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

GUSTAVO MENEZES OLIVEIRA

Injeção de Falhas de Comunicação em Ambientes Distribuídos

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

Profa. Dra. Taisy Silva Weber
Orientador

Porto Alegre, fevereiro de 2011

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Oliveira, Gustavo Menezes

Injeção de Falhas de Comunicação em Ambientes Distribuídos / Gustavo Menezes Oliveira. – Porto Alegre: PPGC da UFRGS, 2011.

76 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2011. Orientador: Taisy Silva Weber.

1. Injeção de falhas de comunicação. 2. Particionamento de rede. I. Weber, Taisy Silva. II. Título.

“Espere o melhor, prepare-se para o pior e aceite o que vier.”
— PROVÉRBIO CHINÊS

AGRADECIMENTOS

Aos meus pilares, Pai, Mãe e Irmã, que me sustentam por todos esses anos com seu amor e apoio incondicionais.

Aos ilustres professores Taisy Silva Weber, orientadora, e Sérgio Luis Cechin, aos quais não tenho palavras para agradecer pelo apoio, dedicação e confiança depositados em mim nesta jornada, que certamente serão levados como exemplo em minha carreira.

À minha namorada, amigos e colegas de laboratório que, de alguma forma, também têm sua parte nesta conquista.

Aos queridos amigos mais distantes, Felipe "ecl" Pena e Higor "enygmata" Eurípedes que, dentre tantos projetos, colaboraram fielmente na implementação do protótipo utilizado para realização de experimentos neste trabalho.

Ao querido colega Carlos Morais pelas nem tão incansáveis dicas e demais contribuições para este trabalho.

Ao CNPq pela concessão da bolsa de mestrado.

Por fim, àqueles que me desprendi durante este período, mas que continuaram vibrando e sofrendo comigo em todos os momentos.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

DBS	Distributed Benchmark System
FAIL	FAult Injection Language
FCI	FAIL Cluster Implementation
FIERCE	Fault Injection Environment for Remote Communication Evaluation
FIMM	Fault Injection Monitoring Module
FIONA	Fault Injector Oriented to Network Applications
FIRMAMENT	Fault Injection Relocatable Module for Advanced Manipulation and Evaluation of Network Transports
FIRMI	Fault Injection for RMI
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
JVM	Java Virtual Machine
JVMTI	JVM Tool Interface
LWFI	LightWeight Fault Injector
NETEM	Network Emulator
PFI	Protocol Fault Injection
PIE	Partitioning Injection Environment
RMI	Remote Method Invocation
SWIFI	Software Implemented Fault Injection
TCL	Tool Command Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UFRGS	Universidade Federal do Rio Grande do Sul
WANEm	Wide Area Network Emulator

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A busca por características de dependabilidade em aplicações distribuídas está cada vez maior. Para tanto, técnicas de tolerância a falhas são componentes importantes no processo de desenvolvimento de um software, e requerem a reprodução de cenários específicos de falhas para possibilitar uma avaliação adequada.

Nestes casos, resta ao engenheiro de teste a integração de experimentos da aplicação-alvo com ferramentas auxiliares para emulação de um ambiente fiel para a execução de testes. Entretanto, tais ferramentas auxiliares, designadas injetores de falhas de comunicação, muitas vezes não estão disponíveis para a comunidade ou, na melhor das hipóteses, apresentam baixa funcionalidade, seja pela incompatibilidade com sistemas mais atualizados, seja pela implementação superficial de funções específicas (protótipos).

Outro fator agravante para a realização de avaliações experimentais em aplicações distribuídas está no suporte a falhas distribuídas, ou seja, injetores de falhas de comunicação não, obrigatoriamente, estão aptos a reproduzir os comportamentos necessários para emulação de ambientes distribuídos adequados. Desta forma, este trabalho destina-se ao estudo e proposta de uma solução para injeção de falhas em ambientes distribuídos, em especial o particionamento de rede, e deu origem ao injetor de falhas PIE.

PIE (*Partitioning Injection Environment*) é um injetor de falhas de comunicação voltado para injeção de particionamentos de rede. Sua arquitetura distribuída permite o controle centralizado do ambiente por parte do engenheiro de testes. Com isso, a criação de uma única carga de falhas pode ser facilmente replicada para os demais nodos componentes do ambiente experimental. Apesar de adotar um coordenador de experimentos, durante a execução de testes, cada nodo interpreta sua carga de falhas e processa-a localmente, garantindo a baixa intrusividade da ferramenta e evitando a ocorrência de comportamentos inesperados pela aplicação-alvo.

Como mecanismo de avaliação desta proposta foram realizados experimentos com diferentes aplicações-alvo, disponibilizadas pelo *framework JGroups*, com um conjunto de cenários de falha específico para cada aplicação. Desta forma, foi possível comprovar a viabilidade e utilidade do modelo e arquitetura do injetor de falhas *PIE* levando em consideração sua funcionalidade, intrusividade e corretude dos resultados experimentais.

Palavras-chave: Injeção de falhas de comunicação, Particionamento de rede.

Communication Fault Injection in Distributed Environments

ABSTRACT

The search for dependability characteristics in distributed applications is increasing quickly. For these, fault tolerance techniques are important components in software development and requires the emulation of specific scenarios to allow a proper evaluation.

In these cases, it remains to the test managers the integration of the target application with extra tools for a faithful emulation environment. However, such tools, named communication fault injectors, are not available to the community or, in other cases, presents a very poor functionality, incompatibility with current systems, either by superficial implementation of specific functions (prototypes).

Another problem for achieving experimental evaluations in distributed applications is the support to distributed faults. Communication fault injectors not necessarily are able to reproduce the behaviors required for proper environment emulation. Thus, this work aims to study and propose a solution for fault injection in distributed environments in particular network partitioning, and led to PIE fault injector.

PIE (Partitioning Injection Environment) is a communication fault injector aimed to network partitioning injection. Its distributed architecture allows centralized control by the test manager. Thus, a fault load can be easily replicated to other nodes. Despite adopting a experiment coordinator, each node interprets its fault load and processes it locally during testing, ensuring PIE low intrusiveness and avoiding the occurrence of unexpected behavior by the target application.

As an assessment of this work, experiments were done with different target applications, provided by JGroups framework, with a set of specific fault scenarios to each application. Thus, it was able to prove the feasibility and usefulness of the model and architecture of the PIE fault injector considering its functionality, intrusiveness and correctness of the experimental results.

Keywords: Communication fault injection, Network partitioning.

1 INTRODUÇÃO

A demanda por aplicações com elevados requisitos para a obtenção de um produto com características de dependabilidade apresentou um crescimento vertiginoso nos últims anos. Isto, pois, na medida em que os avanços tecnológicos passam a fazer parte do nosso cotidiano, novos problemas oriundos desta evolução vem à tona e precisam ser tratados adequadamente para evitar resultados indesejáveis. Não é diferente com as redes de comunicação que, apesar de interligar os pontos mais remotos do planeta em um simples *click* do mouse, sofrem interferências, sejam elas humanas, sejam naturais, e devem ser levadas em consideração (BUCHACKER; SIEH, 2001; TSAI et al., 1999).

Estas interferências, se não tratadas, podem comprometer completamente a transferência de informações pelo canal de comunicação, pois pacotes podem sofrer atrasos na entrega ou, na pior das hipóteses, serem descartados indistintamente. Embora este tipo de evento não apresente soluções eficazes para diminuir o impacto das interferências sobre o canal de comunicação, algumas aplicações passaram a tratar tais interferências no nível de *software* a partir da implementação de mecanismos de tolerância a falhas.

