE (*Network Fault Tolerance and Performance Evaluator*) (STOTT et al., 2000), desenvolvida na Universidade de Illinois, destaca-se por ser uma ferramenta com múltiplos modelos de falhas, diversos modos de disparo para injeção de falhas e também diversos sistemas alvo. Para permitir essa multiplicidade, NFTAPE inova criando o conceito de Injetor "Leve" de Falhas (*LightWeight Fault Injector* – LWFI). Para desenvolver um novo injetor de falhas, apenas um novo LWFI precisa ser desenvolvido, o que facilita sua expansibilidade para novos modelos de falhas. Os modelos incorporados atualmente incluem injeção de falhas em registradores, memória, erros de comunicação e falhas de entrada e saída. Outro aspecto importante de NFTAPE é a possibilidade de realizar não só injeção de falhas por software, mas também injeção de falhas baseada em hardware e baseada em simulação.

FIDe

FIDe (*Fault Injection via Debugging*) (GONÇALVES, 2002) é um injetor de falhas desenvolvido no Instituto de Informática da Universidade Federal do Rio Grande do Sul, e seu princípio de injeção é o uso de recursos de depuração do sistema operacional Linux. Para injetar falhas, o próprio FIDe executa a aplicação alvo em modo de depuração através da chamada de sistema `ptrace()`. Com o uso desta chamada, a aplicação é interceptada por FIDe toda vez que uma chamada de sistema ocorre. Neste momento, o

injetor decide, baseado em arquivos de configuração, se uma determinada falha deve ser injetada ou não. A extensibilidade de FIDe se diferencia de outras ferramentas, como GOOFI e NFTAPE, já que sua capacidade de expansão está diretamente ligada a flexibilidade da chamada `ptrace()`. O modelo de falhas desta ferramenta é tão amplo quanto as chamadas de sistemas que são interceptadas por `ptrace()`. FIDe foi usado para a condução de experimentos de injeção de falhas em bancos de dados (GONÇALVES et al., 2001), porém esta ferramenta teve seu desenvolvimento descontinuado por implicar em uma alta intrusividade temporal na aplicação sob teste.

Jaca

Jaca (MARTINS; RUBIRA; LEME, 2002) é uma ferramenta de injeção de falhas baseada em reflexão computacional em desenvolvimento na Universidade Estadual de Campinas. Sua principal motivação é o auxílio na validação de aplicações desenvolvidas no paradigma de orientação a objetos, na linguagem de programação Java. As principais características de Jaca são: reusabilidade, flexibilidade, portabilidade e extensibilidade. As falhas que podem ser injetadas com esta ferramenta incluem o corrompimento de atributos, de parâmetros e também de valores de retornos de métodos. Uma particularidade de Jaca é ser baseada em um padrão de software desenvolvido para construção de ferramentas de injeção de falhas (LEME, 2001; LEME; MARTINS; RUBIRA, 2001). Este padrão facilita a expansão do modelo de falhas da ferramenta, a qual foi expandida para injeção de falhas de comunicação no protocolo UDP (*User Datagram Protocol*) (JACQUES-SILVA et al., 2004). No entanto, esta implementação tem uma alta intrusividade espacial, pois obriga que a aplicação alvo tenha suas classes de comunicação instrumentadas para permitir o teste de injeção de falhas.

2.4.2 Injetores Distribuídos de Falhas

Nesta subseção são apresentados dois injetores distribuídos de falhas conhecidos. Como se pode ver, esta não é uma abordagem muito explorada. A primeira iniciativa de desenvolver um injetor deste tipo é de 1994, na Universidade de Illinois, no grupo liderado pelo Professor Ravishankar Iyer. A segunda iniciativa iniciou em 1998, também na Universidade de Illinois, porém no grupo de pesquisa do Professor William Sanders. Ambas ferramentas são projetadas para injeção de falhas em sistemas distribuídos de pequena escala.

DEFINE

DEFINE (KAO; IYER, 1994) é um ambiente distribuído de injeção de falhas e monitoramento para avaliar a dependabilidade de um sistema, investigar a propagação das falhas e validar mecanismos de tolerância a falhas. As falhas injetadas por DEFINE podem afetar qualquer processo em execução no sistema distribuído. Por ser distribuída, a ferramenta permite também a injeção de múltiplas falhas em máquinas diferentes simultaneamente. O modelo de falhas considerado inclui falhas de memória, CPU, barramento e também falhas de comunicação. Neste último caso, o injetor possibilita a omissão ou o corrompimento de mensagens dos protocolos UDP e TCP (*Transmission Control Protocol*).

Loki

O injetor de falhas Loki (CUKIER et al., 1999; CHANDRA et al., 2004) além de se diferenciar por ser distribuído, tem como característica o uso do estado global da aplicação sob teste para definir o momento do disparo de uma falha. Isto possibilita o estabelecimento de pontos de injeção de falhas bastante precisos na execução do sistema, permitindo a observação de seus efeitos em detalhe. Para configurar um experimento de injeção de falhas, o usuário deve especificar em qual estado cada máquina do sistema distribuído deve estar (estado global) no momento da injeção da falha. Apesar deste método possuir uma grande flexibilidade, em algumas situações a configuração de um experimento pode se tornar difícil.

FIONA

A ferramenta FIONA, apresentada neste trabalho, tem como alvo a validação de sistemas distribuídos de pequena e larga escala desenvolvidos em Java. Diferentemente das ferramentas DEFINE e Loki, FIONA possui uma arquitetura distribuída escalável, baseada em arquiteturas de sistemas de monitoramento. Outra característica de FIONA é não ser necessária a alteração do código da aplicação sob teste e permitir a emulação de cenários de múltiplas falhas. Para facilitar sua configuração, esta permite a criação de cenários de falhas de forma centralizada, evitando que o usuário tenha que distribuir manualmente as configurações. Para monitoramento, a ferramenta recolhe automaticamente os *logs* de injeção de falhas gerados durante a execução do experimento e forma um *log* único, com uma visão global do experimento. FIONA tem sua arquitetura e implementação descrita detalhadamente no capítulo 3.

2.5 Condução de Experimentos de Injeção de Falhas

A condução de um experimento de injeção de falhas passa por diferentes fases. Primeiramente, é necessário a definição de um cenário de falhas ao qual a aplicação será submetida durante sua execução. Isto inclui a definição de localização, tipo, duração e tempo de ativação das falhas a serem injetadas. A seguir, a aplicação é executada em conjunto com o injetor de falhas, que irá injetar as falhas descritas no cenário previamente especificado. O teste deve ser executado repetidas vezes, além de ser executado com diferentes cargas de trabalho e sob diferentes condições de falha. Durante esta fase são coletados dados do experimento, sendo monitorado e observado como o sistema se comporta no ambiente faltoso. Como etapa final, os dados coletados são analisados para a obtenção de medidas de dependabilidade da aplicação.

As subseções a seguir apresentam como deve ser definido um cenário de falhas, o uso de experimentos de injeção de falhas para depuração de programas e para a obtenção de medidas de dependabilidade.

2.5.1 Cenários de Falhas

Para estabelecer o cenário de falhas de determinado sistema alvo, primeiro é necessário identificar quais os mecanismos de tolerância a falhas implementados pela aplicação. Conhecendo os mecanismos, sabemos quais falhas este sistema está preparado para detectar e, se possível, recuperar. A criação de um cenário de falhas para um experimento deve ser baseado neste mesmo modelo de falhas projetado pelo desenvolvedor da aplicação. Por exemplo, em uma aplicação distribuída projetada para tolerar omissão de mensagens,

falhas de omissão devem ser incluídas durante o experimento de injeção de falhas. Um modelo de falhas mais amplo pode ser usado no teste, entretanto apenas para observar o processo de defeito da aplicação. Provavelmente o sistema ficará em defeito quando sujeito a falhas que este não está preparado para tolerar. A injeção destes tipos de falhas permite antecipar e avaliar o comportamento da aplicação, o qual sem injeção de falhas, seria naturalmente difícil de conseguir.

Depois de decididos quais os tipos de falhas devem ser injetadas, é necessário definir também o momento de ativação da falha, sua duração e comportamento. Quanto à ativação, essa deve ser variada para ocorrer em diversos momentos da execução, pois falhas são fenômenos imprevisíveis e podem se manifestar a qualquer momento. A definição da duração de uma falha e seu comportamento, devem ser baseados em como essas falhas geralmente se manifestam na realidade. Por exemplo, no caso de falhas de omissão, as perdas podem ocorrer em rajadas com uma alta taxa de falhas das mensagens enviadas, ou então perdas constantes na rede, porém com uma baixa porcentagem.

Para realizar um amplo e eficiente teste dos mecanismos de tolerância a falhas é importante a criação de cenários de falhas variados, para que a aplicação seja testada em diversas situações. Quanto mais situações forem testadas e verificadas, mais estará sendo assegurando o funcionamento correto dos mecanismos desenvolvidos e mais precisa será a medida de dependabilidade obtida. Entretanto, ressalta-se que a criação de cenários deve se ater àqueles os quais podem ocorrer na realidade, pois senão os resultados reais do teste podem estar sendo mascarados.

2.5.2 Teste e Depuração

Como citado no capítulo 1, a técnica de injeção de falhas é usada não apenas como um meio para obtenção de medidas de dependabilidade, mas também como uma técnica útil para teste e depuração da aplicação, provendo uma realimentação no processo de desenvolvimento. Quando se desenvolve um mecanismo de tolerância a falhas em uma aplicação, antes de colocá-lo em produção e também antes de submetê-lo a uma carga de testes para obtenção de medidas, este deve ter seu funcionamento testado. Para que não seja necessário esperar a ocorrência natural de uma falha, são conduzidos experimentos de injeção de falhas, que aceleram a ocorrência destas falhas através da emulação dos seus efeitos para a aplicação sob teste. Desta forma é possível verificar se o mecanismo desenvolvido está funcionando de acordo com o especificado quando a falha prevista pela aplicação é injetada.

2.5.3 Monitoramento e Medidas de Dependabilidade

Para que se tenha sucesso na obtenção de medidas de dependabilidade, a ferramenta deve ser eficiente tanto no momento de injetar as falhas quanto ao reportar os resultados dos testes e o comportamento do sistema ao tratar as falhas injetadas. Geralmente, o comportamento da aplicação alvo não é facilmente observado pela pessoa que está conduzindo os experimentos. Por esta razão e por questões de automatização dos testes, a aplicação alvo deve ser instrumentada para monitorar todos os eventos relacionados à detecção de erros e à recuperação de situações de falha.

Este monitoramento é a base para a obtenção de medidas de dependabilidade do sistema sob teste. As medidas podem ser calculadas cruzando os dados fornecidos pela ferramenta de injeção de falhas (tempo, local e tipo da falha injetada), o arquivo de *log* relativo aos mecanismos de tolerância a falhas da própria aplicação e eventualmente dados do próprio sistema operacional relacionados a falhas ocorridas na máquina.

Um exemplo de medida que se pode obter é a cobertura de erros, que pode ser determinada através da comparação do *log* das falhas detectadas e das falhas efetivamente injetadas. O cálculo pode ser feito computando a porcentagem de falhas que ativaram o mecanismo de tolerância a falhas da aplicação alvo. Se a cobertura não for considerada satisfatória, os mecanismos podem ser refinados. Esta operação pode ser repetida até o desenvolvedor considerar suficiente a medida alcançada.

Outra possível medida é a queda de desempenho da aplicação causada por falhas. Considerando aplicações distribuídas, por exemplo, um modo de obtenção dessa medida pode ser através do tempo de resposta para o cliente. O experimento deve ser executado sem injeção de falhas e com injeção de falhas. A diferença indica a queda de desempenho, a qual o desenvolvedor deve avaliar se é uma medida aceitável ou não para a aplicação em teste. Outra forma direta de determinar a queda de desempenho é através da determinação da vazão do sistema (*throughput*), ou seja, a quantia de trabalho que o sistema processa durante um período do tempo. Neste caso, a medida também é avaliada em uma execução livre de falhas e outra com injeção de falhas.

2.5.4 *Benchmarks* de Dependabilidade

O interesse pela definição e obtenção de medidas de dependabilidade é grande, o que vem motivando pesquisas para o desenvolvimento de *benchmarks* de dependabilidade. O objetivo do desenvolvimento de *benchmarks* em geral é que estes possibilitam uma comparação justa entre sistemas, medindo a qualidade de serviço na perspectiva do usuário dada uma determinada carga de trabalho (OPPENHEIMER et al., 2002). No caso de *benchmarks* de dependabilidade, estes têm como finalidade oferecer meios genéricos para caracterizar o comportamento de componentes e sistemas computacionais em presença de falhas. A diferença principal entre *benchmarks* e as técnicas tradicionais de validação é que os *benchmarks* devem usar uma representação baseada em um consenso aceito pela indústrias e a comunidade de usuários (KANOUN; MADEIRA; ARLAT, 2002). Enquanto há muito trabalho a ser feito até a existência de um *benchmark* de dependabilidade e sua aceitação pela indústria e comunidade, técnicas de injeção de falhas são consideradas uma tecnologia importante, se não crucial, para alcançar este desafio (SIEWIOREK; CHILLAREGE; KALBARCZYK, 2004).

2.6 Implementação de Injetores de Falhas por Software

Como descrito na seção 2.1, a técnica de injeção de falhas por software apresenta determinadas vantagens, como a portabilidade e a não danificação de sistemas sob teste, e desvantagens, como a baixa resolução de tempo e a não possibilidade de injeção de falhas a locais inalcançáveis por software. Segundo Hsueh, Tsai e Iyer (1997), para implementar uma ferramenta deste tipo existem dois métodos básicos para injeção de falhas: em tempo de compilação e durante a execução do programa.

Para injetar falhas em tempo de compilação, o código fonte da aplicação sob teste é diretamente alterado. Desta forma, a própria aplicação contém regiões do código onde são emulados os efeitos das falhas consideradas na implementação da aplicação. A realização do experimento de validação, neste caso, se dá simplesmente pela execução do programa gerado na compilação do código fonte alterado. Desta forma, as falhas são ativadas quando a região alterada do código é executada. Esta técnica é de fácil implementação, pois o código a ser adicionado é específico para a aplicação sob teste. Sua principal desvantagem é que somente o programa alterado pode ser testado, portanto sua

reusabilidade é baixa.

Diferentemente do método de injeção de falhas por compilação, o método de injeção durante a execução do programa depende de mecanismos de disparo para ativar as falhas. Os mais comuns são:

- *Time-out* - a ativação da falha acontece quando um temporizador expira seu tempo pré-programado. O efeito de uma falha ativada por tempo é imprevisível, pois não é possível saber em que parte da execução o programa estará no momento da ativação.

- *Exceções/Interrupções* - neste caso, a ativação de falhas ocorre devido ao acontecimento de um determinado evento, podendo ser uma interrupção de hardware ou de software. Neste momento, o controle é passado para uma função de tratamento do injetor de falhas, que decide a ação a ser tomada.

- *Inserção de Código* - este mecanismo adiciona código que permite a ocorrência de injeção de falhas antes de determinadas instruções.

A ferramenta FIONA, apresentada neste trabalho, é implementada com uma técnica mista, baseando-se em princípios da técnica de injeção por falhas em tempo de compilação e dinâmica. FIONA adiciona código de injeção de falhas em dois momentos: (i) em tempo de compilação, quando altera o código do protocolo alvo e recompila a nova classe gerada, e (ii) alteração dinâmica por interrupção e inserção de código, no momento em que depende do acontecimento do evento de carregamento de classe para substituir o código original pelo código da classe alterada e pré-compilada.

2.7 Intrusividade Temporal e Espacial

O conceito de intrusividade, também conhecido como perturbação, é usado não só na área de injeção de falhas, mas como também em outras da computação, como em monitoramento de programas. Para se injetar falha em uma aplicação adiciona-se um código que emula o comportamento de uma falha. Para esta emulação, a aplicação executa um código extra, desviando seu fluxo da execução natural. Este fluxo adicional implica dois tipos de penalidade, sendo estas temporal e espacial.

A penalidade temporal acontece por ser necessária a execução de uma porção de código extra para permitir a emulação das falhas, o que gera um maior tempo de execução, implicando uma queda no desempenho. Quando a aplicação alvo é de tempo real, o injetor de falhas utilizado deve ser construído de forma a não alterar as limitações temporais requisitadas pela aplicação. A geração de instrusividade temporal é intrínsica aos injetores de falhas por software, sendo possível apenas minimizar seus efeitos.

A intrusividade espacial é relacionada à modificação de código. Para injeção de falhas por software, o código de emulação pode ser posicionado em qualquer gancho existente no sistema, seja diretamente na aplicação sob teste ou em níveis mais baixos, como máquina virtual, protocolo em uso ou no sistema operacional. A decisão da localização do código de injeção é um compromisso entre generalidade e portabilidade, pois quanto mais próximo à aplicação, menos genérico o injetor, porém quanto mais próximo ao sistema operacional, menos portável é a ferramenta. Para o testador do software, o mais importante é a ferramenta não requisitar a alteração do código da própria aplicação sob teste, pois a inserção de código pode implicar a criação de novos erros.

2.8 Injeção de Falhas em Programas Java

Existem diferentes abordagens para a injeção de falhas em aplicações Java. Entre elas podemos destacar: *reflexão computacional*, *programação orientada a aspecto*, uso de *injetores de falha de baixo nível* (abaixo da máquina virtual) e uso de *interfaces de monitoramento e depuração*. A seguir são apresentadas as vantagens e desvantagens do uso de cada abordagem, bem como exemplos de ferramentas que utilizam cada uma delas.

2.8.1 Reflexão Computacional

Reflexão computacional (MAES, 1987) tem sido recentemente explorada como abordagem para injeção de falhas. Exemplos de ferramentas que usam essa abordagem são FIRE (MARTINS; ROSA, 2000) e Jaca (MARTINS; RUBIRA; LEME, 2002). A primeira injeta falhas em programas C++, enquanto que a segunda em aplicações Java. A base para a construção de sistemas reflexivos a sua divisão em dois níveis: o *nível base* e o *meta-nível*. No nível base se encontra a implementação da aplicação. O meta-nível é onde as estruturas do nível base podem ser observadas ou ter seu comportamento modificado. Uma possível implementação destes conceitos a linguagens orientadas a objeto é o uso de meta-objetos. Em Jaca, estes meta-objetos são associados aos objetos de nível base que terão seu comportamento alterado, ou seja, terão falhas sendo injetadas.

As principais vantagens do uso de reflexão computacional são: (i) uso do meta-nível para a implementação das rotinas de injeção de falhas, aumentando a modularidade e a reusabilidade; (ii) a separação do nível-base do meta-nível, resultando em independência entre o processo de injeção de falhas e a aplicação em si; (iii) menor dependência a uma plataforma específica, permitindo a execução do injetor de falhas em qualquer plataforma que execute programas desenvolvidos na linguagem escolhida.

Uma desvantagem do uso de reflexão computacional aparece quando classes de sistema devem ser interceptadas para as atividades de injeção de falhas, como usado na extensão de Jaca para falhas de comunicação (JACQUES-SILVA et al., 2004). Usualmente *toolkits* de reflexão, como Javassist (CHIBA, 2000), não permitem a reflexão de classes de sistema, devido ao modelo de segurança de Java. Em algumas situações, isso implica modificar o código da aplicação, o que aumenta a intrusividade espacial do injetor de falhas.

2.8.2 Programação Orientada a Aspectos

Programação Orientada a Aspectos (POA) (KICKZALES et al., 1997) vem sendo estudada como uma nova abordagem para injeção de falhas, já que este paradigma permite a interceptação e a alteração de métodos durante sua execução, sem modificar o código da aplicação sob teste. Quando POA é mapeada para os conceitos de injeção de falhas, pode-se determinar que a ferramenta de injeção de falhas é um *crosscutting concern* da aplicação e que cada tipo de falha pode ser visto como um *aspecto*. Está sendo implementado o injetor de falhas baseado em aspectos FITA (SILVEIRA; WEBER, 2004), desenvolvido em Java e em AspectJ. Esta abordagem já se mostrou factível, porém ainda não se tem elementos suficientes para determinar com precisão suas vantagens e desvantagens.

2.8.3 Nível Inferior a Máquina Virtual

Injetores de nível inferior a máquina virtual usam quaisquer ganchos existentes abaixo do nível da aplicação sob teste, podendo ser os providos pelo sistema operacional, bibliotecas de sistema ou até mesmo de hardware. Um exemplo de ferramenta é ComFIRM

(LEITE, 2000; DREBES et al., 2005), que funciona dentro do núcleo do Linux. ComFIRM inspeciona todos os pacotes transmitidos ou recebidos por um nodo e toma ações baseado em regras de seleção e manipulação. Uma das principais vantagens dessa abordagem é de que falhas podem ser injetadas em qualquer aplicação que execute na plataforma de suporte, sendo independente da linguagem usada para desenvolvimento.

Entretanto, o injetor é específico para uma plataforma, comprometendo sua portabilidade. Além disso, o código de injeção de falhas é executado mesmo quando a aplicação sob teste não está executando. Como a ferramenta está em um nível mais baixo, todas as aplicações que estiverem acima deste nível terão seus pacotes inspecionados, o que implica uma maior sobrecarga no sistema como um todo. Para minimizar este efeito, as regras de configuração do experimento devem ser montadas com cuidado, para perturbar o menos possível o sistema. Dependendo do nível em que o injetor é implementado, pode ser necessário acesso privilegiado ao sistema. Os riscos de um usuário comum ter acesso completo ao sistema é, em alguns casos, inaceitável mesmo em ambientes de teste.

2.8.4 Interface de Monitoramento e Depuração

A última abordagem aqui descrita para injeção de falhas em programas Java é a explorada por FIONA, que é o uso de uma interface nativa para o desenvolvimento de ferramentas de monitoramento e depuração. A arquitetura de depuração da plataforma Java (*Java Platform Debugger Architecture - JPDA*) foi recentemente atualizada em sua versão 1.5 (Java 5). Foi incluída uma nova interface nativa que suporta o desenvolvimento de ferramentas de depuração para programas Java. Essa interface, chamada JVMTI (*Java Virtual Machine Tool Interface*) (SUN MICROSYSTEMS, 2004b), substitui as antigas interfaces: JVMPI (*Java Virtual Machine Profiler Interface*) (SUN MICROSYSTEMS, 2002; VISWANATHAN; LIANG, 2000) e JVMDI (*Java Virtual Machine Debug Interface*) (SUN MICROSYSTEMS, 2004c), que foram descontinuadas nesta nova versão da plataforma.

JVMTI provê meios de inspecionar o estado e controlar a execução de aplicações rodando na máquina virtual Java. O cliente JVMTI é chamado de *agente* e é implementado como uma biblioteca dinâmica que é notificada através de eventos, como o carregamento de uma classe ou a inicialização de uma *thread*. Quando notificado, o agente pode chamar funções que podem alterar ou inspecionar o estado da máquina virtual. A figura 2.3 representa a arquitetura geral de uma ferramenta de depuração baseada em JVMTI.

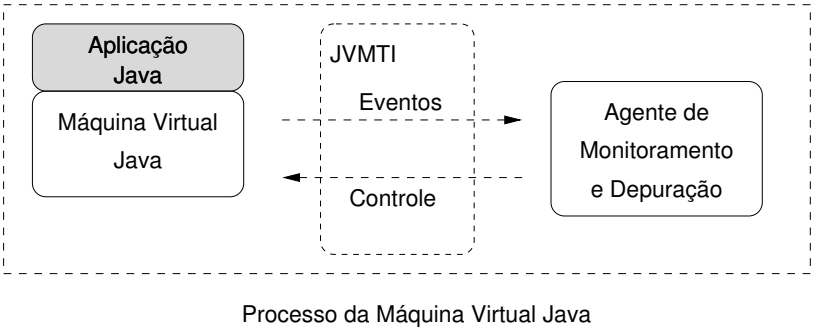

Figura 2.3: Arquitetura Geral de uma ferramenta baseada em JVMTI

Uma das possibilidades interessantes da JVMTI é a chamada inserção de *bytecodes* (*Byte Code Insertion - BCI*), também chamada de injeção ou instrumentação de *bytecodes*.

Essa inserção de código pode ser feita dinamicamente, estaticamente ou no tempo de carga. A instrumentação é estática quando as classes são alteradas mesmo antes de serem carregadas pela máquina virtual. A instrumentação dinâmica acontece quando uma classe já carregada precisa ser modificada durante a execução da aplicação. Essa classe pode ser alterada diversas vezes e inclusive voltar ao seu estado original. Quando a instrumentação ocorre em tempo de carga, o código da classe é enviado para o agente JVMTI que faz a instrumentação necessária. É importante ressaltar que a JVMTI não faz a instrumentação em si, mas apenas possibilita que esta aconteça (O'HAIR, 2004).

Quando consideramos monitoramento, a instrumentação é feita através da notificação de eventos específicos, que são habilitados e desabilitados de acordo com a necessidade do agente. Alguns eventos relevantes para o monitoramento de experimentos de injeção de falhas incluem a morte da máquina virtual (`JVMTI_EVENT_VM_DEATH`) e a captura de exceções (`JVMTI_EVENT_EXCEPTION`).

O recurso de BCI oferecido pela JVMTI pode ser usado para inserir código de injeção de falhas em aplicações Java. Entre as vantagens do uso de um agente JVMTI, é que este dispensa a alteração tanto da aplicação sob teste como da própria máquina virtual. O agente é implementado como uma biblioteca dinâmica, que é carregada opcionalmente pela máquina virtual durante sua inicialização. Em situações de injeção de falhas, o uso do agente só é necessário durante a fase de teste do software. Uma de suas desvantagens é a diminuição da sua portabilidade se comparado com abordagens como POA e reflexão computacional. Devido a interface oferecida ser nativa, o agente é obrigatoriamente implementado em uma linguagem nativa, exigindo, no mínimo, uma recompilação quando houver mudança de plataforma.

2.9 Conclusão do Capítulo

Este capítulo apresentou as técnicas disponíveis para realizar testes de injeção de falhas, sendo elas: hardware, simulação, software e híbrida (hardware e software). Posteriormente foram classificadas falhas de comunicação, que serão as abordadas pela ferramenta desenvolvida neste trabalho e também um modelo de falhas para sistemas distribuídos.

As ferramentas de injeção de falhas brevemente descritas incluíram tanto ferramentas que abordam apenas falhas de comunicação, quanto aquelas que se empenharam em desenvolver uma arquitetura modular, onde é permitida a expansão do modelo de falhas. Foram apresentadas também duas ferramentas de injeção de falhas que têm sua arquitetura distribuída, assim como a ferramenta apresentada neste trabalho.

As ferramentas aqui apresentadas foram todas desenvolvidas dentro de universidades, porém a importância de realização de testes de injeção de falhas são também reconhecidas pela indústria. Tais testes já foram empregados em várias situações como, por exemplo, na avaliação de sistema de trens (AMENDOLA et al., 1997), na validação dos mecanismos de tratamento de erros para supercomputadores da Intel (CONSTANTINESCU, 1998) e na avaliação de sistemas embarcados de controle de aviões (VARDANEGA et al., 1995).

Neste capítulo foram também destacadas a criação de cenários para experimentos de injeção de falhas, bem como a necessidade de monitoramento do experimento para a obtenção de medidas de dependabilidade. Por fim foram apresentadas algumas abordagens conhecidas para a construção de ferramentas de injeção de falhas que permitem o teste de programas desenvolvidos em Java.

A ferramenta FIONA, que é apresentada neste trabalho, tem como objetivo a valida-

ção de aplicações distribuídas em larga escala. A solução apresentada por FIONA difere das outras por ser distribuída e escalável, facilitando a configuração de experimentos com cenários com falhas em múltiplos nodos e simplificando a injeção de falhas de particionamento de rede. A aplicação dos injetores descritos neste capítulo para um cenário distribuído de larga-escala é difícil, visto que o controle de cada injetor é individual. Os injetores distribuídos aqui apresentados também não oferecem facilidades para condução de experimentos em sistemas de larga escala, já que são projetados para uso em sistemas distribuídos de pequena escala. Seu uso sem adaptações para sistemas largamente distribuídos poderia gerar uma sobrecarga temporal inaceitável para a execução da aplicação sob teste. O próximo capítulo apresenta a ferramenta FIONA, juntamente com sua arquitetura, modelo de falhas e expansibilidade da ferramenta.

3 INJETOR DISTRIBUÍDO DE FALHAS - FIONA

Como enfatizado no capítulo 1, em ambientes distribuídos de larga escala não são somente necessárias novas técnicas de tolerância a falhas, mas também novas abordagens para o teste e avaliação destes métodos. O uso direto dos injetores de falhas existentes, como os citados na seção 2.4, acarretam em algumas dificuldades na condução de experimentos de testes em aplicações que executam em ambientes largamente distribuídos. A dificuldade inicia logo na especificação do cenário de falhas, onde cada injetor deve ser configurado individualmente. É necessário ser configurado um cenário específico para cada nodo que sofrerá injeção de falhas durante sua execução. O desafio aumenta se consideramos a configuração de uma falha que afeta mais de um nodo, como o particionamento de rede. Neste caso, a falha deve ser configurada em todos os nodos participantes. Outro empecilho diz respeito à obtenção de medidas de dependabilidade, pois considerando que em injetores individuais não há um *log* único do experimento, mas sim vários *logs* individuais de cada instância do injetor, potencialmente existem horários diferentes.

Para superar estes problemas, foi desenvolvido o injetor de falhas FIONA – *Fault Injector Oriented to Network Applications* –, que é baseado nas seguintes premissas:

- ser escalável para sistemas distribuídos de larga-escala;
- considerar um modelo de falhas consistente com este tipo de sistema;
- não alterar o código da aplicação sob teste;
- permitir a configuração de experimentos e a análise *post-mortem* de *logs* de forma centralizada.

As seções seguintes apresentam o modelo de falhas considerado, a arquitetura local e distribuída da ferramenta, como as falhas são disparadas e como o modelo de falhas de FIONA pode ser expandido.

3.1 Modelo de Falhas da Ferramenta

Em um contexto distribuído, a ocorrência de falhas deve ser assumida como seguindo um modelo de falhas para sistema distribuídos, como os citados da seção 2.3. Como o objetivo de FIONA é a validação de sistemas distribuídos de larga escala, foram consideradas as falhas mais comuns neste tipo de ambiente, dentre aquelas descritas no modelo proposto por Birman (1996). Entre as falhas que afetam as mensagens, são consideradas: colapso, omissão de envio e recepção, temporização e duplicação, sendo esta um subtipo de falhas bizantinas. Para falhas de rede, a ferramenta emula o particionamento de rede.

A emulação de falhas de particionamento de rede com injetores de falhas locais é um desafio, pois o injetor de cada nodo exige uma configuração diferente. Cada nodo deve estar em colapso para todos os nodos da partição oposta. Com FIONA, esta configuração é trivial. Considerando que o injetor de falhas é distribuído, a falha pode ser inicializada de forma centralizada, e notificar os outros componentes do sistema durante a execução do experimento de validação.

Atualmente, FIONA considera injeção de falhas de rede no protocolo UDP. Apesar da entrega de pacotes UDP ser não confiável, esta é freqüentemente usada como base para o desenvolvimento de aplicações de rede ou protocolos de comunicação em *middleware*, onde a confiabilidade necessária é implementada em camadas superiores. Um exemplo é o sistema de comunicação de grupo JGroups (BAN, 1998), que é usado como um bloco básico para a construção de aplicações distribuídas Java de alta disponibilidade. JGroups usa UDP como padrão em sua pilha de protocolos, e são as camadas superiores as responsáveis pelas operações que garantem a confiabilidade requisitada pela aplicação. Neste caso, uma ferramenta de injeção de falhas baseada em UDP é essencial para a validação experimental destas camadas superiores.

Para o protocolo UDP, um modelo mais amplo de falhas bizantinas não é implementado. O teste que poderia ser conduzido seria um teste caixa preta, sem interpretação do conteúdo do pacote UDP, representado em Java pela classe `java.net.DatagramPacket`. A alteração possível seria a troca direta de determinados *bytes* através do método `setData()`, gerando troca de valores em uma região do conteúdo do pacote. Esta alteração pode implicar duas situações: em uma troca de valores de determinados atributos de um objeto transmitido, ou então em um erro na leitura do objeto transmitido no momento da recepção. O erro na leitura é manifestado na forma de exceção para a aplicação, portanto sendo obrigatório o seu tratamento. A injeção de falhas maliciosas, como a emulação de ataques de segurança, requer o desenvolvimento de uma ferramenta específica e não pode ser conduzido apenas com a alteração de *bytes* do pacote.

3.2 Arquitetura de FIONA

A arquitetura de FIONA pode ser dividida em duas partes bastante distintas. A primeira é sua arquitetura local, que é como a ferramenta funciona individualmente em cada nodo. A segunda parte é sua arquitetura distribuída, que é usada para a condução de experimentos onde é necessário que o injetor seja instanciado em mais de um nodo. O projeto de uma ferramenta distribuída não impede que esta seja usada como as ferramentas tradicionais, ou seja, apenas em um nodo do sistema ou então que cada injetor seja controlado individualmente, quando este estiver instanciado em mais de um nodo.

3.2.1 Arquitetura Local

FIONA visa a validação de aplicações distribuídas desenvolvidas em Java, usando a abordagem de injeção de falhas no sistema de troca de mensagens. Para fazer a interceptação de mensagens, FIONA instrumenta o código do protocolo alvo. No caso de aplicações Java, são instrumentadas as classes de sistemas que implementam um determinado protocolo e que implementam a troca de mensagens. Essa instrumentação é feita usando os recursos de BCI da interface de monitoramento e depuração JVMTI. A instrumentação é mista, pois o código da classe de sistema é alterado estaticamente e este substitui, em tempo de carga, o código original do protocolo. FIONA atualmente injeta

falhas no protocolo UDP e está sendo expandida para injeção de falhas no protocolo TCP.

A figura 3.1 mostra a arquitetura local de FIONA. A aplicação Java executa sobre a máquina virtual, invocando métodos das classes de sistema. Quando FIONA está ativo, o agente JVMTI é carregado. Este faz a substituição das classes de sistema originais pelas instrumentadas, as quais interagem com as classes de injeção de falhas. Como pode ser visto na figura, FIONA é composta de três blocos: o agente JVMTI, codificado na linguagem C, as classes de injeção de falhas e o protocolo instrumentado, ambos codificados em Java. Cada bloco é explicado em maior detalhamento na seção 3.3.