Visando avaliar a corretude da implementação dos mecanismos de tolerância a falhas de um projeto de *software*, técnicas de injeção de falhas (HSUEH; TSAI; IYER, 1997) são comumente utilizadas para testes de aplicações de rede com o objetivo de verificar seu comportamento diante da ocorrência de falhas de comunicação. Além disso, o processo de desenvolvimento de *software*, inclui etapas de teste onde o foco é a identificação de falhas transitórias que podem afetar a aplicação-alvo e, por sua vez, devem ser tratadas pelos mecanismos de tolerância a falhas da aplicação. Desta forma, o resultado proveniente dos testes de injeção de falhas, além de auxiliar na avaliação de dependabilidade de aplicações-alvo, também contribui no seu processo de correção por parte dos desenvolvedores (AVIZIENIS et al., 2004).

Dentre a natureza das aplicações desenvolvidas, principalmente as aplicações de rede que utilizam algoritmos de consistência de réplicas, eleição de líder, entre outros, é possível identificar um caráter distribuído. Em outras palavras, trata-se de uma instância da aplicação de rede está presente em cada nodo da rede para que assim possa desempenhar atividades coordenadas. O teste destas aplicações - referenciadas neste trabalho por aplicações distribuídas, utilizando injeção de falhas, é o escopo deste trabalho. Nestas aplicações, características de dependabilidade são atributos importantes que uma aplicação distribuída deve apresentar. Entretanto, tais aplicações enfrentam grandes desafios e problemas para garantir estas características quando ocorrem falhas de particionamento de rede. Isto, pois, um único nodo é incapaz de detectar falhas desta natureza diretamente, impossibilitando a distinção entre colapsos de nodo ou *link* e nodos particionados pela rede.

Tendo em vista a exploração de falhas comuns de comunicação, como colapsos de

nodo, corrompimento de pacotes, entre outros, já realizada por diversos trabalhos na literatura (DAWSON; JAHANIAN; MITTON, 1997; JACQUES-SILVA et al., 2004; DREBES, 2005; WIERMAN, 2006), aqui optou-se por explorar falhas específicas de ambientes distribuídos, em especial o particionamento de rede (BIRMAN, 1997; VERÍSSIMO; RODRIGUES, 2001).

Conforme sua especificação, aplicações distribuídas, assim como qualquer outra com requisitos de dependabilidade, devem-se manter operantes mesmo diante de situações anormais. Do contrário, quaisquer falhas não identificadas, e devidamente tratadas no processo de desenvolvimento, podem comprometer seriamente um sistema em produção. Logo, é necessária a realização de testes bem definidos em um ambiente propício para a reprodução de falhas no canal de comunicação.

Uma abordagem para a realização de testes é a espera pela ocorrência natural de falhas no canal de comunicação, que implica em um tempo indeterminado para a conclusão da etapa de testes sobre a aplicação alvo. Uma alternativa mais eficaz sugere a emulação de falhas com alta controlabilidade a partir de técnicas de injeção de falhas. Outra alternativa apresenta a emulação manual de falhas, através da desconexão de cabos por exemplo, que também é funcional, embora apresente baixa controlabilidade e eficiência sobre o ambiente desejado.

Tendo em vista estas abordagens e os requisitos para realização de testes adequados em aplicações distribuídas, para obter-se alta controlabilidade no processo de emulação de falhas, uma ferramenta de injeção de falhas deve ser capaz de mapear os nodos que compõe a rede utilizada pela aplicação distribuída. Além disso, a ferramenta deve suportar a manipulação do(s) protocolo(s) trafegado(s) no canal de comunicação, uma vez que quaisquer mensagens indevidas que violem o bloqueio podem comprometer a integridade do experimento. Por fim, a ferramenta de injeção de falhas deve ser independente da instrumentação do código fonte da aplicação alvo. Entretanto, os injetores de falhas encontrados na literatura impõem diversas dificuldades ao engenheiro de testes, seja devido às limitações do injetor no que diz respeito aos tipos de falhas suportados (JACQUES-SILVA et al., 2004; HOARAU; TIXEUIL, 2005; WIERMAN, 2006), seja pelo suporte a um único protocolo específico (GERCHMAN; WEBER, 2006; VACARO; WEBER, 2006).

1.1 Objetivos

O foco desta dissertação é observar o comportamento de aplicações distribuídas sensíveis a falhas de particionamento de rede, com o objetivo de avaliar a cobertura de falhas dos mecanismos de tolerância a falhas de particionamento implementados na aplicação. Conforme apresentado brevemente nesta introdução, há ausência de ferramentas voltadas especificamente a esse fim. Assim, o maior objetivo do trabalho é a elaboração de uma solução para injeção de falhas de particionamento de rede em aplicações distribuídas.

1.2 Motivação

Sendo o particionamento de rede uma falha potencial em qualquer sistema distribuído, é comum encontrarmos mecanismos de tolerância a falhas nas mais diferentes áreas de atuação, que vão desde algoritmos de consistência de réplicas em uma agência bancária até mecanismos de controle de tráfego aéreo.

Entretanto, para possibilitar a avaliação destes ambientes, um engenheiro de testes

necessita de um sistema de injeção de falhas com características específicas, porém inexistentes nos injetores de falhas atuais (DAWSON; JAHANIAN; MITTON, 1996; LEFEVER et al., 2003,?; HOARAU; TIXEUIL, 2005; DREBES, 2005), dentre elas destacam-se:

- a necessidade de coação entre sistemas seguindo alguns estados, a fim de garantir que rotinas específicas sejam executadas de forma transparente à aplicação;
- a necessidade de definir métricas e coletar dados acerca do comportamento da aplicação-alvo diante de falhas, para que seja possível identificar vulnerabilidades, bem como a comparação de resultados com aplicações similares.

Além disso, um benefício adicional proporcionado por esta proposta é oferecer ao engenheiro de testes a capacidade de testar várias aplicações com o aprendizado de uma única ferramenta.

1.3 Resultados Alcançados

Entre os resultados alcançados por este trabalho, estão:

- Definição de requisitos para uma ferramenta de injeção de falhas de particionamento de rede voltada ao teste e avaliação de aplicações distribuídas.
- Análise do potencial de ferramentas e *frameworks* existentes para a injeção de falhas de particionamento de rede em aplicações distribuídas e conclusão sobre a necessidade de uma nova ferramenta.
- Especificação de uma solução para o problema de injeção de falhas em aplicações distribuídas visando suprir a necessidade identificada. A solução é adequada para a realização de testes de caixa preta, podendo auxiliar na obtenção de métricas de dependabilidade.
- Desenvolvimento de *PIE*, um protótipo para injeção de falhas de particionamento de rede em aplicações distribuídas, buscando cumprir com os requisitos identificados para a solução. O protótipo pode ser aplicado para injetar falhas em aplicações que se comuniquem através de qualquer protocolo baseado em *IP*. *PIE* provê boa representatividade de falhas, emulando falhas de forma realística. Também possui boa portabilidade em ambientes baseados em *unix* e não requer a instrumentação do código fonte das aplicações alvo.
- Condução de experimentos de injeção de falhas com o protótipo desenvolvido, mostrando a viabilidade do modelo e da arquitetura propostos para o teste de aplicações distribuídas sensíveis ao particionamento de rede;
- Até o momento, o estudo dedicado nesta pesquisa conquistou duas publicações, uma no *X Workshop de Testes e Tolerância a Falhas* e outra no *11th Latin American Test Workshop (LATW)*, e podem ser encontradas nas referências bibliográficas (OLIVEIRA; CECHIN; WEBER, 2009, 2010).