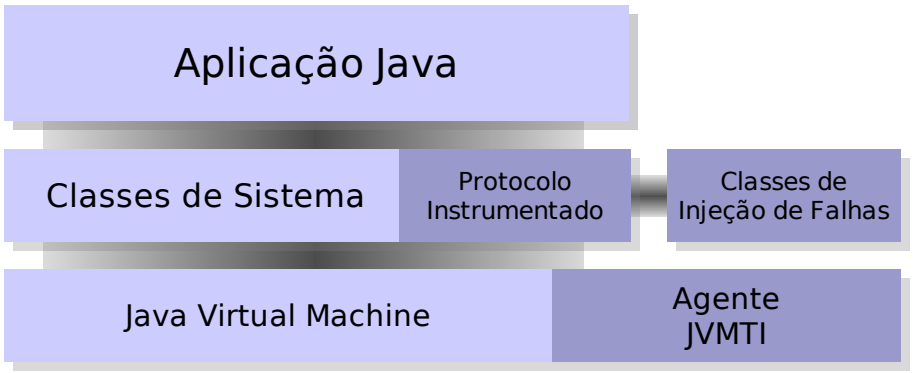

Figura 3.1: Arquitetura local de FIONA

3.2.2 Arquitetura Distribuída

A proposta de arquitetura distribuída de FIONA tem o objetivo de cumprir as principais premissas estabelecidas pela ferramenta, sendo estas: (i) escalabilidade; (ii) permitir a configuração e análise de *log* do experimento de forma centralizada; e (iii) emulação de um modelo de falhas consistente com sistemas de larga escala. Nesta seção é mostrado como cada uma destas premissas são trabalhadas.

Para alcançar escalabilidade, a arquitetura de injeção distribuída de falhas de FIONA foi baseada em arquiteturas de monitoramento de larga escala. Técnicas de injeção de falhas têm similaridades com técnicas de monitoramento, entretanto ao invés de somente inspecionar uma determinada aplicação, a injeção de falhas também pode modificar o seu comportamento. Portanto, foram escolhidas duas estratégias escaláveis para monitoramento de sistemas distribuídos de larga-escala, sendo elas: GRM (BALATON et al., 2001) e Ganglia (MASSIE; CHUN; CULLER, 2003).

A estratégia proposta por GRM consiste em estabelecer uma hierarquia entre os monitores, sendo estes classificados em: *monitor principal*, *monitor de site* e *monitor local*. Os monitores locais são executados em cada máquina do sistema e se comunicam com os monitores de *site*. Cada monitor de *site* recolhe todas as informações e as passam ao monitor principal, que coordena os monitores de *site* e a coleta global de dados. Um monitor de *site* é iniciado em cada centro de computação, sendo seu principal objetivo deixar mais eficiente a comunicação.

Ganglia considera sistemas distribuídos de larga escala baseado em *clusters* federados. O protocolo TCP é usado para comunicação inter-cluster e *multicast* UDP para comunicação intra-cluster.

Em FIONA, é combinada a arquitetura baseada em árvore proposta por GRM e a estratégia de protocolos usada por Ganglia. A Figura 3.2 mostra como a arquitetura de

monitoramento foi adaptada para uma arquitetura de injeção distribuída de falhas. A comunicação entre os *injetores locais* e os *injetores de site* é baseada em UDP *multicast*, e a comunicação entre os *injetores de site* e o *injetor principal* é feita via TCP.

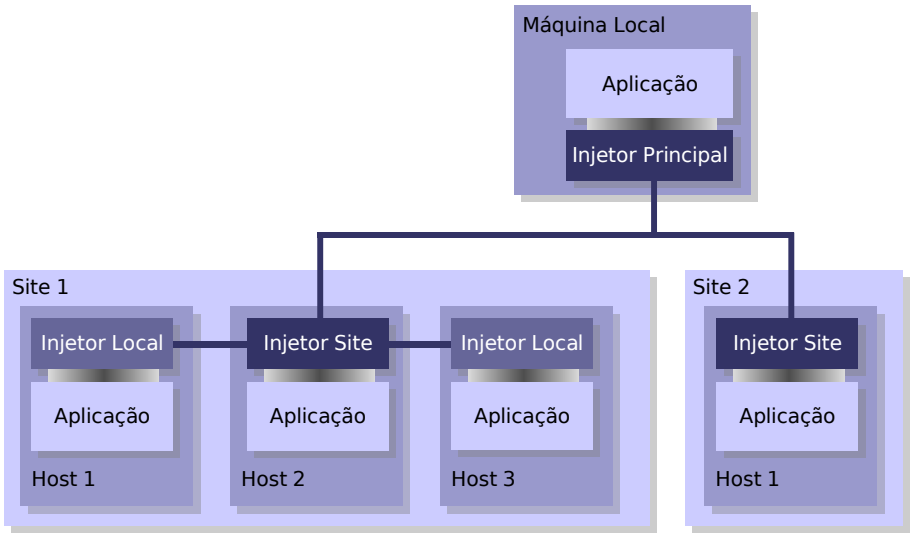

Figura 3.2: Arquitetura distribuída de FIONA

3.2.3 Estrutura de Monitoramento

Para obter medidas de dependabilidade após o experimento de injeção de falhas, é necessário analisar tanto o *log* da ferramenta de injeção de falhas como o *log* da aplicação alvo. FIONA inclue um sistema de monitoramento que visa permitir a análise do *log* do experimento de forma centralizada. Para isso, todos os nodos gravam um registro no seu *log* a cada falha injetada, seja ela um injetor local, de site ou até mesmo principal. Durante um experimento distribuído, cada nodo é responsável por seu próprio *log*, e quando a execução termina este *log* é enviado para o nodo pai na hierarquia (Figura 3.2), se este é um injetor local ou de *site*. Se um nodo é um injetor de *site* ou principal, ele espera até todos os nodos conectados enviarem seus *logs*. Assim que um injetor de *site* recebe algum arquivo de *log*, este é repassado ao injetor principal. Os *logs* da aplicação alvo são dependentes da aplicação e não são tratados pelo sistema de monitoramento de FIONA. Estes devem ser passados como entrada para uma ferramenta apropriada de análise de dependabilidade, em conjunto com os *logs* do experimento de injeção de falhas.

Cada *log* é escrito baseado na hora local do nodo onde é criado. Para estabelecer uma linha de tempo global para o experimento, é usado um algoritmo *offline* de sincronização de relógios, proposto por E. Maillet e C. Tron (1995). Este método é baseado nas diferenças de relógio entre uma máquina de referência r e cada uma das máquinas do sistema. Esta diferença é medida no começo e no final do experimento. r envia uma mensagem para cada máquina, a qual lê o seu tempo local e envia este valor novamente para r. O relógio local de r é lido antes e depois de receber esta mensagem. A média destes dois valores é considerada equivalente ao valor lido na segunda máquina. Os resultados obtidos com essas medições são usados para corrigir os horários de cada um dos *logs* das máquinas.

Este método de sincronização é implementado por um programa auxiliar. Este programa é executado duas vezes, antes do início do experimento e logo após sua finaliza-

ção. Nestas fases, uma operação de *ping-pong* é feita, entre a máquina de referência r e os outros nós do sistema, para recolher dados relacionados ao tempo de cada máquina do sistema distribuído. Esses dados são usados para gerar os parâmetros de correção de tempo de cada máquina. Após estas operações, o programa lê os *logs* coletados por FIONA e ajusta seus horários baseado nos parâmetros de correção. A saída é um *log* único ordenado por tempo com todas as atividades de injeção de falhas. Este método de sincronização de relógios é adequado para sistemas distribuídos de pequena escala. Porém, como para experimentos de injeção de falhas não é necessária uma resolução de tempo com alta precisão, acredita-se que este método atinja o seu objetivo quando aplicado a sistemas largamente distribuidos.

3.2.4 Distribuição de Cenários de Falhas

Para executar um experimento de teste é necessária a criação de um cenário de falhas no qual a aplicação irá executar. Um cenário de falhas define quais os tipos de falhas que serão injetadas durante o experimento de acordo com o modelo de falhas suportado pelo injetor. Como ressaltado no início desta seção, um dos objetivos de FIONA é permitir configuração centralizada, facilitando assim o estabelecimento de cenários de falhas para a condução de um experimento. Considerando que FIONA é um injetor distribuído de falhas, pode-se aproveitar das conexões existentes entre os injetores para distribuir as configurações de falhas relativas a cada nodo.

Com FIONA, a especificação de um cenário de falhas se dá através de sua descrição em um arquivo de configuração. Tal arquivo só é exigido no injetor principal. Todos os outros nodos podem requisitar sua configuração ao nodo pai (injetor de *site* ou principal) se não existir um arquivo local de configuração de falhas. Esta configuração é necessária já na inicialização da máquina virtual, para permitir a instanciação correta do injetor. No momento da interpretação de um arquivo de configuração de falhas, um objeto que representa uma falha (`Fault`) é instanciado para cada falha especificada e inserido no banco de falhas do experimento (`FaultBase`). Durante a execução da aplicação sob teste, o banco de falhas é consultado toda a vez que uma mensagem é recebida ou enviada, para verificar se uma falha será injetada ou não. Por questões de otimização, somente informações relativas a falhas que podem ser injetadas no próprio nodo são fornecidas pelo nodo pai. Poderia ser esperada uma intrusividade temporal excessiva no injetor principal, porém a arquitetura não requer que este nodo execute atividades de injeção de falhas e permite que este apenas controle o experimento.

A última premissa estabelecida é a emulação de um modelo de falhas condizente com o ambiente em execução, ou seja, um sistema distribuído de larga escala. Entre os tipos de falhas consideradas pelo modelo descrito na seção 3.1, o único tipo de falha que não é facilmente emulável em um injetor de falhas local é o particionamento de rede. É justamente este tipo de falha, que se manifesta comumente em um sistema largamente distribuído, que pode tomar vantagem da implementação de um injetor distribuído de falhas. Quando é conduzido um experimento de injeção de falhas usando a arquitetura distribuída, todas as falhas são ativadas localmente, com exceção das falhas de particionamento de rede. No início do experimento, a configuração de uma falha de particionamento é enviada para todos os nodos. A falha é ativada após um tempo pré-determinado em sua configuração. Quando este tempo expira, o injetor principal envia uma mensagem de ativação para todos os injetores de *site*, os quais retransmitem aos seus injetores locais. A falha é emulada como um colapso na conexão com todos os nodos da partição oposta. Quando o tempo de duração da falha expira, o injetor principal envia uma mensagem de desativação e en-

tão a partição se reconecta (*merge*). Esta comunicação entre injetores é implementada no agente JVMTI (código nativo), usando as bibliotecas do próprio sistema operacional. Se esta implementação usasse a API de *sockets* do próprio Java, tais conexões estariam sujeitas a atividades de injeção de falhas. Desta forma, é evitado que estas mensagens passem pelo código de injeção de falhas, dispensando a necessidade de qualquer controle adicional.

3.3 Implementação do Protótipo

Após a exposição da arquitetura da ferramenta, tanto local como distribuída, é apresentada a implementação do protótipo. Como detalhado na subseção 3.2.1 e mostrado na Figura 3.1, a ferramenta tem três blocos básicos: o agente JVMTI, o protocolo instrumentado e as classes auxiliares de injeção de falhas. Cada bloco é explicado a seguir, enfatizando suas funções básicas na arquitetura local e seu papel na arquitetura distribuída.

3.3.1 Agente JVMTI na Arquitetura Local

Na arquitetura local, o agente JVMTI tem três funções básicas: (i) interceptar o carregamento e substituir o código das classes de sistema que implementam o protocolo que terá falhas injetadas; (ii) inicializar as classes estáticas do injetor, como classes de monitoramento e o banco de falhas do experimento; (iii) receber opções de disparo do injetor, como o nome de arquivo de falhas e se o experimento será local ou distribuído.

Quando uma aplicação é disparada com FIONA, a primeira função executada do agente JVMTI é `Agent_OnLoad`, que é chamada assim que este é carregado. O agente deve registrar os eventos dos quais será notificado durante a execução da aplicação e também quais suas funções de tratamento. No caso de FIONA, três eventos são significativos. O primeiro é o evento que indica a inicialização completa da máquina virtual (`JVMTI_EVENT_VM_INIT`). É no tratamento deste evento que as classes de monitoramento e o banco de falhas do experimento são inicializadas. A função de tratamento do segundo evento, nomeado `JVMTI_EVENT_CLASS_FILE_LOAD_HOOK`, é invocada toda vez que uma classe é carregada. Em FIONA, esta função é programada para verificar se a classe atualmente em processo de carga deve ter seu código substituído ou não. Se a classe em questão precisa ser substituída, o trecho de memória ocupado pela mesma é sobrescrito pelo arquivo da classe previamente instrumentada e compilada com código de injeção de falhas. É este mecanismo disponibilizado pela JVMTI que permite que isto ocorra transparentemente para a aplicação, o que evita que o código desta seja alterado para que testes de injeção de falhas sejam conduzidos. O último evento ocorre quando a máquina virtual é finalizada (`JVMTI_EVENT_VM_DEATH`), seja pelo término correto ou abrupto da aplicação. Em seu tratamento, é gravado em disco o *log* do experimento, que é feito pelas classes de monitoramento.

A figura 3.3 mostra o código resumido das ações que o agente JVMTI executa ao ser carregado. Primeiramente é criada a estrutura de funções de tratamento (*callbacks*), onde é indicado o endereço de cada função. Posteriormente, é solicitada à máquina virtual a notificação dos eventos supracitados. Por fim, as opções passadas para o agente são analisadas. As opções incluem o tipo do injetor (se executará localmente ou distribuído), além de ser verificado se existe um arquivo de descrição de falhas indicado. Esta opção é significante quando o injetor está executando de forma distribuída.

```
JNIEXPORT jint JNICALL Agent_OnLoad(JavaVM *vm, char *options,
                                    void *reserved) {
   int ret = 0;
   jvmtiEventCallbacks callbacks;

   // Obtenção do ambiente JVMTI
   (*vm)->GetEnv(vm, (void **) &jvmti, JVMTI_VERSION_1_0);

   // Indicação do endereço das funções de tratamento
   callbacks.VMInit = (jvmtiEventVMInit) vmInitCallback;
   callbacks.VMDeath = (jvmtiEventVMDeath) vmDeathCallback;
   callbacks.ClassFileLoadHook = (jvmtiEventClassFileLoadHook)
                                      loadHookCallback;
   (*jvmti)->SetEventCallbacks(jvmti, &callbacks,
                                    sizeof(callbacks));

   // Habilitação para notificação dos eventos
   (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE,
                JVMTI_EVENT_VM_INIT, NULL);
   (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE,
                JVMTI_EVENT_VM_DEATH, NULL);
   (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE,
                JVMTI_EVENT_CLASS_FILE_LOAD_HOOK, NULL);

    // Configura opções passadas ao injetor
    hasFaultFile = setup(options, faultFile);

    return ret;
}
```

Figura 3.3: Implementação de FIONA para o agente JVMTI

3.3.2 Agente JVMTI na Arquitetura Distribuída

Quando o injetor é usado distribuído, o agente JVMTI desempenha funções adicionais, pois deve executar seções diferentes de código dependendo de qual tipo de injetor este é na hierarquia explicada na subseção 3.2.2. Tal identificação é feita no lançamento do programa, em sua linha de comando, onde é indicado se o injetor irá se comportar como um *injetor principal*, *injetor de site* ou *injetor local*. A seguir são descritos o funcionamento do injetor principal e dos injetores de *site*. Os injetores locais foram discutidos na seção 3.3.1.

3.3.2.1 Injetor Principal

O *injetor principal* é o responsável pelo gerenciamento do experimento. É esta instância que distribui a configuração do experimento e faz o recolhimento dos *logs* de cada máquina participante, além de ser a responsável pela ativação de falhas de particionamento de rede. Para implementar a distribuição de configurações, o agente JVMTI lança um fluxo de execução adicional (*thread*). Tal *thread* abre um *socket*, o qual fica aguardando a conexão de injetores de *site*. Quando chega uma requisição de conexão, são consultadas no banco de falhas local as falhas relativas a este injetor e seus respectivos injetores locais. Tais falhas são enviadas, de forma textual, para o injetor de *site*, o qual distribui para os injetores locais quando requisitado. Após, o injetor principal insere o novo *socket*, que representa a conexão com o injetor de *site*, em uma tabela de conexões. No caso de o injetor de *site* ter sua configuração própria e não precisar requisitar ao injetor principal, este apenas envia um pedido de conexão, o que implica apenas ter sua conexão

inserida na tabela.

Para recolher os *logs* dos outros injetores, o injetor principal cria outra *thread*, a qual é responsável pela verificação de atividades de escrita nos *sockets* contidos na tabela de conexões. Esta implementação é feita com a chamada `select()` do sistema operacional, que verifica se houve escrita em um determinado conjunto de descritores de arquivos, neste caso os *sockets* com os injetores de *site*. A chamada retorna quando há escrita em pelo menos um *socket* ou então um *timeout* expira. Se houve escrita, o injetor principal recebe o *log* e, se necessário, remove a conexão da tabela. O log recebido é gravado localmente em um arquivo, contendo o nome do nodo que o gerou. Isto posteriormente permite a identificação de qual índice de correção de relógio será usado para a correção de horários. Se o *timeout* expirou, a tabela de conexões é relida (para verificar a existência de novas conexões), e `select()` é novamente chamado. Este procedimento é executado até que o *log* de todas as máquinas tenha sido recolhido.

Como descrito na seção 3.2.2, a ativação de falhas de particionamento de rede é dada no injetor principal. A falha é configurada para ser ativada em um determinado tempo da execução, que quando expira, deve ser ativada em todos os nodos participantes. Uma falha de particionamento é identificada por um nome, que é criado automaticamente a partir da leitura do arquivo de configuração do experimento. Este nome é repassado aos nodos inferiores da hierarquia no corpo da mensagem de ativação, e é usado como o identificador de falha no momento da ativação e desativação. A implementação da ativação da falha por tempo é feita nas classes Java, porém quem detém a tabela de conexões é o agente JVMTI, implementado em C. Para possibilitar o envio dessas mensagens, no momento da ativação da falha, a classe Java faz uma chamada ao código nativo via JNI. A função nativa então envia a mensagem para todos os *sockets* da tabela de conexões. Os injetores de *site* repassam para os injetores locais via *multicast* UDP, que, quando recebem a mensagem, iniciam a injeção de falhas emulando um colapso para todas as conexões definidas na configuração. O mesmo processo ocorre na desativação da falha.

3.3.2.2 Injetor de Site

O *injetor de site* funciona de forma similar ao injetor principal, diferenciando-se apenas no protocolo de comunicação com os injetores inferiores da hierarquia (UDP *multicast*). Desta forma, não é necessário manter uma tabela de conexões, mas apenas uma contagem do número de injetores locais associados. Esta contagem é incrementada no momento de pedido de configuração e decrementada no momento da entrega do *log* do experimento. No lançamento dos injetores de *site*, estes são informados na linha de comando qual o endereço IP (*Internet Protocol*) do nodo hospedeiro do injetor principal. O mesmo ocorre com os injetores locais, que são informados do endereço IP do injetor de *site* ao qual deve se associar.

3.3.3 Instrumentação do Protocolo

Como mencionado no início da seção, a instrumentação de FIONA é mista, sendo esta parcialmente estática e parcialmente em tempo de carga. A alteração em tempo de carga é feita pelo agente JVMTI, o qual detecta o carregamento da classe do protocolo e faz a substituição de seu código. Este novo código contém o mesmo código da classe original, porém adicionado da lógica das operações de injeção de falhas. Tais operações são inseridas em pontos de interceptação previamente estudados e determinados, que variam de acordo com o protocolo em questão.

O protocolo atualmente suportado por FIONA é UDP, que é implementado pela classe

de sistema `java.net.DatagramSocket`. Esta classe contém dois métodos públicos de troca de mensagens, sendo estes (`send()` e `receive()`). Tais métodos estabelecidos como pontos de interceptação para injeção de falhas de comunicação neste protocolo. Para instrumentar este classe, primeiramente é copiado o código original destes métodos para um novo método, com os mesmos parâmetros, porém com um nome diferente (`originalSend()` e `originalReceive()`). Assim, os métodos da API contém apenas código para atividades de injeção de falhas e os métodos originais são invocados sempre que necessários.

Tanto a instrumentação do método de envio como o de recepção contém operações similares, onde interagem com classes auxiliares de injeção de falhas. Essas classes incluem um banco de falhas do experimento, classes de descrição de cada tipo de falha e uma classe de monitoramento, que guarda as atividades do injetor de falhas para posterior gravação em um *log*.

A seqüência básica de operações dos métodos instrumentados inicia pela consulta ao banco de falhas do experimento, verificando se há alguma falha relacionada ao *socket* em questão. Se não há falhas a serem injetadas, o método original é invocado e seu resultado retornado para a aplicação. Se estas existem, é verificado se a falha já está ativa (de acordo com o tempo da execução ou número da mensagem trocada), e se esta será injetada na mensagem sendo enviada/recebida. Em caso positivo, é tomada uma ação de acordo com o tipo de falha (omissão ou atraso de mensagem, por exemplo). Se a falha foi injetada, é gravada uma entrada no *log* do experimento.

3.3.3.1 Instrumentação de Envio e Recepção em UDP

A figura 3.4 mostra a instrumentação do método `send()` para falhas de omissão e colapso. Nota-se que a consulta ao banco de falhas pode retornar mais de uma falha. Isso permite que seja configurado mais de um tipo de falha para o mesmo *socket*, sendo cada uma ativada em momentos diferentes. Por exemplo, inicia-se um experimento contendo falhas de omissão em uma determinada conexão e esta após um tempo colapsa completamente. A invocação do método `inject()` determina se a falha está ativa no momento e se esta deve ser injetada ou não (de acordo com uma probabilidade, por exemplo). Pode ser vista também a ação tomada em caso de falhas de omissão e colapso. Se a falha não deve ser injetada, o método original de envio de mensagens é invocado. No outro caso apenas é chamada a rotina de registro de *log* e o método de envio retorna para a aplicação como se a mensagem tivesse sido normalmente enviada, porém na realidade esta foi omitida.

A figura 3.5 mostra a instrumentação feita no método `receive()` para os mesmos tipos de falhas. A diferenciação básica entre a implementação do envio e da recepção, é que na última a chamada original sempre deve ser invocada. Para possibilitar a análise da falha ser ou não injetada, o pacote primeiramente deve ser lido da rede, o que implica em colocar como primeira ação uma invocação ao método original. Após o pacote ter sido lido da rede, este vai ser verificado se terá falhas injetadas ou não. Se não houver, a execução sairá do laço de repetição e retornará para a aplicação. Se houver falha a ser injetada, tal falha é gravada no *log* e é aguardada a chegada de outro pacote. O laço executa até receber uma mensagem sem falhas.

Nas figuras 3.4 e 3.5 foram mostradas as implementações apenas de falhas de omissão e colapso, tanto no envio como na recepção, por questões de clareza e facilidade de entendimento.

```
public void send(DatagramPacket p) throws IOException {
   boolean toSend = true;

   // Consulta ao banco de falhas
   Vector faults = FaultBase.getFaults(p);

   // Verificação de múltiplas falhas para um mesmo socket
   for (int i = 0; i < faults.size(); i++) {
      Fault f = (Fault) faults.get(i);
      if (!f.isTransmission())
         continue;
      // Se falha é de omissão ou colapso e deve ser injetada
      if ( f instanceof UdpOmissionFault ||
           f instanceof UdpCrashFault ) {
         if (f.inject())
            toSend = false;
      }
    }
    if (toSend) {
        // Falha não injetada e mensagem enviada
        originalSend(p);
        return;
    } else {
        // Falha injetada e mensagem não enviada
        // Gravação no log
        Monitor.logFault(f, p);
    }
}
```

Figura 3.4: Instrumentação do método `send()` no protocolo UDP para falhas de omissão e colapso

3.3.3.2 Extensão para Multicast

A vantagem de implementar falhas de omissão tanto no envio como na recepção é que deste modo é permitida a validação de protocolos que usem primitivas de *multicast*. Considerar omissões somente no envio implica em uma mensagem ser suprimida para todos os nodos de um determinado grupo *multicast*. Com omissões na recepção, também é possível a emulação de uma falha de recebimento de uma mensagem para somente um dos membros do grupo. Desta forma, pode-se validar experimentalmente, por exemplo, protocolos de *multicast* confiável, onde é definido que se um membro do grupo recebe uma determinada mensagem, todos os membros devem recebê-la.

Em aplicações que usam *multicast*, a troca de mensagems é feita usando os métodos `send()` e `receive()` da classe `java.net.MulticastSocket`, que é uma subclasse de `java.net.DatagramSocket`. Devido a classe *multicast* não reimplementar os métodos de comunicação, são os métodos da superclasse que são efetivamente invocados. Considerando que estes métodos já estão instrumentados, pode-se também injetar falhas em *sockets multicast*. O mesmo se aplica a qualquer outro protocolo que for construído com herança da classe `java.net.DatagramSocket` e que chama seus métodos. Estes estarão automaticamente instrumentados e pronto para serem validados.

3.3.4 Classes de Injeção de Falhas

Como explicado anteriormente, as classes de injeção de falhas são codificadas em Java. Esta abordagem proporciona, além de uma maior portabilidade, uma diminuição

```
public void receive(DatagramPacket p) throws IOException {
   boolean toReceive = true;

   do {
      // Recepção do pacote
      originalReceive(p);
      toReceive = true;

      // Consulta ao banco de falhas
      Vector faults = FaultBase.getFaults(this, p);

      // Verificação de múltiplas falhas para um mesmo socket
      for (int i = 0; i < faults.size(); i++) {
         Fault f = (Fault) faults.get(i);
         if ( !f.isReception()  )
               continue;
         // Se falha é de omissão ou colapso e deve ser injetada
         if ( f instanceof UdpOmissionFault ||
              f instanceof UdpCrashFault ) {
            if (f.inject()) {
               // Injeção da falha
               Monitor.logFault(f, p);
               toReceive = false;
            }
         }
   } while (!toReceive);
}
```

Figura 3.5: Instrumentação do método `receive()` no protocolo UDP para falhas de omissão e colapso

na sobrecarga temporal imposta pelo injetor de falhas. Potencialmente, a execução das atividades de injeção de falhas no código nativo seria mais rápida, porém a JVMTI usa JNI (*Java Native Interface*), que impõe uma penalidade de desempenho em aplicações que trocam constantemente entre o modo nativo e o modo *bytecode*. Portanto, com o código de instrumentação escrito puramente em Java, este é executado em velocidade máxima, podendo também ser otimizado pela máquina virtual.

Atualmente, FIONA contém dez classes de injeção de falhas auxiliares, sendo distribuídas em: uma para armazenamento de falhas, uma de monitoramento, uma para implementação de ativação temporal de falhas, e sete para descrição dos tipos de falhas.

Para armazenamento das falhas, tem-se o banco de falhas do experimento (`FaultBase`), que contém todas as falhas que poderão ser injetadas durante a execução da aplicação sob teste. Esta classe ainda é responsável pela interpretação do arquivo de falhas passado para o injetor em seu lançamento. O banco armazena tanto falhas relativas ao próprio nodo, como em relação aos demais nodos, no caso do injetor de *site* e injetor principal. O banco mantém duas estruturas internas, sendo uma para falhas locais (relativas ao próprio nodo), e outra para falhas remotas (relativas aos outros nodos). Esta divisão é feita para diminuir a intrusividade temporal na aplicação, já que a estrutura de falhas locais é consultada a cada envio ou recepção de mensagens. As falhas remotas não são instanciadas como objetos da classe de falha (Fault) e sim como própria cadeia de caracteres lida do arquivo de configuração. Desta forma, a distribuição das falhas é facilitada, pois já se encontram em forma textual, e são facilmente interpretadas pelo outro nodo.

A classe de monitoramento (`Monitor`) armazena todas as mensagens para gravação posterior no *log* do experimento. Cada implementação de falha possui dados específicos para gravação, que são conhecidos pela implementação do método `log()`. O formato básico do registro é a hora do acontecimento do evento (falha injetada) juntamente com os dados de cada falha. Esta classe também é responsável pela gravação do *log* em arquivo. Tais métodos são invocados apenas pelo injetor mestre, responsável pelo recolhimento dos logs do experimento.

Para ativação de falhas, existem duas maneiras: ativação por número de mensagens e ativação por tempo. A primeira maneira é gerenciada dentro de cada falha, que mantém um contador que verifica se a falha deve ser ativada a partir daquela mensagem ou não. Para a ativação por tempo, é definido qual tempo em segundos, a partir do início da execução, que a falha poderá ser injetada ou não (dependendo de suas outras configurações). Pode também ser definido um tempo para a falha ser desativada. Esta ativação/desativação é implementada através de uma *thread* (`TriggeringThread`), que fica suspensa até expirar os tempos estabelecidos. Esta *thread* é inicializada no momento da instanciação da falha pelo banco de falhas.

A descrição dos tipos de falhas é feita a partir de uma classe genérica de falha (`Fault`). Esta classe tem atributos básicos, como sua repetição (permanente, transiente ou intermitente), sua configuração de início e fim, se está ativa ou não, entre outros. Cada tipo de falha é uma subclasse desta classe genérica, e especializa determinados métodos, como o de determinação de injeção de falhas (`inject()`) e o de gravação de registro no *log*. As classes implementadas incluem falhas de omissão (`UdpOmissionFault`), temporização (`UdpTimingFault`), duplicação (`UdpDuplicationFault`), colapso (`UdpCrashFault`) e particionamento de rede (`UdpPartitioningFault`). A falha de temporização exige uma classe adicional (`UdpTimingThread`), a qual implementa a *thread* que simula o atraso de uma determinada mensagem.

3.4 Expansão do Modelo de Falhas

Uma característica adicional de FIONA é a possibilidade de expansão de seu modelo de falhas e de seus protocolos alvo. O desenvolvimento de uma ferramenta expansível tem a vantagem de ser adaptável a outros tipos de falhas e sistemas, aumentando sua reusabilidade. Se começarmos a partir de uma ferramenta pré-existente, o tempo de desenvolvimento é bem mais curto, pois é necessário somente o desenvolvimento do código específico da expansão.

O modelo de falhas de FIONA pode ser expandido para adicionar novos tipos de falhas, tanto para o próprio protocolo UDP, como para novos protocolos. Como pode ser visto na Figura 3.1, as classes de FIONA podem ser divididas em duas categorias: o protocolo instrumentado e as classes genéricas de injeção de falhas. Esta última categoria inclui o banco de falhas, responsável por interpretar e armazenar as falhas que serão usadas no experimento, e as classes que descrevem estas falhas, que são subclasses da classe falha (`Fault`). São estas classes que implementam o método `inject()`, responsável pela lógica de ativação da falha em si.

Se considerarmos o protocolo UDP, são necessários três passos para adicionar um novo tipo de falha: (i) criação de uma classe especializada com sua lógica de ativação; (ii) alteração do banco de falhas para interpretar a nova falha do arquivo de configuração do experimento; e (iii) alteração dos métodos `send()` ou `receive()` da classe instrumentada `java.net.DatagramSocket`. Estes métodos devem ser adaptados para

receber da consulta ao banco de falhas este novo tipo de falha e também decidir como a mensagem será tratada (i.e., duplicada, atrasada, etc).

Considerando outros protocolos, a primeira decisão a ser tomada é onde a falha será ativada, ou seja, quais classes devem ser instrumentadas para permitir a interceptação das mensagens do protocolo. Esta classe deve ser instrumentada para referenciar as classes de injeção de falhas. A transparência para a aplicação é obtida através da indicação ao agente JVMTI que este deve substituir o código desta classe em tempo de carga. Após a instrumentação das classes, novas falhas devem ser adicionadas seguindo os mesmos passos usados para o protocolo UDP. A expansão do injetor de falhas para um novo protocolo requer seu código fonte. Se o usuário não tiver acesso, podem ser usadas ferramentas que alteram diretamente o *bytecode* da classe, como Javassist (CHIBA, 2000) e BCEL (DAHM, 2001).

A adaptação de uma ferramenta de injeção de falhas para a injeção de novos tipos de falhas ou novos sistemas alvo representa um compromisso (*trade-off*) entre generalidade e facilidade de uso (*user-friendliness*) (AIDEMARK et al., 2001). A expansibilidade de FIONA não é trivial. Entretanto é esperado que esta expansão seja feita por desenvolvedores de ferramentas de validação. Atualmente, FIONA está tendo seu modelo de falhas expandido para o protocolo TCP (GERCHMAN, 2004).

3.5 Conclusão do Capítulo

Este capítulo apresentou a ferramenta de injeção de falhas FIONA e suas premissas básicas, sendo elas: (i) escalar para sistemas distribuídos de larga escala, (ii) considerar um modelo de falhas apropriado, (iii) não modificar o código da aplicação alvo do teste de injeção de falhas, e (iv) permitir a configuração e a análise dos *logs* dos experimento de forma centralizada. O capítulo abordou como cada uma das premissas é tratada pela ferramenta, mostrando seu modelo de falhas, e tanto sua arquitetura local como sua arquitetura distribuída.

Também é enfatizado como foi desenvolvida a ferramenta, mostrando a implementação do agente JVMTI, o protocolo instrumentado e as classes auxiliares, abordando a funcionalidade de cada parte para a arquitetura local e a distribuída. Por fim é mostrado como o modelo de falhas de FIONA pode ser expandido, tanto para novos tipos de falhas como para outros protocolos.

Com a arquitetura proposta, FIONA ataca problemas enfrentados por outros injetores na validação de sistemas distribuídos de larga escala, como a dificuldade de estabelecimento de cenários de múltiplas falhas e a emulação de falhas que afetam múltiplos nodos, como particionamento de redes. Através destas características e também da possibilidade de análise centralizada do *log* do experimento, FIONA auxilia na avaliação de dependabilidade de sistemas distribuídos de larga escala.

4 CONDUÇÃO DE EXPERIMENTOS COM FIONA

Este capítulo tem o objetivo de mostrar como podem ser criados cenários de falhas para a ferramenta FIONA, mostrando as opções de configuração de cada falha, bem como o uso de sua interface gráfica. É abordado também como podem ser lançados programas com a ferramenta, tanto localmente como de forma distribuída. Por fim, são mostrados pequenos experimentos que demonstram o funcionamento de FIONA e como esta pode ser usada para a condução de testes de injeção de falhas.