1.4 Organização do Texto

O restante deste trabalho organiza-se como segue: o capítulo a seguir levanta conceitos importantes acerca de injeção de falhas, enfatizando falhas em meios de comunicação.

O capítulo 3 relaciona um estudo sobre trabalhos relevantes encontrados na literatura, com o objetivo de encontrar pontos de intersecção para com a presente proposta.

Já o capítulo 4 apresenta a proposta do injetor de falhas *PIE*, definindo sua arquitetura, mecanismos de construção de cenário, e demais atributos que compõem sua estrutura.

O capítulo 5, por sua vez, demonstra a aplicabilidade da ferramenta com experimentos que justificam o uso de um injetor de falhas no processo de desenvolvimento de *software*, levando em consideração seu funcionamento, nível de intrusividade, e integridade dos resultados apresentados pelo mesmo.

Por fim, este trabalho termina com as conclusões, contribuições e trabalhos futuros no capítulo 6.

2 REVISÃO DE LITERATURA

Com os avanços tecnológicos no meio computacional, as redes de computadores de alta velocidade permitem que centenas de nodos espalhados pelo mundo possam trocar informações em um piscar de olhos. Tal comportamento infere na possibilidade de cooperação entre os nodos, formando um sistema distribuído.

Segundo Birman (1997), um sistema distribuído consiste em um grupo de aplicações, executando em um ou mais nodos, que coordenam atividades através da troca de mensagens. Tanenbaum e Steen (2006), por sua vez, definem um sistema distribuído como um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente.

Apesar de agregar uma série de problemas inexistentes em sistemas auto-contidos, um sistema distribuído objetiva construir um sistema que possa recuperar-se de falhas parciais automaticamente, sem efeitos colaterais aos demais nodos (TANENBAUM; STEEN, 2006). Ou seja, um sistema distribuído deve permanecer em funcionamento de maneira aceitável mesmo na presença de falhas.

Em virtude da possibilidade ocorrência de falhas em sistemas distribuídos, aplicações que operam nestes ambientes devem incorporar técnicas de tolerância a falhas que visam, primeiramente, cobrir quaisquer ocorrências anômalas, de modo que a aplicação mantenha um serviço íntegro e confiável. Entretanto, tais mecanismos de tolerância a falhas devem ser testados e avaliados, pois, caso seu papel não seja cumprido, há margem para a propagação de falhas que, por sua vez, podem levar a aplicação-alvo a apresentar defeito. Além disso, a avaliação de uma aplicação-alvo não é uma tarefa trivial, visto que são formadas por protocolos de comunicação e sistemas distribuídos que estão cada vez mais difíceis de testar (DAWSON; JAHANIAN; MITTON, 1997).

Na literatura, é possível encontrar diferentes mecanismos de avaliação de dependabilidade de aplicações distribuídas, tais como modelos analíticos e formais. Contudo, este trabalho adota mecanismos de injeção de falhas implementadas em *software* (SWIFI – *Software Implemented Fault Injection*), uma solução factível e eficiente, uma vez que não depende da ocorrência natural de falhas, provê alta controlabilidade sobre o ambiente de testes, e não requer o uso de periféricos com custos elevados (CLARK; PRADHAN, 1995; MARTINS; RUBIRA; LEME, 2002; LOOKER; MUNRO; XU, 2005).

2.1 Injeção de Falhas

Desde a década de 70, a utilização de injeção de falhas como um método experimental da verificação de dependabilidade em diversos sistemas vem resultando em diferentes técnicas para a reprodução de comportamentos anômalos, seja por simulação, por *hardware*, ou ainda por *software*. A caracterização de injetores de falhas está relacionada com diver-

sos atributos, tais como: controlabilidade, indicador de tempo (quando) e espaço (onde) para a injeção de falhas; repetibilidade, que diz respeito à precisão na habilidade de repetir um experimento inúmeras vezes sem a necessidade de reconfiguração; observabilidade, atributo indicador da capacidade de prover mecanismos para coleta de métricas e análise dos efeitos causados por falhas no sistema.

Outro aspecto importante no projeto de um injetor de falhas é a diminuição do impacto da intrusividade na aplicação-alvo, uma vez que não pode ser evitada. Desta forma, cabe aos desenvolvedores destas ferramentas buscarem a minimização deste impacto.

De acordo com sua natureza, a intrusividade pode ser de duas formas: temporal e espacial. A intrusividade temporal, também chamada de perturbação, é identificada quando a ação de um injetor de falhas implica no aumento do tempo de processamento da aplicação-alvo. Isto, pois *SWIFI* demanda tempo para execução de suas rotinas de interceptação do fluxo da aplicação-alvo, bem como de espaço adicional em memória, sendo impossível anular este tipo de intrusividade.

A intrusividade espacial, geralmente, está relacionada à instrumentação direta do código fonte da aplicação-alvo. Ou seja, a aplicação já vem compilada com rotinas necessárias para injeção de falhas. Entretanto, seu uso pode levar a inserção de *bugs* não intencionais e alterar o ciclo de execução da aplicação, caracterizando-se também um cenário com intrusividade temporal.

Independente de sua abordagem para injeção de falhas, a maior parte das ferramentas de injeção de falhas não estão acessíveis à comunidade ou, se disponibilizadas, há escassez de documentação e, muitas vezes, são pouco funcionais, seja por sua obsolescência, seja sua arquitetura voltada para ambientes muito específicos. Por essa razão, injetores de falhas tornaram-se ferramentas muito valiosas no âmbito de avaliação e análise de dependabilidade de sistemas.

Dentre os injetores de destaque pode-se citar Xception (CARREIRA; MADEIRA; SILVA, 1998), um injetor de falhas de *hardware* por *software*. Neste, mecanismos de injeção de falhas são implementadas em *software* que possibilitam a introdução de falhas no nível de *hardware*, como corrompimento de memória, alteração de registradores, entre outros.

Entretanto, conforme citado anteriormente, este trabalho volta-se para avaliação de dependabilidade de aplicações em sistemas distribuídos. Logo, o foco desta pesquisa concentra-se em injetores de falhas de comunicação, em especial àqueles com suporte a sistemas distribuídos.

2.2 Modelo de Falhas de Comunicação

Atualmente, injetores de falhas de comunicação adotam um cenário genérico em diferentes níveis: (i) nível de mensagem, onde as falhas afetam toda e qualquer mensagem que componha o fluxo de rede do barramento, seja atrasando-a, seja descartando-a, e (ii) nível lógico, onde as falhas afetam componentes de rede, forçando-o a assumir um determinado comportamento, seja por colapso, omissão ou temporização. Existe ainda um outro nível, físico, que infere na alteração de comportamento da topologia utilizada em um determinado ambiente, mas não é comum nos injetores de falhas encontrados na literatura.

Além disso, injetores de falhas de comunicação não suportam, obrigatoriamente, injeção distribuída de falhas, referentes à realização de experimentos de forma coordenada e cooperativa para execução de uma determinada tarefa em ambientes distribuídos.

A carência de suporte a sistemas distribuídos, por parte dos injetores de falhas encontrados na literatura, descritos no capítulo a seguir, dão margem para o estudo de um tipo especial de falha de comunicação. Trata-se do particionamento de rede, uma fragmentação da rede em sub-redes desconectadas que impede a coordenação e atividades entre estações presentes em partições distintas da rede (BIRMAN, 1997; VERÍSSIMO; RODRIGUES, 2001).