4.1 Especificação de Cenários de Falhas

Como descrito na seção 2.5.1, o cenário de falhas descreve a carga de falhas a qual a aplicação sob teste será submetida. A especificação das falhas depende do modelo suportado pela ferramenta, sendo em FIONA falhas de omissão, duplicação, colapso, temporização e particionamento de redes, sendo todas elas injetadas em aplicações que usam comunicação com o protocolo UDP.

Uma das premissas da ferramenta FIONA é permitir a configuração deste cenário de forma centralizada. Assim, o usuário consegue especificar em apenas um local como todos os componentes, que fazem parte do sistema, terão falhas injetadas. Isto facilita a condução do experimento, pois não exige que o usuário crie um arquivo de configuração para cada nodo individualmente, e também que distribua manualmente estes arquivos para cada um dos nodos que terão falhas injetadas.

FIONA permite a especificação de um cenário de falhas através de sua descrição em um arquivo de configuração. Tal arquivo descreve os tipos de falhas que serão injetadas, em quais nodos, o momento de ativação de cada falha e, dependendo do caso, o seu comportamento. Este arquivo é passado como parâmetro durante a inicialização do programa e é interpretado pelo injetor. O injetor armazena as descrições em seu banco de falhas de acordo com o descrito na seção 3.3.4, ou seja, separando em estruturas de falhas locais e de falhas remotas. A configuração do experimento é distribuída de acordo com a demanda dos injetores inferiores a hierarquia.

Em determinadas ferramentas de injeção de falhas, como ComFIRM e ORCHESTRA, a configuração de cenários de falhas é feita via *scripts*. O uso de *scripts* permite uma maior poder na expressão do comportamento do injetor durante o experimento, porém o uso de arquivos simplifica a configuração de cenários. A configuração via arquivos também pode ser estendida, viabilizando a adição de novos campos que especifiquem comportamentos diferentes. Em FIONA, a escolha de configuração por arquivos se deu por razões de simplicidade no momento da distribuição das configurações para os outros nodos do experimento e também por herança da ferramenta Jaca, cuja expansão para falhas de comunicação (JACQUES-SILVA et al., 2004) foi precursora deste trabalho.

Um esquema de como são criadas falhas para o teste de uma aplicação é mostrado na figura 4.1. Em FIONA, as falhas podem ser descritas através de um editor de texto qualquer ou diretamente em sua interface gráfica, a qual é descrita na seção 4.1.6. Esta criação é baseada no modelo de falhas implementado por FIONA. O arquivo gerado como saída do criador de cenários é a carga de falhas a qual a aplicação será submetida. Este é lido pelo injetor, que faz a interpretação e injeta as falhas descritas na aplicação sob teste.

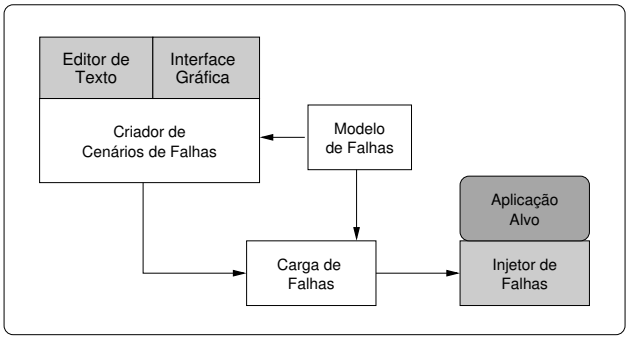

Figura 4.1: Processo de criação de cenários de falhas

Em FIONA, cada falha é descrita através de uma linha de configuração, que contém parâmetros específicos para cada tipo de falha. No arquivo de falhas, cada falha é descrita em uma nova linha e cada parâmetro é separado pelo caracter dois pontos (":").

4.1.1 Falhas de Omissão

Falhas de omissão são implementadas através da supressão de mensagens enviadas pela aplicação e podem ser configuradas da seguinte forma:

```
UdpOmissionFault:<padrão_de_repetição>:<taxa_de_falhas>:
<tipo_ativação(tempo|mensagem)>:<início>:<fim>:
<nodo_origem>:<porta_origem>:<nodo_destino>:<porta_destino>:
<envio|recepção>:
```

O campo de `padrão_de_repetição` indica como a falha irá se repetir durante o experimento, podendo ser permanente (`p`), transiente (`t`) ou intermitente (`i`). Quando configurada como permanente, a falha, a partir de sua ativação, é injetada em todas as mensagens subseqüentes. Falhas transientes se manifestam apenas no momento da ativação da falha e não se repetem mais durante o experimento. Desta forma as falhas podem ser injetadas deterministicamente. As falhas intermitentes se manifestam de acordo com uma taxa de falhas (definida em `taxa_de_falhas`), que é definida como a probabilidade de manifestar uma falha. Por exemplo, uma taxa de 0.8 indica que há 80% de chance de o pacote corrente ser omitido. A cada inspeção de pacote, a probabilidade de perda é a mesma. A perda é definida através da verificação do resultado de uma função que sorteia um número randômico. Essa função obedece uma distribuição uniforme.

O próximo parâmetro é referente ao tipo de ativação da falha (`tipo_ativação(tempo|mensagem)`). Inicialmente a falha está inativa, ou seja, não será injetada até que sua condição de início seja cumprida. A falha pode se tornar ativa por um tempo definido em segundos (`t`) ou pelo número da mensagem que está sendo trocada (enviada ou recebida) por um determinado *socket* (`m`). Os campos de `início` e `fim` de falha definem o intervalo da execução da aplicação que a falha estará

ativa. Se o campo `fim` for especificado como nulo (0) ou menor que `início`, a falha ficará ativa até o final da execução do programa.

A parte final da configuração indica em qual *socket* será injetada a falha. Esta identificação é feita indicando o endereço (`nodo_origem`) e a porta de conexão (`porta_origem`) do nodo emissor, e também os mesmos dados sobre o nodo receptor (`nodo_destino` `porta_destino`). Para flexibilizar a configuração, é permitido o uso do caracter asterisco ("`*`") para quando o usuário dispensar que um determinado campo seja analisado. Por exemplo, se é requisitado apenas que o endereço de destino seja analisado no momento da injeção de falhas, sem considerar a porta de conexão, o usuário deve preencher o campo `porta_destino` com o caractere asterisco.

O último parâmetro (`send|receive`) define em que local a falha será injetada, se será no envio (`s`) ou na recepção (`r`) da mensagem. O uso deste campo não interfere na ordem da definição dos endereços de origem e destino, que sempre serão interpretados com a mesma semântica. Este campo implica na determinação de qual nodo irá interpretar a regra. Por exemplo, se uma regra for configurada com o endereço de origem como `buick.inf.ufrgs.br` e o endereço de destino como `maverick.inf.ufrgs.br` e a regra for configurada como injeção de falha no envio, a regra será interpretada por `buick` e irá omitir as mensagens enviadas para `maverick`. Caso a regra seja configurada com injeção na recepção, será interpretada por `maverick`, que irá omitir mensagens advindas da máquina `buick`.

4.1.2 Falhas de Duplicação

A configuração de falhas de duplicação de mensagens é similar a falhas de omissão, diferenciando-se apenas por não possuir o parâmetro (`send|receive`). Este parâmetro não é incluído por não proporcionar um comportamento diferente quando se trata de falhas de duplicação de mensagens. As falhas de duplicação são implementadas através do envio duplo de uma mensagem, a partir de uma invocação única do método de envio de mensagens pela aplicação. Este tipo de falha é especificada com a seguinte linha:

```
UdpDuplicationFault:<padrão_de_repetição>:<taxa_de_falhas>:
<tipo_ativação(tempo|mensagem)>:<início>:<fim>:
<nodo_origem>:<porta_origem>:<nodo_destino>:<porta_destino>:
```

4.1.3 Falhas de Colapso

As falhas de colapso incluem tanto colapso de *link* como colapso de nodo. A implementação deste tipo de falha é através do corte total da comunicação entre dois ou mais nodos. O corte é feito através da supressão de todas as mensagens enviadas ou recebidas por um determinado canal. Um processo que não pode enviar mensagens é percebido como falho pelos outros nodos do sistema. As mensagens recebidas também são suprimidas para evitar que o processo troque seu estado. Esta mudança de estado poderia significar uma emulação errônea do comportamento de uma falha, no caso, por exemplo, de uma falha temporária de colapso de *link*. Através desta implementação não é possível a emulação das variadas classificações de falhas de colapso, como descrita por Cristian (CRISTIAN, 1991). Uma falha de colapso é configurada no arquivo da seguinte forma:

```
UdpCrashFault:<tipo_ativação(tempo|mensagem)>:<início>:<fim>:
<nodo_origem>:<nodo_destino>:
```

Para falhas de colapso não é necessária a especificação de portas na configuração, pois quando uma falha de colapso se manifesta, ou todas as conexões com todos os nodos são perdidas (colapso de nodo) ou então todas as conexões com um determinado nodo (colapso de *link*). Um colapso de nodo é facilmente emulado através do uso do caracter asterisco no campo `nodo_destino`, representando o endereço de qualquer nodo de destino. Para um colapso de *link*, o usuário pode especificar a conexão com apenas um nodo em particular. Desta forma, qualquer comunicação com esta máquina será cortada. Falhas de colapso também podem ser ativadas a partir de um determinado tempo ou então de um número pré-estabelecido de mensagens. A definição de fim de falha também é possível, mas é plausível apenas no caso de colapso de *link*, já que quando um nodo falha por colapso dificilmente o processo retornará no mesmo estado em que falhou.

4.1.4 Falhas de Temporização

Falhas de temporização provocam o atraso de mensagens da aplicação. Para sua implementação, foi criada uma *thread* que emula o atraso da mensagem. Este novo fluxo faz uma cópia da mensagem original e fica bloqueado pelo tempo que o pacote deve ser atrasado. Quando este tempo expira, a mensagem é enviada normalmente pelo *socket*. A cópia do pacote implica em uma queda de desempenho na execução, porém esta é necessária pois a referência ao pacote pode ser reutilizada e o pacote original ser então perdido. A linha de configuração de falhas de temporização é mostrada a seguir:

```
UdpTimingFault:<padrão_de_repetição>:<taxa_de_falhas>:
<tipo_ativação(tempo|mensagem)>:<início>:<fim>:
<atraso_mínimo>:<atraso_máximo>:<nodo_origem>:
<porta_origem>:<nodo_destino>:<porta_destino>:
```

Os parâmetros de mesmo nome têm a mesma semântica apresentada anteriormente. A nova opção para este tipo de falha é a determinação do intervalo de atraso, definido pelos campos `atraso_mínimo` e `atraso_máximo`. Tais campos são definidos em milisegundos. Caso o campo de atraso máximo seja definido como menor ou igual que o atraso mínimo, o atraso das mensagens é fixo.

4.1.5 Falhas de Particionamento

Um tipo de falha que afeta mais de um nodo por vez é a falha de particionamento de rede. Por diversas razões, como a quebra de um *link*, determinados nodos se isolam de outros, evidenciando uma divisão entre os nodos. Nodos dentro de uma mesma partição se comunicam entre si, mas ficam isolados dos nodos da partição oposta. Na emulação deste tipo de falha é usada a arquitetura distribuída de FIONA. A configuração da falha é distribuída no início do experimento e o injetor principal notifica os demais a partir do envio de uma mensagem de ativação de falha. A falha é configurada como se segue:

```
UdpNetworkPartitioningFault:<início>:<fim>:<nodos_partição_1>:
<nodos_partição_2>:
```

Devido a este tipo de falha sempre ser ativada por tempo, o campo de tipo de ativação é suprimido. Desta forma, os campos de `início` e `fim` são definidos como o tempo em segundos que a falha será ativada e desativada. Os campos `nodos_partição_1` e `nodos_partição_2` definem quais nodos estarão em cada partição. Estes são especificados através de uma lista de nodos separados por vírgula (`,`). Para facilitar a configuração e evitar listar explicitamente o nome ou endereço de todos os hosts participantes,

FIONA permite o uso de máscaras. Por exemplo, se o primeiro campo de nodos for definido como `143.54.5.0/24` e o segundo campo definido como `143.54.12.0/24`, todos os nodos com endereço IP `143.54.5.*` não se comunicarão com os nodos com endereço `143.54.12.*`.

4.1.6 Interface Gráfica

Para facilitar a criação e edição de cenários de falhas, foi desenvolvida uma interface gráfica para FIONA. A interface possui uma aba para cada tipo de falha, que permite configurar os parâmetros específicos de cada falha. As falhas são adicionadas em uma caixa de texto, onde as falhas podem ser selecionadas e editadas, se necessário. O cenário configurado é gravado em um arquivo pelo usuário, que é passado como parâmetro na inicializaçao de FIONA. A versão atual da interface não permite a configuração de falhas de duplicação (CHIAO et al., 2004).

A figura 4.2 mostra a aba de configuração de falhas de omissão. Nesta aba podem ser preenchidos todos os parâmetros deste tipo de falha. A interface para configuração de falhas de colapso, duplicação e temporização são similares.

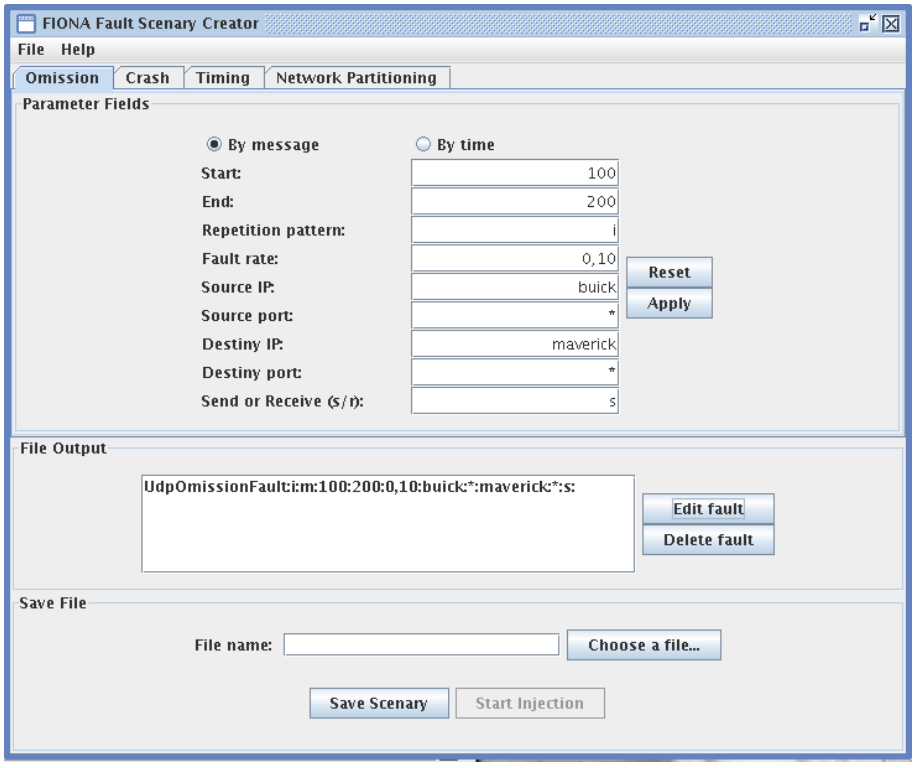

Figura 4.2: Interface gráfica para criação de falhas de omissão

A figura 4.3 mostra a aba de configuração de falhas de particionamento de rede. A interface usada se diferencia das demais (omissão, colapso e temporização), pois é necessário possibilitar a inserção dos endereços dos nodos. Para configurar, é necessário primeiramente adicionar todos os nodos relativos a uma partição (botão 'Add Host'). Depois de montada, o usuário adiciona a partição (botão 'Add Partition'). Após a adição da primeira partição, o usuário pode adicionar os nodos da outra partição. Ambas as partições podem ser posteriormente editadas ('Edit Partition') ou diretamente excluídas ('Remove Partition'). A interface permite a adição de mais partições e neste caso FIONA

interpreta a falha como o isolamento de todas as partições entre si. Por exemplo, se a falha é definida com três partições, as três ficarão isoladas entre si.

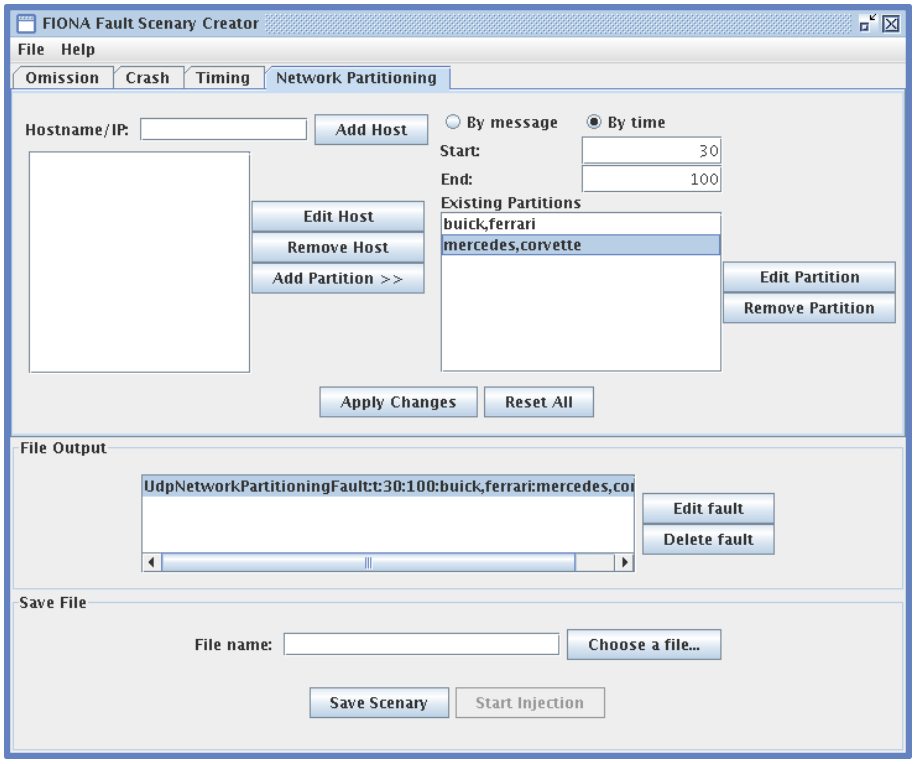

Figura 4.3: Interface gráfica para criação de falhas de particionamento

Parte comum em todas as abas é a mostra para o usuário da saída parcial do arquivo de configuração em 'File Output' e também a parte de salvamento de arquivo ('Save File'). O outro botão comum é para inicialização do experimento ('Start Injection'). Este botão apenas lança o processo de injeção quando FIONA é usado apenas em um nodo. A execução de FIONA de forma distribuída é descrita na próxima seção.

Para reusabilidade de cenários já configurados, a interface permite a reedição de arquivos. O usuário pode abrir um arquivo através do menu 'File' e da opção 'Open'. Assim, o conteúdo do arquivo é mostrado na caixa 'File Output', permitindo que o usuário selecione qual falha será editada. No momento em que o usuário clica no botão 'Edit fault', os parâmetros das falhas aparecem automaticamente na aba relativa a falha em questão. O botão 'Apply Changes' deve ser pressionado para aplicar a alteração realizada. Para posterior salvamento, o usuário escolhe o nome do arquivo (novo ou não) e clica em 'Save Scenary'.

4.2 Execução de FIONA

A execução da ferramenta FIONA depende primeiramente da máquina virtual instalada no nodo em que a aplicação será executada. FIONA usa como abordagem de injeção de falhas a interface nativa JVMTI, incorporada recentemente na versão 1.5 da máquina virtual Java (Java 5). Portanto, a premissa básica para possibilitar a execução com injeção de falhas é a instalação da versão 1.5 da JVM (*Java Virtual Machine*). Esta estará habilitada a carregar e executar agentes desenvolvidos com a interface JVMTI.

O disparo da ferramenta em si, depende de como esta será executada, ou seja, se esta executará apenas em um nodo ou então se executará de forma distribuída. Tal informação deve ser passada no disparo, que executará os procedimentos de acordo com o tipo de injetor (principal, de *site* ou local) que deve ser executado. A segunda informação a ser passada é o arquivo que descreve o cenário de falhas. Este é obrigatório apenas quando o injetor for utilizado localmente ou quando o injetor for configurado como o principal da arquitetura distribuída.

A linha de disparo da máquina virtual, indicando que é necessário o carregamento do agente JVMTI e os parâmetros para FIONA, é encapsulada através de um *script*. A figura 4.4 mostra como este *script* deve ser usado, considerando o disparo de cada tipo de injetor. O parâmetro `-file` indica a passagem do nome do arquivo de configuração, e é opcional no caso dos injetores de *site* e locais. O parâmetro `class` e `params` indica onde o usuário deve informar o nome da sua classe Java de disparo (classe que tem o método estático `main()`) e os parâmetros da própria aplicação. Quando o injetor é usado de forma distribuída, deve ser também indicado o seu tipo (`-main`, `-site` e `-local`). No caso de o injetor ser de *site* ou local, o endereço IP do injetor superior na hierarquia (`mainInjHostname` e `siteInjHostname`) deve ser informado.

```
- Injetor em nodo único:
   fiona -file {faultConfigFile} {class} {params}
- Injetor principal:
   fiona -main -file {faultConfigFile} {class} {params}
- Injetor de site:
   fiona -site  {mainInjHostname} [-file {faultConfigFile}]
                                        {class} {params}
- Injetor local:
   fiona -local {siteInjHostname} [-file {faultConfigFile}]
                                         {class} {params}
```

Figura 4.4: Linhas para execução de FIONA

4.3 Experimentos de Injeção de Falhas

Esta seção apresenta a condução de quatro experimentos com a ferramenta FIONA, mostrando sua viabilidade para realização de experimentos de injeção de falhas em aplicações distribuídas. Os três primeiros testes são conduzidos usando uma aplicação de quadro compartilhado (*shared whiteboard*), o que permite uma avaliação visual e rápida do funcionamento do injetor de falhas. O quarto teste apresenta uma breve avaliação de desempenho do sistema de comunicação de grupo JGroups em condição de falhas de omissão. Os experimentos 1 e 2 usaram FIONA em seu modo local, para execução em apenas um nodo, enquanto que os experimentos 3 e 4 usaram FIONA de forma distribuída. As subseções seguintes apresentam em detalhe os experimentos, mostrando os resultados obtidos.

4.3.1 Experimento 1

Como mencionado, a aplicação alvo para os três primeiros experimentos é similar e implementa um quadro compartilhado. Cada nodo tem um tela local em que o usuário pode fazer desenhos. Estes desenhos são então propagados, em tempo real, para as telas dos outros usuários. Os eventos de desenho são enviados como datagramas (pacotes UDP)

para os outros nodos, portanto mensagens perdidas resultam em desenhos incompletos. A aplicação necessita implementar, se os desenhos devem ser consistentes, mecanismos de detecção e retransmissão de pacotes perdidos. Esta aplicação é baseada em uma aplicação de exemplo do *middleware* JGroups, mas adaptada para utilizar exclusivamente UDP e para recuperar de falhas de omissão de mensagens. O objetivo deste experimento é a demonstração do uso de falhas de omissão com FIONA, além de evidenciar que as falhas configuradas são efetivamente injetadas pela ferramenta.

No primeiro teste, foi utilizado puramente UDP, sem considerar nenhum mecanismo de tolerância a falhas. FIONA foi usada em apenas um dos processos, injetando falhas de omissão de mensagens em um dos *links* de comunicação com uma taxa de 80%. A figura 4.5 ilustra que o injetor funciona como o esperado. A janela 'Draw 1' corresponde ao processo onde as falhas estão sendo injetadas e onde o desenho é feito. As falhas de omissão se manifestam no envio da mensagem, portanto o resultado destas falhas deve ser observado na janela que correspondente ao ponto de recepção do *link* faltoso. Neste caso, FIONA está injetando falhas no *link* entre os processos 1 e 2. A janela 'Draw 2' tem um desenho incompleto, pois as mensagens não foram recebidas como esperado e não há nenhum mecanismo de retransmissão de mensagens implementado. O *link* de comunicação com o processo associado a janela 'Draw 3' é livre de falhas, portanto o desendo desta janela é idêntico ao original.

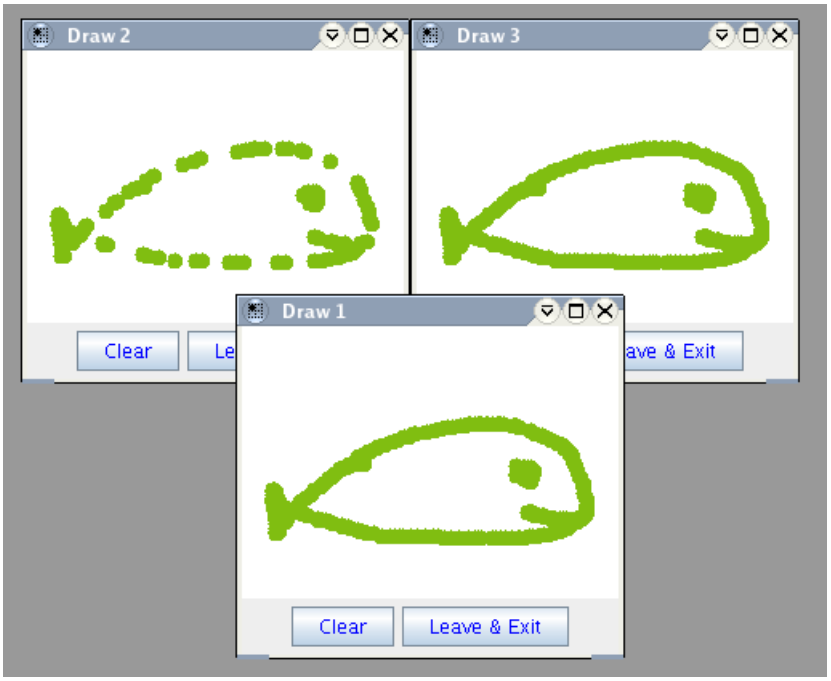

Figura 4.5: Experimento com falhas de omissão sem retransmissão

No teste seguinte, a mesma aplicação foi usada, porém agora com a implementação de um mecanismo de detecção de erros de transmissão e de recuperação de mensagens. Falhas de omissão de mensagens foram escolhidas como o tipo de falha para testar este mecanismo da aplicação de recuperação de pacotes perdidos. Quando um pacote UDP é perdido, é feita uma detecção baseada em *timeout* e assim o pacote é retransmitido. Para implementar o *timeout*, foi utilizado o método `setSoTimeout()` da classe `java.net.DatagramSocket`, o qual habilita o *timeout* do *socket* para expirar depois

de um intervalo especificado, neste caso, 100 milisegundos. Portanto, uma invocação do método `receive()` fica bloqueada até o tempo passado como parâmetro ao método `setSoTimeout()`. Se este *timeout* expirar, uma exceção indicando esta situação é lançada. Quando esta alcança a aplicação, esta exceção inicia as funções de recuperação de erro e a retransmissão da mensagem perdida. A retransmissão da mensagem também é confirmada pelo receptor, para evitar que mensagens retransmitidas também sejam perdidas.

A figura 4.6 mostra os resultados do segundo teste. Este foi conduzido de maneira similar ao primeiro: falhas sendo injetadas no processo representado pela janela 'Draw 1'. A comunicação faltosa foi estabelecida com o processo com a janela 'Draw 2', com uma taxa de falhas de omissão de 80%. Pode-se observar que, mesmo com as falhas sendo injetadas, o desenho na janela 'Draw 2' é idêntico aos outros dois. Isto mostra o comportamento esperado, ou seja, as falhas sendo mascaradas pelo mecanismo de tolerância a falhas. Neste caso, o mecanismo de detecção e recuperação de falhas teve uma cobertura de falhas de 100%.

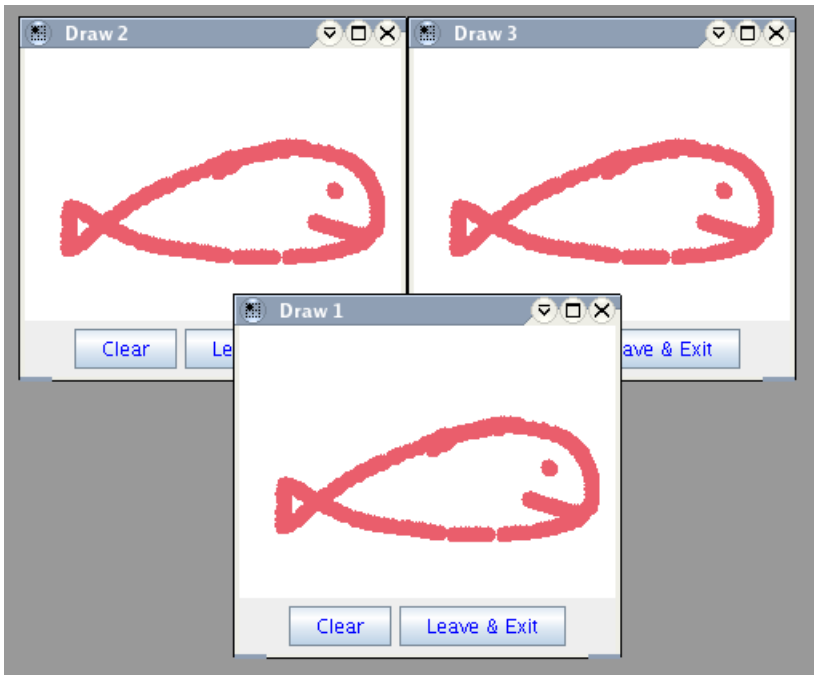

Figura 4.6: Experimento com falhas de omissão sem retransmissão

4.3.2 Experimento 2

Neste experimento, a aplicação sob teste usada é a mesma do primeiro teste descrito na subseção 4.3.1, ou seja, um quadro compartilhado que usa comunicação UDP sem mecanismos de tolerância a falhas implementados. A diferença na condução do teste é que neste caso somente dois processos são usados. Foram configuradas falhas de temporização entre 8 e 10 segundos no *link* de comunicação entre os dois processos, com uma taxa de falhas de 80%. Os atrasos escolhidos foram de valores altos para permitir a visualização clara nas figuras do efeito das falhas de temporização. Este experimento foi conduzido para mostrar a emulação correta de falhas de temporização com a ferramenta FIONA.

Os desenhos são feitos na tela do processo da janela ‘Draw 1’ e enviados para o processo da janela ‘Draw 2’. A figura 4.7 mostra os resultados obtidos. O processo 2 é representado duas vezes, para mostrar o mesmo processo em tempos diferentes da execução. A terceira imagem foi capturada alguns segundos depois da segunda imagem. Como pode ser visto, falhas de temporização causam reordenação de mensagens. Partes do desenho que foram feitas originalmente mais cedo, são apresentadas posteriormente. Esta aplicação não requer o uso de algoritmos de reoordenação, pois cada uma das mensagens têm coordenadas absolutas dos pontos do desenho.

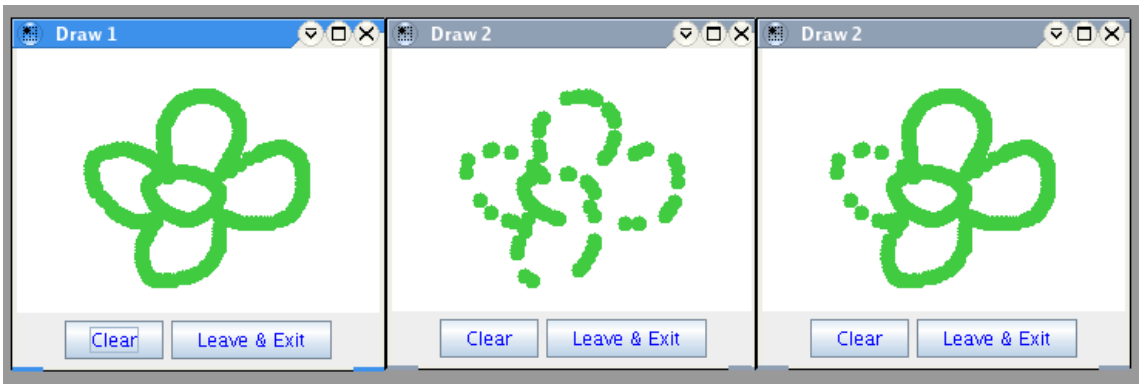

Figura 4.7: Experimento com falhas de temporização

4.3.3 Experimento 3

Para o terceiro experimento também foi utilizada uma aplicação de quadro compartilhado. A aplicação executada é diretamente o exemplo do *middleware* JGroups, que usa UDP *multiscast* para comunicação e as outras camadas do *middleware*, como retransmissão de mensagens e protocolo de *membership*. Por utilizar o protocolo de *membership*, sempre que um dos nodos do grupo falha, a visão do grupo é alterada para excluir o membro falho. Neste teste, a aplicação foi instanciada em quatro nodos (`corvette`, `maverick`, `buick` e `bentley`) e no teste foi usada uma falha de particionamento de rede, que isola dois nodos em uma partição e os outros dois em outra. A ferramenta FIONA foi usada de forma distribuída e configurada para ser injetor principal em `corvette` e injetor de *site* nos outros três nodos. O experimento visa demonstrar a emulação de falhas de particionamento. A escolha de uma aplicação que usa um protocolo de *membership* facilita a percepção do resultado do experimento, por implicar na formação de dois grupos independentes, sendo um em cada partição da rede.

Para sua execução, primeiramente todas as instâncias do programa se juntaram em um mesmo grupo, formado então por quatro membros. Durante esta fase, foi feito o desenho de uma letra em cada uma das máquinas: ‘A’ em `corvette`, ‘B’ em `maverick`, ‘C’ em `buick` e ‘D’ em `bentley`. Como todas as máquinas estavam se comunicando entre si, e portanto no mesmo grupo, todas elas receberam as mesmas mensagens, tendo um desenho consistente nas quatro janelas.

Posteriormente, a falha de particionamento foi ativada. A falha foi descrita para ser ativada em 50 segundos após o início da execução e para dividir a rede em duas partições: (i) `corvette` e `maverick`, e (ii) `buick` e `bentley`. Portanto, após 50 segundos, a máquina instanciada como injetor principal (`corvette`), enviou uma mensagem para ativação da falha de particionamento. Após a ativação da falha foram desenhadas novas

letras: ‘E’ em `corvette`, ‘F’ em `maverick`, ‘G’ em `buick` e ‘H’ em `bentley`. A figura 4.8 mostra que a ativação da falha provocou uma troca de visão nos nodos de ambas as partições. Cada nodo atualizou sua visão para conter apenas dois membros, excluindo os membros da partição oposta. É visto na figura que desenhos feitos em uma partição só foram propagados para nodos da mesma partição e não da partição oposta. Com este tipo de falha podem ser testados, por exemplo, mecanismos de tolerância a falhas que prevêem o reagrupamento de partições.