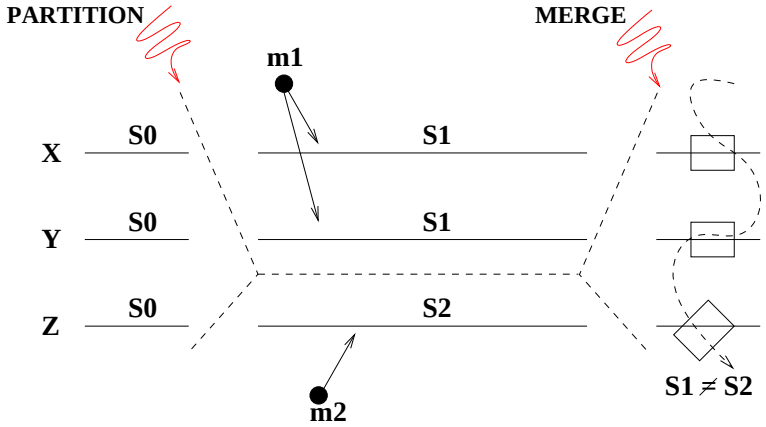

Figura 2.1: Exemplo de particionamento de rede.

A figura 2.1 exemplifica o problema de particionamento de rede. Três processos iniciam com o mesmo estado *S0*. Um eventual particionamento de rede mantém as estações *X* e *Y* conectadas, enquanto *Z* encontra-se isolada em outra partição. Em seguida, a mensagem *m1* é processada por *X* e *Y* que passam para o estado *S1*. Na outra partição, a mensagem *m2* é processada pela estação *Z*, alterando seu estado atual para *S2*. Quando o particionamento de rede é tratado (*merge*), *S1* e *S2* são diferentes acarretando em uma divergência entre estados. Logo, se a aplicação necessita garantir alguma propriedade de dependabilidade ou consistência, são necessárias estratégias para tratar conflitos gerados pelo particionamento que, por sua vez, precisam ser provocados para que estas estratégias possam ser avaliadas.

2.2.1 Modelo de Sistema Adotado

O modelo de sistema adotado neste trabalho é ilustrado na figura 2.2. Assim como na figura anterior (figura 2.1), consiste em uma série de nodos (*X*, *Y*, *Z*) variando no tempo *t*, onde são definidas as ocorrências de eventos. Os eventos, por sua vez, podem ser de particionamento de rede, ou de *merge*, e são referenciados por *E1* até *En*.

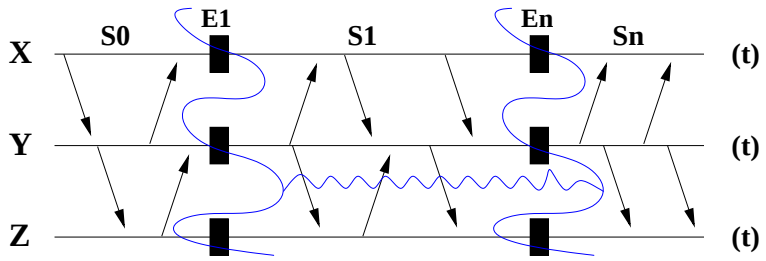

Figura 2.2: Modelo de sistema adotado.

Apesar de em um ambiente distribuído a troca de estados de uma aplicação ocorrer através do envio e recebimento de mensagens, referenciados por setas, o conceito de esta-

dos (*S0*, *S1*, *Sn*...) adotados neste trabalho refere-se à conectividade dos nodos envolvidos no ambiente experimental. Para um sistema de injeção de falhas de particionamento de rede, o fator determinante para uma emulação correta de particionamento de rede é a visão de rede de cada nodo (particionado ou não).

Definido o presente cenário e suas considerações, foge ao escopo deste trabalho a análise das trocas de estados de uma aplicação durante a campanha de testes, função devidamente exercida pelo engenheiro de testes, e assume-se o conceito de estados físicos dos nodos no ambiente experimental.

2.3 Mecanismos de Injeção de Falhas de Comunicação

Para injeção de falhas, dois métodos podem ser considerados; em tempo de compilação, que consiste na alteração do código do sistema em teste; e em tempo de execução, que permite a emulação de falhas através de *time-outs*, interrupções, entre outros.

Seguindo métodos de injeção de falhas em tempo de compilação, tem-se a instrumentação de código da aplicação-alvo. Em outras palavras, modifica-se o código-fonte das aplicações, ou ainda de bibliotecas de comunicação, alterando instruções geradas pelo compilador, ou até mesmo na adição de novas instruções.

Apesar do baixo custo de implementação e alta controlabilidade, a utilização desta abordagem exige a modificação da aplicação-alvo, prejudicando a reusabilidade em outras aplicações-alvo e gerando grande intrusividade espacial. Contudo, seu nível de perturbação temporal geralmente é baixo, visto que as rotinas de injeção de falhas estão embutidas em pontos apropriados do código onde a falha é planejada para ocorrer.

A figura 2.3 descreve um modelo para interceptação que apresenta uma aplicação e sua respectiva requisição de acesso à função *XYZ()*. Em um fluxo normal de operações, figura 2.3(a), a definição do símbolo em questão é determinado em tempo de execução a partir da biblioteca padrão referenciada por *libZ.so*. Logo, a adição de uma nova biblioteca, pré-carregada no espaço de endereçamento de processos à frente da biblioteca padrão, com uma definição alternativa para a função *XYZ()* (*libMyZ.so*) permite a interceptação da definição original, conforme apresentado na figura 2.3(b).

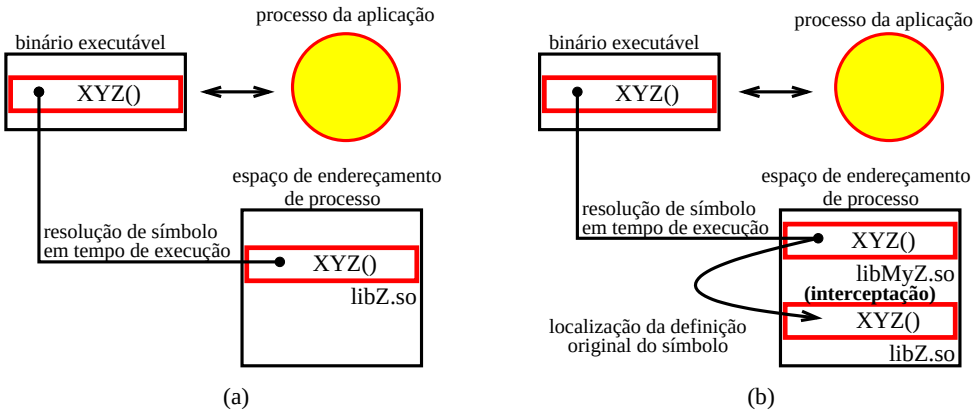

Figura 2.3: Injeção de falhas em tempo de compilação.

Nesta abordagem, apesar de propiciar facilidades de implementação, a usabilidade é limitada a aplicações muito específicas. Desta forma, aplicações que dependem de rotinas distintas das rotinas alteradas para proporcionar a injeção de falhas, não seriam suportadas neste modelo, a menos que sejam reimplementadas.

Já os métodos baseados em tempo de execução, além do alto grau de controlabilidade, podem apresentar alto grau de perturbação (intrusividade temporal). A interceptação de *system calls*, por exemplo, permite a emulação de situações de falha através da manipulação das chamadas ao sistema interceptadas de tal maneira a forçar o sistema em uso a operar sob situações críticas. Porém, verificar as ocorrências de cada chamada de sistema na tabela do sistema operacional para cada mensagem no fluxo de rede se torna um processo oneroso, degradando o desempenho da aplicação-alvo.

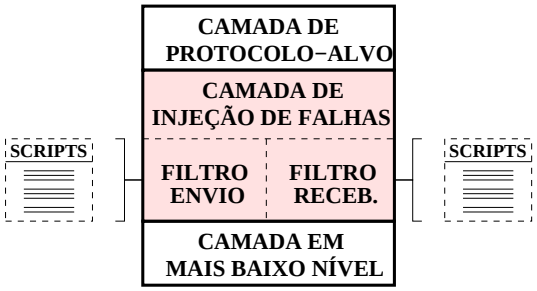

Figura 2.4: Injeção de falhas por inserção de camada na pilha de protocolos.

Outra possibilidade para injeção de falhas em tempo de execução, e mais comumente adotada, está baseada na inserção de uma camada injetora de falhas entre duas camadas na pilha de protocolos do sistema, ou seja, não é feita uma distinção entre protocolos em seus diversos níveis.

Conforme visto na figura 2.4, a camada inserida abaixo do protocolo alvo realiza o filtro e manipulação sobre a troca de mensagens entre os participantes na comunicação a fim de provocar anomalias. Logo, quaisquer mensagens trafegantes na pilha de protocolos do sistema operacional são interceptadas pela camada de injeção de falhas que, após realizar os procedimentos de entrada e saída descritos por seus *scripts*, repassam (ou não) a informação para sua camada superior imediata.

Além destes mecanismos, muitos outros são adotados por injetores de falhas. Entretanto, em um nível mais baixo de abstração, são similares a um dos mencionados acima, seja emulando um *driver ethernet* (LI et al., 2002), seja instrumentando códigos a partir de ferramentas de apoio, como *JVMTI* (JACQUES-SILVA et al., 2004).

Outras peculiaridades também podem ser encontradas, como as diferentes formas de descrição de uma carga de falhas a ser reproduzida pelo ambiente, bem como os mecanismos de coleta de dados experimentais.

2.4 Mecanismos de Descrição de Carga de Falhas

Dada a necessidade de avaliação de aplicações distribuídas nos mais variados tipos de ambiente (seja com diferentes protocolos de comunicação, seja com diferentes topologias de rede), injetores de falhas de comunicação devem prover meios para descrição de tais ambientes de forma que estas informações permaneçam transparentes à aplicação, bem como ao engenheiro de testes.

Para a descrição de ambientes, injetores de falhas implementam em sua estrutura uma linguagem para permitir ao engenheiro de testes a definição de comportamentos específicos dos nodos participantes dos experimentos. Esta descrição do cenário experimental determina que comportamento um determinado nodo assume diantes dos demais. Além disso, podem ser definidos mecanismos de disparo de falhas de acordo com a necessidade

do engenheiro de testes, uma vez que o disparo incorreto pode comprometer o resultado final do experimento.

Abordagens com ênfase em alta portabilidade para diversas plataformas podem ser encontradas na literatura. Aidemark et al. (2001) apresentam um *framework* com um modelo descritor de cargas de falhas genérico. Neste, o engenheiro de testes pode utilizar métodos pré-disponibilizadas, ou ainda desenvolver à sua maneira e incorporar ao *framework* em forma de *plug-in* seguindo um modelo padrão associado ao mesmo.

Outros modelos baseados em uma linguagem própria para descrever cenários de falhas também podem ser encontrados, seja de alto ou baixo nível. Esta prática, apesar de oferecer uma linguagem voltada unicamente para descrição de falhas, impõe a necessidade de aprendizado da mesma. Em contrapartida, outros injetores de falhas optam pela adoção de linguagens de mais alto nível e mais difundidas, como *TCL scripts*, *C++*, *Java*, entre outros.

2.5 Mecanismos de Coleta de Dados Experimentais

Complementos ao mecanismo de descrição de cargas de falhas também podem ser encontrados. A coleta de dados experimentais, por sua vez, permite a definição de métricas a serem avaliadas no ambiente, possibilitando a identificação de gargalos, a propagação de falhas, erros no sistema, entre outros, e será abordada a seguir.

Estudos realizados, para este trabalho, sobre testes experimentais com injetores de falhas permitem observar a grande dificuldade na coleta de dados do ambiente avaliado. Normalmente, injetores de falhas não agregam técnicas de filtragem de dados, tornando o processo de monitoramento de falhas injetadas um processo lento e custoso, uma vez que implica em recolher vestígios (com ferramentas auxiliares como o *tcpdump*, *klog*, *Wireshark*, etc.) e avaliá-los manualmente por meio de programas de especificação de casos de teste.

Kanawati, Kanawati e Abraham (1995), por exemplo, propõem o *logging* de resultados para cada execução do experimento, onde são armazenados dados como a localização da falha/erro (endereço virtual), o *bit* alterado, assim como o registrador alterado. Desta forma, um monitor adiciona *flags* de estado para cada execução para indicar se a falha foi corretamente injetada e levou o sistema a ocorrência de erros. Por fim, o módulo de coleta de resultados filtra estes resultados, com sua respectiva *flag* de estado associada, e determina contadores e valores percentuais, como cobertura de falhas, latência e tipos de mecanismos para detecção de erros.

Neogi, De, e Chiueh (2003), por sua vez, apresentam um mecanismo de análise que opera a partir do monitoramento e contagem de pacotes de interesse trocados entre os participantes do experimento. Sua natureza distribuída ainda requer um protocolo de controle para a troca de informações entre nodos do sistema em teste, como contadores, entre outros.

Conforme visto anteriormente, a integração de modelos para coleta e análise de dados experimentais ainda é pouco frequente em ferramentas injetoras de falhas e, dada sua necessidade durante a realização de testes de dependabilidade de aplicações distribuídas, é incluída a associação destes mecanismos na proposta final deste trabalho.

2.6 Conclusões de Capítulo

Conforme apresentado no capítulo, técnicas injetoras de falhas podem interferir diretamente na execução de um sistema e apresentam vantagens e desvantagens.

A abordagem de injeção de falhas em tempo de compilação, por exemplo, traz facilidades de implementação, muito embora sua usabilidade possa ser limitada à aplicações específicas. Em outras palavras, alterações realizadas no código-fonte de um determinado aplicativo nem sempre podem ser reproduzidas em outras aplicações.

Já a abordagem para injetar falhas em tempo de execução apresenta-se mais flexível, uma vez que não depende de alterações no código-fonte de aplicações-alvo para injeção de falhas. Além disso, a definição de gatilhos para controle das injeções dá grande liberdade ao desenvolvedor para injeção de falhas específicas, que vão desde um determinado pacote na rede, até um estado específico do sistema para reprodução de comportamentos difíceis de se conseguir sem o uso de um injetor de falhas.

Em um âmbito para a criação do cenário de falhas, cada proposta é associada às necessidades da arquitetura vigente, ou seja, abordagens mais simples, como a interceptação de *system calls*, possuem mecanismos descritores de falhas em linguagens amplamente difundidas (*C++*, *Java*, entre outras), pois as regras definidas pelo usuário estão diretamente ligadas à manipulação de *system calls* de acordo com os requisitos do sistema. Entretanto, conforme será apresentado no próximo capítulo, algumas ferramentas necessitam reproduzir comportamentos específicos do sistema com grau de controle e peculiaridade elevados, justificando a criação de uma linguagem própria como mecanismo para criação de ambientes específicos em experimentos.