Figura 4.8: Experimento com falhas de particionamento de rede

4.3.4 Experimento 4

Este experimento mostra como FIONA foi aplicado na condução de um experimento de injeção de falhas com o *middleware* JGroups, comumente usado como sistema de comunicação de grupo. O teste consiste em avaliar seu desempenho em condição de falhas de omissão. O objetivo não é mostrar um experimento completo de avaliação de dependabilidade com JGroups ou analisar seu desempenho em várias situações de falha, mas sim ilustrar como FIONA pode ser usado para testes de dependabilidade e de degradação de desempenho em um sistema distribuído. Como carga de trabalho para o teste do desempenho de JGroups, foi usado o mesmo *benchmark* descrito em Abdellatif, Cecchet e Lachaize (2004). A diferença principal entre o experimento aqui descrito e o original é que este último não considera falhas de comunicação em sua avaliação de desempenho.

O *benchmark* realiza uma comunicação *um para muitos*, onde um nodo mestre envia mensagens para todos os membros de um determinado gurpo, inclusive para ele mesmo. *Multicast* UDP é usado para transmissão, e o *middleware* em si garante que todas as mensagens enviadas são entregues para todos os membros do grupo, usando *unicast* UDP para retransmisão de mensagens perdidas. Cada mensagem tem o tamanho de 1 000 bytes. Para cada execução, o mestre envia 10 000 mensagens. Os experimentos foram conduzidos em quatro máquinas AMD Athlon XP 2000+, com 512 MB de memória RAM, executando o sistema operacional Linux com *kernel* versão 2.6.5 e conectados por um chaveador (*switch*) em uma rede Ethernet de 100 Mbps. A versão usada do JGroups é 2.2.7, executando sobre a JVM da Sun em sua versão 1.5.

Os dois cenários de falhas considerados foram: (A) falhas de omissão no envio injetadas no nodo mestre, e (B) falhas de omissão de recepção injetadas em dois membros do grupo. Em ambos experimentos, FIONA executou em três nodos: como *injetor principal* no nodo mestre e como *injetor de site* nos outros dois nodos. Para fazer uma comparação justa, os nodos falhos são configurados com taxas de falhas de 2%, 4%, 8% e 16%, enquanto que nodos livres de falha onde o injetor também é executado são configurados com uma taxa de falha de 0%. Isto garante que, em ambos os casos, os mesmos métodos ins-

trumentados são usados para transmissão e recepção de mensagens, evitando diferenças temporais criadas pela intrusividade intrínsica do injetor de falhas.

A queda de desempenho foi medida através da vazão (*throughput*), que é expressa em número de mensagens por segundo. Em cada receptor, o número total de mensagens recebidas foi dividido pelo tempo decorrido entre a recepção da primeira e da última mensagem. A vazão média para os quatro membros do grupo foi então calculado. Cada experimento foi repetido cinco vezes, e a média da degradação de desempenho, para cada taxa de falhas, é mostrado na figura 4.9. A porcentagem indicada é relativa a vazão máxima, sem falhas sendo injetadas. Os experimentos conduzidos para obtenção da média não apresentaram um desvio padrão significativo.

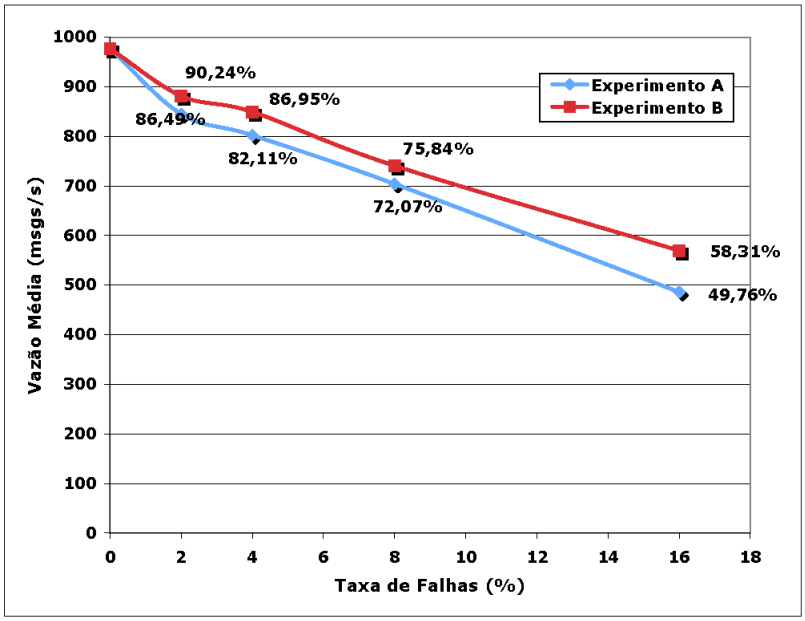

Figura 4.9: Queda de desempenho do JGroups com falhas de omissão

Em ambos os cenários, a medida que a taxa de falhas aumenta, a vazão do JGroups é reduzida. A redução é ainda mais significativa no experimento A. Quando falhas de omissão de envio são injetadas, as falhas são detectadas por todos os membros do grupo, portanto a retransmissão deve ocorrer para todos os quatro membros. No experimento B, a diminuição da vazão não é tão expressiva quanto no primeiro experimento, pois as falhas são detectadas por somente dois membros, forçando um número menor de operações de retransmissão e implicando em uma menor queda de desempenho quando calculada a vazão média. Esta é uma situação onde fica evidente a vantagem de FIONA permitir injeção de falhas de omissão tanto no envio como na recepção.

A camada de retransmissão utilizada pelo *bechmark* é uma das disponíveis pelo próprio *middleware*. A camada escolhida (`org.jgroups.protocols.NAKACK`) implementa a técnica de confirmações negativas (*negative acknoledgments*), pareadas com confirmações positivas (*positive acknoledgments*). Para estes experimentos, a cobertura de falhas deste método foi de 100%, pois todas as mensagens enviadas foram recebidas por todos os membros do grupo.

4.4 Conclusão do Capítulo

Este capítulo mostrou como podem ser criados cenários de falhas para a ferramenta FIONA, mostrando os possíveis parâmetros para cada uma delas. Também é mostrado como a interface desenvolvida pode auxiliar na criação destes cenários. Posteriormente é descrito como FIONA pode ser executada, tanto para experimentos em que esta é instanciada em apenas um nodo, como para aqueles em que FIONA é usada de forma distribuída.

Para mostrar a aplicabilidade de FIONA na condução de experimentos de injeção de falhas, foram também mostrados quatro testes, usando tanto FIONA local como distribuído. Os cenários de falhas utilizados nestes casos foram simples. Porém, cenários mais complexos, com múltiplas falhas e de vários tipos, podem ser criados para um avaliação de dependabilidade real de uma determinada aplicação.

5 CONSIDERAÇÕES FINAIS

Quando sistemas de alta disponibilidade são desenvolvidos, estes devem ser testados antes de ser colocados em um ambiente de produção. O teste deve incluir a observação do comportamento sob falhas, ou seja, deve-se testar os mecanismos de tolerância a falhas implementados do sistema verificando seu funcionamento. Considerando sistemas de larga escala usados para a execução de aplicações de longa duração, não se pode correr o risco de perder toda a computação realizada devido a um comportamento inadequado dos mecanismos aplicados para o tratamento das falhas. Com isso, a implementação de mecanismos de tolerância a falhas e seus respectivos testes é mandatória.

Falhas são fenômenos aleatórios e de baixa probabilidade de ocorrência. Portanto, para a realização do teste é necessária que esta ocorrência seja acelerada, evitando que se espere pela ocorrência por uma falha real. Para isso, uma técnica bastante usada é a realização de experimentos de injeção de falhas, emulando a ocorrência de situações faltosas para a aplicação. Neste trabalho, foi apresentada a ferramenta FIONA que implementa esta técnica em software, injetando falhas de comunicação para validação de sistemas largamente distribuídos. Falhas de comunicação afetam o sistema de troca de mensagens dos nodos sob avaliação e permitem o teste dos mecanismos de detecção e correção de falhas em sistemas distribuídos.

As ferramentas existentes para validação de sistemas distribuídos enfrentam limitações ao serem diretamente transpostas para sistemas de larga escala, como a dificuldade de configuração de cenários de múltiplas falhas e o gerenciamento do experimento em si, pois cada instância do injetor é independente em cada um dos nodos do sistema. FIONA visa superar estes problemas através das seguintes características:

- possuir uma arquitetura distribuída e escalável, baseada em arquiteturas de monitoramento para sistemas de larga escala (*grids*);
- permitir a emulação direta de um modelo de falhas condizente com este tipo de sistema, incluindo particionamento de redes;
- possibilitar a configuração centralizada do sistema e também a construção de um *log* global do experimento de injeção de falhas.

A ferramenta FIONA pode ser usada para dois propósitos distintos: auxiliar no desenvolvimento de software, e obtenção de medidas de dependabilidade. Ferramentas deste tipo são importantes da fase de teste do software, na depuração dos mecanismos de tolerância a falhas desenvolvidos. O desenvolvedor pode testar sua aplicação com o injetor até esta apresentar um comportamento de acordo com a sua especificação.

Ferramentas de injeção de falhas são essenciais também para a obtenção de medidas de dependabilidade, pois estas são obtidas quando o software é exposto a situações faltosas. A obtenção de métricas é importante para duas situações: assegurar que o software desenvolvido obedece a determinadas exigências de funcionamento, mesmo em condição de falhas; e permitir a avaliação de ferramentas desenvolvidas externamente, como componentes de software ou *middleware*, verificando se estas são adequadas para um determinado projeto, ou até mesmo comparando componentes similares. Este é um dos objetivos dos *benchmarks* de dependabilidade, área de bastante pesquisa atualmente. FIONA permite a avaliação de componentes externos devido à sua baixa intrusividade espacial, que não exige que as aplicações sob teste tenham seu código fonte disponível.

A baixa intrusividade espacial de FIONA se deve a esta usar uma nova abordagem para injeção de falhas em programas Java: o uso de uma interface nativa de programação disponibilizada pela máquina virtual. A interface nativa JVMTI, originalmente desenvolvida para construção de ferramentas de monitoramento e depuração, foi utilizada em FIONA para inserir código de injeção de falhas nas aplicações sob teste. O uso desta interface permite que o código da aplicação alvo não precise ser alterado para conduzir os testes de injeção de falhas. Além disso, a utilização de um agente JVMTI é simples, pois é necessário apenas indicar um parâmetro no lançamento da máquina virtual.

FIONA teve seu protótipo desenvolvido, com todo seu modelo de falhas implementado. O protótipo se mostrou eficiente na emulação das falhas do modelo proposto e na condução de experimentos, tanto como um injetor de falhas utilizado em um único nodo, como quando usado de forma distribuída.

5.1 Trabalhos Futuros

Como trabalhos futuros são propostos:

- a extensão de FIONA para outros protocolos de comunicação, como TCP, RMI e SOAP. Isto aumentaria o número de aplicações que poderiam usar FIONA como ferramenta de injeção de falhas.
- a utilização de FIONA para validar experimentalmente o sistema de comunicação de grupo JGroups;
- a condução de experimentos reais de injeção de falhas, diferente dos exemplificados no capítulo 4. O ambiente de execução da aplicação alvo deve ser um sistema distribuído de larga escala;
- análise da execução de FIONA com ferramentas de rastreamento da programas, para identificar gargalos de desempenho e minimizar sua intrusividade temporal;
- o desenvolvimento de um conversor de regras de injeção de falhas de FIONA para a ferramenta ComFIRM e a implementação de um módulo de comunicação entre FIONA e ComFIRM, para permitir que este último funcione como um *injetor local* da arquitetura distribuída. Desta forma, FIONA poderá injetar falhas não apenas em aplicações Java, mas também sobre qualquer aplicação que execute sobre o sistema Linux;
- inclusão de outras distribuições estatísticas, diferente da distribuição uniforme, e possibilitar a descrição do acontecimento de falhas em rajadas;

- desenvolvimento de mecanismos para monitoramento e coleta de dados das aplicações sob teste, para facilitar a obtenção de medidas de dependabilidade.

Dentre estes, já se encontram em fase de desenvolvimento a expansão de FIONA para o protocolo TCP, a análise da execução da ferramenta, o conversor de regras para ComFIRM e a inclusão de outras distribuições estatísticas.

REFERÊNCIAS

ABDELLATIF, T.; CECCHET, E.; LACHAIZE, R. Evaluation of a group communication middleware for clustered J2EE application servers. In: COOPIS/DOA/ODBASE, 2004, Agia Napa, Cyprus. **Proceedings...** [S.l.]: Springer, 2004. p.1571–1589. (Lecture Notes in Computer Science, v.3291).

AIDEMARK, J.; VINTER, J.; FOLKESSON, P.; KARLSSON, J. GOOFI: generic object-oriented fault injection tool. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 2001, Gotebörg, Sweden. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2001. p.83–88.

AMENDOLA, A. M.; IMPAGLIAZZO, L.; MARMO, P.; POLI, F. Experimental Evaluation of Computer-Based Railway Control Systems. In: FAULT-TOLERANT COMPUTING SYMPOSIUM, 1997, Seattle, Washington. **Proceedings...** [S.l.: s.n.], 1997. p.380–384.

ARLAT, J.; AGUERA, M.; AMAT, L.; CROUZET, Y.; FABRE, J.-C.; LAPRIE, J.-C.; MARTINS, E.; POWELL, D. Fault Injection for Dependability Validation: a methodology and some applications. **IEEE Transactions on Software Engeneering**, [S.l.], v.16, n.2, p.166–182, 1990.

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. E. Basic Concepts and Taxonomy of Dependable and Secure Computing. **IEEE Transactions on Dependable and Secure Computing**, [S.l.], v.1, n.1, p.11–33, 2004.

BALATON, Z.; KACSUK, P.; PODHORSZKI, N.; VAJDA, F. From Cluster Monitoring to Grid Monitoring Based on GRM. In: EUROPEAN CONFERENCE ON PARALLEL COMPUTING, 2001, Manchester, UK. **Proceedings...** [S.l.]: Springer, 2001. p.874–881. (Lecture Notes in Computer Science, v.2150).

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

BIRMAN, K. P. **Building Secure and Reliable Network Applications**. Greenwich: Manning Pub., Co, 1996.

CARREIRA, J.; MADEIRA, H.; SILVA, J. G. Assessing the Effects of Communication Faults on Parallel Applications. In: INTERNATIONAL COMPUTER PERFORMANCE AND DEPENDABILITY SYMPOSIUM, 1995, Erlangen, Germany. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1995. p.214–223.

CARREIRA, J.; SILVA, J. G. Why do Some (weird) People Inject Faults? **ACM SIGSOFT Software Engineering Notes**, New York, NY, USA, v.23, n.1, p.42–43, Jan. 1998.

CHANDRA, R.; LEFEVER, R. M.; JOSHI, K. R.; CUKIER, M.; SANDERS, W. H. A Global-State-Triggered Fault Injector for Distributed System Evaluation. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.15, n.7, p.593–605, July 2004.

CHIAO, C. M.; TRINDADE, J. M. F. da; JACQUES-SILVA, G.; JANSCH-PÔRTO, I.; WEBER, T. S. Uma Interface Gráfica para a Criação de Cenários de Falhas para a Ferramenta FIONA. In: LIVRO DE RESUMOS DO SALÃO DE INICIAÇÃO CIENTÍFICA DA UFRGS, 2004. **Anais...** [S.l.: s.n.], 2004. p.46.

CHIBA, S. Load-time Structural Reflection in Java. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 2000, Cannes, France. **Proceedings...** [S.l.: s.n.], 2000. p.313–336.

CHOI, G. S.; IYER, R. K. FOCUS: an experimental environment for fault sensitivity analysis. **IEEE Transactions on Computers**, [S.l.], v.41, n.12, p.1515–1526, 1992.

CONSTANTINESCU, C. Validation of the Fault/Error Handling Mechanisms of the Teraflops Supercomputer. In: FAULT-TOLERANT COMPUTING SYMPOSIUM, 1998, Munich, Germany. **Proceedings...** [S.l.: s.n.], 1998. p.382–389.

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

CUKIER, M.; CHANDRA, R.; HENKE, D.; PISTOLE, J.; SANDERS, W. H. Fault Injection Based on a Partial View of the Global State of a Distributed System. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 18., 1999, Lausanne, Switzerland. **Proceedings...** Los Alamitos: IEEE Computer Society, 1999. p.168–177.

DAHM, M. **Byte Code Engineering with the BCEL API**. [S.l.]: Freie Universität at Berlin, Institut für Informatik, 2001. (B-17-98).

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

DREBES, R. J.; LEITE, F. O.; JACQUES-SILVA, G.; MOBUS, F.; WEBER, T. S. ComFIRM: a communication fault injector for protocol testing and validation. In: DIGEST OF PAPERS OF IEEE LATIN-AMERICAN TEST WORKSHOP, 2005, Salvador, Brazil. **Proceedings...** [S.l.: s.n.], 2005. p.152–157.

FOSTER, I.; KESSELMAN, C.; TUECKE, S. The Anatomy of the Grid: enabling scalable virtual organization. **The International Journal of High Performance Computing Applications**, [S.l.], v.15, n.3, p.200–222, Fall 2001.