Após a realização de experimentos, as dificuldades concentram-se na coleta de dados para análise do comportamento do sistema, uma vez que o mesmo está sujeito a estouro de *buffers*, por exemplo, dificultando o filtro de informações geradas pelo próprio sistema. Entretanto, a pequena, ou ainda inexistente documentação dos modelos propostos dificulta a análise desta função, restando o emprego de heurísticas por parte dos desenvolvedores, que focam em suas respectivas áreas de interesse.

Baseados nestes mecanismos estudados, o capítulo a seguir apresentará uma relação de injetores de falhas de comunicação encontrados na bibliografia com o objetivo de analisar modelos de implementação com baixa intrusividade no sistema operante e alta controlabilidade sobre falhas injetadas.

3 TRABALHOS CORRELATOS

Este capítulo apresenta uma revisão de literatura acerca de diversas ferramentas destinadas à avaliação de protocolos ou aplicações distribuídas. Estes, por sua vez, estão divididos entre ferramentas de teste, e ferramentas de injeção de falhas. Injetores de falhas ainda encontram-se separados de acordo com sua natureza: local e distribuída.

3.1 Ferramentas de Teste

Com o passar dos anos, diversas ferramentas auxiliares para testes de protocolos de rede foram desenvolvidas com o objetivo de avaliar não apenas sua implementação, como também seu desempenho.

Ferramentas de teste de comunicação diferenciam-se de injetores de falhas de comunicação por não suportarem mecanismos de construção de cenários específicos de falhas. Em outras palavras, ferramentas de teste não permitem a elaboração de cenários específicos e com alta controlabilidade no que diz respeito às ocorrências dos eventos desejados. A seguir, segue uma relação das ferramentas encontradas com maior intensidade na literatura.

Dentre as ferramentas mais populares destacam-se (i) *DBS* (*Distributed Benchmark System*) (MURAYAMA; YAMAGUCHI, 1997), uma ferramenta de avaliação de desempenho de redes TCP/IP, que permite múltiplos envios de dados de forma coordenada para posterior avaliação do comportamento do protocolo *TCP*; (ii) *Dummynet* (RIZZO, 1997), uma ferramenta com alto grau de flexibilidade para gerenciamento de banda a partir da simulação de condições desejáveis no tráfego de rede, como atrasos, filas, descartes de pacotes, entre outros; por fim, uma ferramenta de proposito geral, (iii) *NIST Net* (CARSON; SANTAY, 2003), que permite a emulação de características críticas presentes em redes de larga escala, como congestionamentos e descarte de pacotes, para fins de avaliação do desempenho dinâmico em redes *IP* (*Internet Protocol*).

Recentemente outras ferramentas foram desenvolvidas para permitir a avaliação de protocolos de comunicação e/ou aplicações distribuídas.

HexInject (ACRI, 2010), é um injetor de pacotes e um *sniffer* de rede baseado no *framework Netfilter*, que provê mecanismos para leitura, interceptação e alteração do tráfego de rede de maneira transparente à aplicação-alvo e permite a construção de poderosos *scripts* em conjunto com outras ferramentas de linha de comando.

NetEm (HEMMINGER, 2005), ou *Network Emulation* é outra ferramenta de teste de protocolos de comunicação que emula propriedades de redes de larga escala para avaliação de propriedades de qualidade de serviço. Tais propriedades consistem no atraso, descarte, duplicação e reordenamento de mensagens através do *framework Traffic Control*, disponível no Kernel do Linux 2.6.

Outra ferramenta, também baseada no *framework Traffic Control*, é o *WANEm* (*Wide Area Network Emulator*). Além das propriedades emuladas pela ferramenta *NetEm*, este permite a emulação de corrompimento, colapsos de *link* e *jitter* para possibilitar a reprodução de comportamentos peculiares ao tráfego de voz.

Entretanto, apesar da alta disseminação destas ferramentas de teste, as necessidades de avaliação dos requisitos mínimos para alcançar dependabilidade não são satisfeitos. Mesmo com o suporte à reprodução de comportamentos característicos de ambientes de comunicação, como o descarte e atraso de mensagens, ferramentas de teste não suportam a construção de um cenário de falhas elaborado para a realização de testes bem definidos de acordo com o engenheiro de testes, de maneira que funções específicas do alvo sob teste possam ser executadas corretamente.

3.2 Injetores de Falhas

Em contrapartida às ferramentas de teste, e para proporcionar melhores condições de descrição de cenários, diversos injetores de falhas de comunicação vêm sendo propostos pela comunidade cientifica. Estes, por sua vez, visam garantir a realização de testes, seguindo especificações definidas pelo engenheiro de testes, seja para avaliação de protocolos de comunicação, seja para avaliação de aplicações distribuídas. De acordo com sua natureza, injetores de falhas de comunicação podem ser locais ou distribuídos.

3.2.1 Injetores de Falhas Locais

Apesar de tratarem de falhas de comunicação, alguns injetores de falhas podem ser classificados como locais por não possuírem suporte à coordenação de atividades. Desta forma, para elaborar um ambiente onde há interação entre diversas estações com um determinado comportamento sobre o canal de comunicação, é necessário percorrer os nodos da rede um a um para efetuar o preparo da ferramenta, bem como o disparo da execução dos testes.

3.2.1.1 ORCHESTRA

ORCHESTRA (DAWSON; JAHANIAN; MITTON, 1996) é uma ferramenta de injeção de falhas para avaliação e validação de características temporais e de dependabilidade de protocolos distribuídos. Baseada em um simples e poderoso *framework* que introduz um conceito de sondagem e injeção de falhas orientada *scripts* (DAWSON; JAHANIAN, 1995), ORCHESTRA foca-se no desenvolvimento de técnicas de injeção de falhas que podem ser empregadas para (i) detecção de erros de modelo ou implementação de protocolos, (ii) identificação de violação de sua especificação, e (iii) obtenção de conhecimento sobre decisões e definições de modelos por parte dos desenvolvedores.

Criada com o objetivo de operar em diferentes plataformas, a portabilidade e capacidade de injetar falhas diretamente na pilha de protocolos são objetivos a serem alcançados com um mínimo de intrusividade no protocolo alvo. Para tanto, ORCHESTRA define uma arquitetura baseada em camadas onde nenhuma distinção é feita entre protocolos de nível de aplicação, comunicação, ou físico. Esta abordagem consiste na inserção de uma camada *PFI* (*Protocol Fault Injection*) à pilha de protocolos, abaixo da camada a ser testada. Ao inserir esta camada entre duas camadas da pilha de protocolos, torna-se possível o filtro e manipulação de mensagens trocadas entre participantes da comunicação sem a necessidade de alteração do código da camada alvo.

Para coordenar experimentos, ORCHESTRA interpreta *scripts* de injeção de falhas

escritos em uma linguagem de alto nível, *TCL*. Eleita por sua popularidade e emprego em outras ferramentas, a natureza (interpretada) da linguagem *TCL* permite a construção de *scripts* relativamente complexos para manipulação de mensagens, uma vez que o conteúdo da mensagem torna-se acessível à máquina de estados da ferramenta e capacita o acompanhamento do histórico de mensagens, bem como a manutenção de contadores e outras informações de estado. Desta forma, nenhuma recompilação da camada *PFI* é necessária para a realização de novos testes já suportados pela mesma.

Apesar de possuir um poderoso sistema para descrição de cenários de falhas de comunicação, ORCHESTRA não suporta falhas distribuídas para emulação de particionamentos de rede. Ressalta-se também que a utilização desta ferramenta para injeção de falhas em protocolos distribuídos não infere na capacidade de realizar testes coordenados, ou seja, esta não é capaz de injetar falhas em um nodo de acordo com o comportamento de um segundo nodo participante do experimento.

3.2.1.2 NFTAPE

NFTAPE (STOTT et al., 2000) é um *framework* voltado para avaliação de dependabilidade de sistemas distribuídos. Desenvolvido a partir de dificuldades em encontrar ferramentas automatizadas para injeção de falhas que satisfaçam uma série de requisitos, NFTAPE emprega uma arquitetura modular com suporte a múltiplos modelos de falhas, múltiplos mecanismos de disparo de falhas, múltiplos alvos e métodos versáteis de coleta e reporte de erros.

Ao contrário das demais ferramentas, normalmente, a função dos componentes de um injetor de falhas é assumido pelos módulos (LWFI) (*LightWeight Fault Injector*, injetor de falhas leve). Estes módulos correspondem a pequenos programas que se responsabilizam única e exclusivamente pela injeção de falhas, despreocupando-se com demais funções desempenhadas pelo próprio NFTAPE, como gatilhos, *logging*, comunicação, e outros. Além disso, há uma independência entre tais componentes, o que permite a execução de diferentes combinações de módulos *LWFI*, gatilhos e cargas de trabalho para a construção do ambiente de testes.

Motivado pela avaliação de experimentos científicos realizados no espaço sideral, NFTAPE fornece uma série de tipos de injeção de falhas. LWFIs podem ser executados através de mecanismos de depuração, baseados em controladores de dispositivo, com mecanismos específicos para um alvo, baseados em simulação, ou ainda por injeção de falhas físicas. Apesar de focar-se na emulação de falhas geradas por radiação eletromagnética em componentes, NFTAPE também provê suporte para falhas de comunicação, através da injeção de falhas diretamente nos controladores de comunicação dos sistemas distribuídos.

Sua arquitetura baseada em componentes torna o NFTAPE uma ferramenta com alto grau de portabilidade para diferentes plataformas, como *Solaris*, *Linux*, *Windows*, e *Lynx* (sistema operacional de tempo real). Apesar de sua separação entre mecanismos de disparo e de injeção de falhas através de módulos LWFI ser factível para ambientes de testes descritos anteriormente, a ferramenta demonstra-se inviável para testes de protocolos de comunicação. Nestes, a integração de mecanismos de disparo e a injeção de falhas é desejado, uma vez que falhas tendem a ser injetadas sobre pacotes de acordo com seu conteúdo. Em outras palavras, tanto o mecanismo de disparo, quanto o módulo responsável pela injeção necessitam acesso ao mesmo fluxo de comunicação.

3.2.1.3 FIRMAMENT

FIRMAMENT (DREBES, 2005) (acrônimo para *Fault Injection Relocatable Module for Advanced Manipulation and Evaluation of Network Transports*), também desenvolvido pelo grupo de tolerância a falhas da UFRGS, destina-se à validação experimental de técnicas de tolerância a falhas de protocolos de comunicação e sistemas distribuídos. Sua construção a partir do uso da extensibilidade do núcleo *Linux* através de módulos relocáveis minimizam o efeito de sondagem no núcleo.

A ferramenta permite diversas opções para a injeção de falhas de comunicação e permite a especificação de testes através de *faultlets*, uma linguagem de baixo nível para descrição de cenários de falhas. Diferentemente da abordagem proposta pela ferramenta ORCHESTRA, FIRMAMENT foi desenvolvido para utilizar características específicas do núcleo de código fonte aberto *Linux*, de modo que os *faultlets* são convertidos para um conjunto de instruções (*bytecode*) e interpretados pela ferramenta através de sua máquina virtual, proporcionando maior poder para especificação de cenários. Esta linguagem de *bytecode* ainda permite a inspeção e seleção de mensagens de forma determinística ou estatística e provê ações a serem realizadas que o fazem imitar o comportamento de falhas reais, como descarte e duplicação de mensagens, atraso e modificação de conteúdo.

Seu modelo de injeção de falhas explora a arquitetura da interface de programação *Netfilter*, disponível a partir da versão 2.4 do núcleo *Linux*, para acessar o fluxo de execução dos protocolos *IPv4* e *IPv6* através da inserção de uma camada pilha de protocolos do sistema operacional. As mensagens que passam por essa camada são processadas por funções de *callback*, que possuem acesso completo ao seu conteúdo, de acordo com a descrição contida no *faultlet*. Portanto, FIRMAMENT pode ser carregado em qualquer dispositivo executando versões recentes desse sistema sem a necessidade de recompilação do núcleo.

3.2.1.4 FIRMI

FIRMI (VACARO; WEBER, 2006) (acrônimo para *Fault Injector for RMI*), é voltado para avaliação do comportamento mediante a presença de falhas em aplicações *Java* baseadas no protocolo *RMI* (*Remote Method Invocation*). Para possibilitar a integração com diferentes ambientes de desenvolvimento e teste (*JUnit*, *ANT*, etc.), FIRMI foi projetado para ter uma arquitetura simples e modular, e seu mecanismo de descrição de cenários de falhas é realizada através da criação de classes *Java*.

Apesar de suportar em seu modelo de falhas como colapso de nodo, colapso de *link*, falhas de temporização e falhas bizantinas, FIRMI não apresenta uma arquitetura distribuída. Desta forma, não é possível emular falhas de particionamento de rede, alvo deste trabalho, pois não considera a ação coordenada entre injetores de maneira simultânea em um experimento.

Para injeção de falhas de colapso de nodo e colapso de *link*, o injetor combina a interceptação de requisições *RMI* no nível da *JVM* (*Java Virtual Machine*) e a interação com o sistema operacional através de arquiteturas de *firewall*. Desta forma, a emulação de falhas de *RMI* não se prende unicamente no nível da *JVM*, visto que o protocolo opera sobre o protocolo *TCP* e este, por sua vez, é gerenciado pelo *kernel* do sistema operacional. Entretanto, (VACARO; WEBER, 2006) conclui que a emulação de falhas apenas no nível de *kernel* do sistema operacional é inviável, pois o fluxo de mensagens *RMI* neste nível de abstração se encontra serializada. Em outras palavras, é necessária a análise do fluxo de dados com o objetivo de extrair apenas as informações úteis ao injetor.

Entretanto, conforme visto acima, além de não apresentar uma arquitetura distribuída que possibilite a coordenação de ações simultâneas entre os nodos do experimento, este injetor foca-se na injeção de falhas em aplicações *Java* baseadas no protocolo *RMI*. Logo, tais características limitam o uso desta ferramenta em ambientes que independam do(s) protocolo(s) utilizado(s), inviabilizando seu reaproveitamento no presente trabalho.

3.2.1.5 FIERCE

FIERCE (GERCHMAN; WEBER, 2006) (*Fault Injection Environment for Remote Communication Evaluation*) trata-se de um injetor de falhas de comunicação voltado para teste de aplicações *Java*. Entretanto, este injetor aborda unicamente aplicações baseadas no protocolo *TCP*.

Para injetar falhas, FIERCE adota a interface de depuração e monitoramento *JVMTI* para instrumentação das classes de comunicação responsáveis tanto pelo estabelecimento de conexões *TCP*, quanto pelo envio e recebimento de mensagens, permanecendo transparente sua ativação frente à aplicação-alvo.