GERCHMAN, J. **FIONA TCP**: extensão TCP do injetor de falhas FIONA. Porto Alegre: Universidade Federal do Rio Grande do Sul, Instituto de Informática, 2004.

GONÇALVES, L. C. R. **Injetor de Falhas por Depuração**. 2002. 134p. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

GONÇALVES, L. C. R.; RODEGHERI, P. R.; MANFREDINI, R. A.; WEBER, T. S. Testing Fault Tolerance Mechanisms in DBMS Through Fault Injection. In: DIGEST OF PAPERS OF LATIN-AMERICAN TEST WORKSHOP, 2001, Cancún, México. **Proceedings...** [S.l.: s.n.], 2001. p.278–284.

GOSWAMI, K. K.; IYER, R. K.; YOUNG, L. DEPEND: a simulation-based environment for system level dependability analysis. **IEEE Transactions on Computers**, [S.l.], v.46, n.1, p.60–74, 1997.

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

HSUEH, M.-C.; TSAI, T.; IYER, R. Fault Injection Techniques and Tools. **IEEE Computer**, [S.l.], v.30, n.4, p.75–82, April 1997.

HUTCHINSON, N. C.; PETERSON, L. L. The x-Kernel: an architecture for implementing network protocols. **IEEE Transactions on Software Engineering**, Washington, DC, v.17, n.1, p.64–76, 1991.

IYER, R. K. Experimental Evaluation. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 1995, Pasadena, California. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 1995. p.115–132.

IYER, R. K.; TANG, D. Experimental Analysis of Computer System Dependability. In: PRADHAN, D. K. (Ed.). **Fault-Tolerant Computer System Design**. New York: Prentice-Hall, Inc., 1996. p.282–392.

J. R. SAMSON, J.; MORENO, W.; FALQUEZ, F. A Technique for Automated Validation of Fault Tolerant Designs Using Laser Fault Injection (LFI). In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 1998, Munich, Germany. **Proceedings...** Los Alamitos: IEEE Computer Society, 1998. p.162–167.

JACQUES-SILVA, G.; MORAES, R. L. O.; WEBER, T. S.; MARTINS, E. Validando Sistemas Distribuídos Desenvolvidos em Java Utilizando Injeção de Falhas de Comunicação por Software. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, 5., 2004, Gramado. **Anais...** Porto Alegre: II/UFRGS, 2004. p.53–64.

JENN, E.; ARLAT, J.; RIMÉN, M.; OHLSSON, J.; KARLSSON, J. Fault Injection into VHDL Models: the mefisto tool. In: FAULT-TOLERANT COMPUTING SYMPOSIUM, 1994, Austin, Texas. **Proceedings...** [S.l.: s.n.], 1994. p.66–75.

KANAWATI, N. A.; KANAWATI, G. A.; ABRAHAM, J. A. Dependability evaluation using hybrid fault/error injection. In: INTERNATIONAL COMPUTER PERFORMANCE AND DEPENDABILITY SYMPOSIUM, 1995, Erlangen, Germany. **Proceedings...** Los Alamitos: IEEE Computer Society, 1995. p.224–233.

KANOUN, K.; MADEIRA, H.; ARLAT, J. A Framework for Dependability Benchmarking. In: WORKSHOP ON DEPENDABILITY BENCHMARKING, 2002. **Proceedings...** [S.l.: s.n.], 2002. p.F7–F8.

KAO, W.-L.; IYER, R. K. DEFINE: a distributed fault injection and monitoring environment. In: IEEE WORKSHOP ON FAULT-TOLERANT PARALLEL AND DISTRIBUTED SYSTEMS, 1994, College Station, USA. **Proceedings...** [S.l.: s.n.], 1994. p.252–259.

KARLSSON, J.; LIDEN, P.; DAHLGREN, P.; JOHANSSON, R.; GUNNEFLO, U. Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms. **IEEE Micro**, [S.l.], v.14, n.1, p.8–11, 13–23, 1994.

KICKZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.-M.; IRWIN, J. Aspect-Oriented Programming. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 1997, Jyväskylä, Finland. **Proceedings...** [S.l.: s.n.], 1997.

LEITE, F. O. **ComFIRM**. 2000. 117p. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

LEME, N. G. M. **Um Sistema de Padrões para Injeção de Falhas por Software**. 2001. 88p. Dissertação (Mestrado em Ciência da Computação) — Instituto de Computação, UNICAMP, Campinas.

LEME, N. G. M.; MARTINS, E.; RUBIRA, C. M. F. A Software Fault Injection Pattern System. In: CONFERENCE ON PATTERN LANGUAGES OF PROGRAMS, 8., 2001, Monticello, USA. **Proceedings...** [S.l.: s.n.], 2001.

LOOKER, N.; XU, J. Assessing the Dependability of OGSA Middleware by Fault Injection. In: SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 22., 2003, Florence, Italy. **Proceedings...** [S.l.: s.n.], 2003. p.293–302.

MAES, P. Concepts and Experiments in Computational Reflection. In: OOPSLA-87: CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS, 1987, Orlando, Florida. **Proceedings...** [S.l.: s.n.], 1987. p.147–155.

MAILLET, E.; TRON, C. On Efficiently Implementing Global Time for Performance Evaluation on Multiprocessor Systems. **Journal of Parallel and Distributed Computing**, [S.l.], v.28, n.1, p.84–93, July 1995.

MARTINS, E.; ROSA, A. C. A. A Fault Injection Approach Based on Reflective Programming. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, 2000, New York, New York. **Proceedings...** Los Alamitos: IEEE Computer Society Press, 2000. p.407–416.

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

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

O'HAIR, K. **The JVMPI Transition to JVMTI**. Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/jvmpitransition>. Acesso em 15 de agosto de 2004.

OPPENHEIMER, D.; BROWN, A. B.; TRAUPMAN, J.; BROADWELL, P.; PATTERSON, D. A. Practical Issues in Dependability Benchmarking. In: WORKSHOP ON EVALUATING AND ARCHITECTING SYSTEM DEPENDABILITY, 2., 2002. **Proceedings...** [S.l.: s.n.], 2002.

SCHNEIDER, F. B. What Good are Models and What Models are Good? In: MULLENDER, S. (Ed.). **Distributed Systems**. 2nd ed. Workingham: Addison-Wesley, 1993. p.17–26.

SIEWIOREK, D. P.; CHILLAREGE, R.; KALBARCZYK, Z. T. Reflections on Industry Trends and Experimental Research in Dependability. **IEEE Transactions on Dependable and Secure Computing**, [S.l.], v.1, n.2, p.109–127, 2004.

SILVEIRA, K. K.; WEBER, T. S. Uma Arquitetura de Injetor de Falhas Orientada a Aspectos para Validação de Sistemas de Comunicação de Grupo. In: ESCOLA REGIONAL DE REDES DE COMPUTADORES, 2., 2004, Canoas, RS. **Anais...** [S.l.: s.n.], 2004. p.35–40.

STOTT, D. T.; FLOERING, B.; BURKE, D.; KALBARCZYK, Z.; IYER, R. K. NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. In: IEEE INTERNATIONAL PERFORMANCE AND DEPENDABILITY SYMPOSIUM, 2000, Chicago, USA. **Proceedings...** [S.l.: s.n.], 2000. p.91–100.

STOTT, D. T.; RIES, G.; HSUEH, M.-C.; IYER, R. K. Dependability Analysis of a High-Speed Network Using Software-Implemented Fault Injection and Simulated Fault Injection. **IEEE Transactions on Computers**, [S.l.], v.47, n.1, p.108–119, 1998.

SUN MICROSYSTEMS. **Java Virtual Machine Profiler Interface**. Disponível em: <http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html>. Acesso em: 15 de julho 2004.

SUN MICROSYSTEMS. **Java Remote Method Invocation**. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide/rmi/>. Acesso em: 15 de janeiro de 2005.

SUN MICROSYSTEMS. **Java Virtual Machine Tool Interface**. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/>. Acesso em: 15 de julho de 2004.

SUN MICROSYSTEMS. **Java Virtual Machine Debug Interface**. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jvmdi-spec.html >. Acesso em: 15 de julho de 2004.

TANENBAUM, A. S.; STEEN, M. V. **Distributed Systems**: principles and paradigms. [S.l.]: Prentice Hall PTR, 2002.

VARDANEGA, T.; DAVID, P.; CHANE, J.-F.; MADER, W.; MESSAROS, R.; ARLAT, J. On the Development of Fault-Tolerant On-Board Control Software and its Evaluation by Fault Injection. In: FAULT-TOLERANT COMPUTING SYMPOSIUM, 1995, Pasadena, California. **Proceedings...** [S.l.: s.n.], 1995. p.510–515.

VISWANATHAN, D.; LIANG, S. Java Virtual Machine Profiler Interface. **IBM Systems Journal**, [S.l.], v.39, n.1, p.82–95, 2000.

VOAS, J. Software Fault Injection: growing 'safer' systems. In: IEEE AEROSPACE CONFERENCE, 1997, Manhattan Beach, California. **Proceedings...** [S.l.: s.n.], 1997. v.2, p.551–562.

WIESMANN, M.; DÉFAGO, X.; SCHIPER, A. Group Communication based on Standard Interfaces. In: IEEE INTERNATIONAL SYMPOSIUM ON NETWORK COMPUTING AND APPLICATIONS, 2003, Cambridge, Massachusetts. **Proceedings...** [S.l.: s.n.], 2003.

APÊNDICE A HISTÓRICO DE DESENVOLVIMENTO

Nos últimos anos, o Grupo de Tolerância a Falhas tem desenvolvido trabalhos na área de injeção de falhas por *software*. Dentre os mais recentes estão: FIDe, INFIMO e ComFIRM. Todas estas ferramentas são capazes de injetar falhas de comunicação, porém, devido à abordagem utilizada, cada uma apresenta suas limitações. FIDe, por interceptar as aplicações através da chama `ptrace()`, tem uma alta intrusividade temporal, penalizando consideravelmente o desempenho da aplicação sob teste. INFIMO é um *toolkit* para comparar a intrusividade de métodos de injeção de falhas. Uma de suas abordagens é a alteração de uma biblioteca de comunicação. Neste caso, a ferramenta ficou vinculada apenas a esta biblioteca instrumentada. Em INFIMO, houve uma contribuição no aprofundamento do modelo de falhas de comunicação utilizado, descrito em [Barcelos et al 2004]. ComFIRM, em sua versão original, requer que a ferramenta seja alterada a cada versão do sistema operacional. Atualmente ComFIRM opera como um módulo no sistema [Drebes et al 2005], mas ainda é vinculado ao sistema operacional Linux.

A partir disso, começou-se a procurar alternativas a estas abordagens, iniciando pela pesquisa de outros injetores de falhas, como GOOFI e Jaca, para expandi-los e adaptar para a validação de sistemas distribuídos. GOOFI, desenvolvido na Universidade de Tecnologia de Chalmers, foi a primeira ferramenta a ser estudada para expansão [Jacques-Silva e Weber 2003], porém seu código-fonte não foi disponibilizado pelos desenvolvedores. A segunda alternativa estudada foi a expansão de Jaca, chamada Jaca.net. A expansão foi abandonada por apresentar, no caso de falhas de comunicação, uma intrusividade espacial na aplicação sob teste. Este trabalho gerou um artigo publicado em evento nacional [Jacques-Silva et al 2004a], um em evento internacional [Jacques-Silva et al 2005] e outro artigo submetido para uma revista internacional [Jacques-Silva et al 2004c].

Como alternativa, usou-se uma interface nativa da máquina virtual Java para desenvolvimento de ferramentas de monitoramento e depuração, chamada JVMTI. O uso de interfaces nativas foi motivado pelo conhecimento prévio destas interfaces para o desenvolvimento de um agente de geração de rastros para aplicações Java [Jacques-Silva, Schnorr e Stein 2003]. A primeira fase do trabalho, portanto, foi o desenvolvimento do injetor de falhas FIONA baseado na interface JVMTI, explorando suas facilidades para injeção de falhas de comunicação em aplicações desenvolvidas usando o protocolo UDP. Estes resultados foram publicados em um evento regional [Gerchman, Jacques-Silva e Weber 2004] e um evento internacional [Jacques-Silva et al 2004b].

Depois de definida a técnica para injeção de falhas da ferramenta, definiu-se a arquitetura de distribuição e a abordagem para injeção distribuída de falhas. A adaptação da ferramenta para aplicações de larga escala se deu devido ao interesse do grupo de pesquisa em técnicas de tolerância a falhas e validação em ambientes de computação em *grid*. As

características incluídas em FIONA, como a distribuição de configurações e possibilidade de injeção direta de falhas de particionamento de rede, a distingue não apenas das ferramentas anteriores do grupo como também de outros injetores existentes. Trabalhos sobre a versão distribuída de FIONA foram submetidos para um evento nacional [Gerchman et al 2005] e para um evento internacional [Jacques-Silva et al 2005].

Outro trabalho em desenvolvimento do grupo é a integração de FIONA e a nova versão de ComFIRM. As aplicações alvo de FIONA são desenvolvidas em Java e as de ComFIRM são quaisquer aplicações executando sobre Linux. Para permitir a execução em um ambiente heterogêneo, está em desenvolvimento um conversor de regras de falhas e um módulo de comunicação entre nodos ComFIRM e FIONA, permitindo sua interação e a coordenação de experimentos com nodos ComFIRM. Sobre este trabalho foi publicado um artigo em um evento nacional [Mobus et al 2005].

Artigos Publicados

- G. Jacques-Silva, L. M. Schnorr, B. O. Stein, *JRastro: A Trace Agent for Debugging Multithreaded and Distributed Java Programs*. Proceedings of the Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2003). p. 46-54. São Paulo.

- G. Jacques-Silva, T. S. Weber, *Uma Estratégia para Validação Experimental de um Sistema de Comunicação de Grupo*. Anais da I Escola Regional de Redes de Computadores (ERRC 2003). p. 146-151. Porto Alegre.

- P. P. de A. Barcelos, R. J. Drebes, G. Jacques-Silva, T. S. Weber, *A Toolkit to Test the Intrusion of Fault Injection Methods*. Digest of Papers of Latin-American Test Workshop (LATW 2004). p. 152-157. Cartagena, Colombia.

- G. Jacques-Silva, R. L. de O. Moraes, T. S. Weber, E. Martins, *Validando Sistemas Distribuídos Desenvolvidos em Java Utilizando Injeção de Falhas de Comunicação por Software*. Anais do V Workshop de Testes e Tolerância a Falhas (WTF 2004). p. 53-64. Gramado.

- J. Gerchman, G. Jacques-Silva, T. S. Weber, *Implementação de um Injetor de Falhas de Comunicação para Aplicações Java Baseado em JVMTI*. Anais da II Escola Regional de Redes de Computadores (ERRC 2004). p. 23-28. Canoas.

- G. Jacques-Silva, R. J. Drebes, J. Gerchman, T. S. Weber, *FIONA: A Fault Injector for Dependability Evaluation of Java-Based Network Applications*. Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (NCA 2004). p. 303-308. Cambridge, United States.

- G. Jacques-Silva, R. J. Drebes, T. S. Weber, E. Martins, *Injecting Communication Faults to Experimentally Validate Java Distributed Applications*. Proceedings of the International School & Symposium on Advanced Distributed Systems (ISSADS 2005). Guadalajara, Mexico. Lecture Notes in Computer Science, v. 3563, p. 235-245.

- R. J. Drebes, F. O. Leite, G. Jacques-Silva, F. Mobus, T. S. Weber, *ComFIRM: a Communication Fault Injector for Protocol Testing and Validation*. Digest of Papers of Latin-American Test Workshop (LATW 2005). p. 115-120. Salvador.

- F. Mobus, R. J. Drebes, G. Jacques-Silva, T. S. Weber, I. Jansch-Pôrto, *Estendendo FIONA para uma arquitetura híbrida*. Anais do VI Workshop de Testes e Tolerância a Falhas (WTF 2005). p. 125-128. Fortaleza.

- J. Gerchman, G. Jacques-Silva, R. J. Drebes, T. S. Weber, *Ambiente Distribuído de Injeção de Falhas de Comunicação para Teste de Aplicações Java de Rede*. Aceito para publicação no 19º Simpósio Brasileiro de Engenharia de Software (SBES 2005).

Artigos em Avaliação

- G. Jacques-Silva, R. J. Drebes, T. S. Weber, E. Martins, *Integrating Network Fault Scenarios in a Dependability Evaluation Tool to Validate Java-based Distributed Applications*. Submetido ao Journal of Systems and Software em Dezembro de 2004.

- G. Jacques-Silva, R. J. Drebes, J. Gerchman, T. S. Weber, I. Jansch-Pôrto, *A Network-level Distributed Fault Injector for Experimental Validation of Dependable Distributed Systems*. Submetido ao Symposium on Reliable Distributed Systems (SRDS 2005).

- J. Vacaro, G. Jacques-Silva, T. S. Weber, I. Jansch-Pôrto, *Distribuições de Probabilidade na Descrição de Carga de Falhas para Injetores de Falhas de Comunicação*. Submetido a III Escola Regional de Redes de Computadores (ERRC 2005).

- J. M. F. Trindade, G. Jacques-Silva, T. S. Weber, *Geração de logs de Experimentos de Injeção de Falhas para Análise de Dependabilidade de Aplicações Distribuídas*. Submetido a III Escola Regional de Redes de Computadores (ERRC 2005).