Seu modelo de falhas baseia-se na utilização de códigos de erro retornados pela biblioteca de comunicação do sistema operacional, bem como no conhecimento prévio dos estados que os módulos do protocolo vigente podem assumir para o desenvolvimento de módulos de detecção de falhas de comunicação para aplicações distribuídas tolerantes a falhas, conforme proposto em (NEVES; FUCHS, 1997). Dentre os códigos de erros, FIERCE identifica quatro tipos de falhas que resultam no término de um processo com diferentes comportamentos na interface de *sockets*: término de processo, colapso, reinicialização e colapso com reinicialização.

Entretanto, assim como FIRMI, este injetor não apresenta uma arquitetura distribuída, e foca-se apenas na injeção de falhas em aplicações *Java* baseadas em *TCP*, limitando o uso desta ferramenta em ambientes distribuídos, e que independam do(s) protocolo(s) utilizado(s). Dadas as suas características, não é possível reaproveitá-lo no escopo deste trabalho.

3.2.2 Injetores de Falhas Distribuídos

Ao contrário dos injetores de falhas locais, classificam-se como injetores de falhas distribuídos aqueles que possuem algum tipo de coordenação de atividades, seja em tempo de configuração do ambiente, seja em tempo de operação dos testes.

3.2.2.1 FAIL-FCI

FAIL-FCI (HOARAU; TIXEUIL, 2005) é um *framework* voltado para avaliação de dependabilidade de aplicações distribuídas. Sua proposta consiste em dois componentes principais, (i) *FAIL* (*FAult Injection Language*), uma linguagem abstrata de alto nível para fácil construção de cenários de falhas, e (ii) *FCI* (*FAIL Cluster Implementation*), uma plataforma para injeção de falhas distribuídas onde a linguagem de entrada para a descrição de cenários de falhas é feita em *FAIL*.

A definição de cenários de falhas descreve máquinas de estado com seu respectivo modelo de ocorrência de falhas, como também a associação entre estas máquinas com um ou mais computadores componentes da rede. A plataforma *FCI*, por sua vez, é composta por três blocos, (i) um *compilador FCI* responsável pela pré-compilação dos cenários descritos em *FAIL*, que produz códigos-fonte na linguagem *C++* e arquivos-padrão de configuração, (ii) uma *biblioteca FCI*, encarregada do empacotamento e distribuição dos

arquivos gerados pelo compilador às máquinas participantes do experimento, e (iii) um *daemon FCI*, executáveis gerados por cada máquina na rede de acordo com os arquivos submetidos pela biblioteca *FCI*.

Com vias de operar em diferentes ambientes, tanto os arquivos gerados pelo compilador *FCI*, quanto os arquivos da biblioteca *FCI* são submetidos como códigos-fonte para cada máquina-alvo compilá-los segundo seu respectivo ambiente. Desta forma, *FCI* adota uma abordagem de injeção de falhas baseada em *software* de depuração. A aplicação sob testes pode ser interrompida ao executar funções específicas (ou uma determinada linha de seu código fonte), e retomada posteriormente de acordo com a descrição do cenário de falhas construído. A utilização de um *software* de depuração para injeção de falhas viabiliza a instrumentação da aplicação sob testes. Em outras palavras, é possível realizar injeção de falhas arbitrárias como a modificação do contador de programa, ou ainda das variáveis locais para emular um ataque de estouro de *buffer*, sem a necessidade de alterar o código fonte da aplicação-alvo.

Sua arquitetura distribuída e escalável (para a emulação de redes de larga escala), a partir da comunicação explícita entre *daemons FCI*, permite ao engenheiro de testes a reprodução de comportamentos específicos da aplicação-alvo difíceis (ou até mesmo impossíveis) de serem alcançados em um fluxo normal de execução. Além disso, *daemons FCI* possuem dois modos de operação, (i) aleatório, destinado a um cenário de falhas probabilístico, e (ii) determinístico, para uma reprodução com alto grau de controlabilidade do cenário de falhas.

Apesar de integrar técnicas de disparo e de injeção de falhas, e apresentar relatos utilizando apenas falhas de colapso e suspensão de processo (e não de nodo) para emulação de um nodo sobrecarregado, *FCI* adota mecanismos de depuração de aplicativos para avaliação de dependabilidade. Entretanto, para avaliação protocolos de comunicação é desejável que o injetor tenha acesso ao fluxo de comunicação, que, neste caso, é obstruído pelo *software* de depuração.

3.2.2.2 FIONA

FIONA (JACQUES-SILVA et al., 2004) (acrônimo para *Fault Injector Oriented to Network Applications*), é um injetor de falhas para validação experimental de mecanismos tolerantes a falhas existentes em aplicações distribuídas implementados em *Java*. Tal ferramenta também proporciona livre escolha de plataforma ao engenheiro de testes, pois baseia-se no protocolo *UDP*.

O modelo de injeção de falhas utiliza a interface *JVMTI* (*Java Virtual Machine Tool Interface*) para o desenvolvimento de ferramentas de depuração e monitoramento que permitem a instrumentação da classe de comunicação *UDP* responsável pelo envio e recebimento de mensagens. Tal 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. Esta abordagem proporciona transparência total entre a aplicação-alvo e o injetor de falhas, assim como sua portabilidade.

Sua extensão (JACQUES-SILVA et al., 2006) propõe uma arquitetura distribuída que permite uma configuração centralizada de múltiplos cenários de falhas, bem como suporte a mais modelos de falhas associados a sistemas distribuídos, como o particionamento de rede.

Sua arquitetura divide-se em duas partes, *local*, segmento da ferramenta que se responsabiliza pela injeção de falhas em sistemas auto-contidos; e *distribuída*, segmento usado para a condução de experimentos onde seja necessária a existência de uma controladora

de testes, pois pode afetar mais de um nodo simultaneamente. O segmento distribuído mantém a escalabilidade do sistema através de uma árvore (de nível 2) para condução de experimentos em diversos nodos. Além disso, apresenta suporte para configuração e análise de *logs* do experimento de forma centralizada, bem como a emulação de um modelo de falhas consistente com sistemas de larga escala.

Assim como FIONA, FIERCE e FIRMI também foram desenvolvidos pelo grupo de tolerância a falhas da UFRGS e voltam-se para validação experimental de apenas aplicações *Java*. Sua principal distinção está relacionada ao protocolo de comunicação adotado para efetuar a troca de mensagens (*UDP*, *TCP* e *RMI - Remote Method Invocation*, respectivamente).

3.2.2.3 LOKI

LOKI (CHANDRA et al., 2004) é um injetor de falhas para aplicações distribuídas. A ocorrência de defeitos em sistemas distribuídos e sua possível dependência de um estado global do mesmo para o disparo de falhas levou à implementação deste injetor. Entretanto, o desenvolvimento de mecanismos sofisticados baseados em um estado global do sistema requer certos cuidados, como a (i) baixa intrusividade, para não inferir na degradação de desempenho no sistema, (ii) alta precisão no disparo e análise de falhas, e (iii) alta flexibilidade em um âmbito de reprodução e metrificação de testes.

Para o seu desenvolvimento, LOKI utiliza a ideia de visão parcial do estado global do sistema, em conjunto com mecanismos de sincronização *offline*, como solução para obter uma implementação pouco intrusiva. Em outras palavras, é utilizado um algoritmo de sincronização de *clock offline* para tradução dos tempos locais de cada nodo em uma linha de tempo global.

O processo de execução, por sua vez, inclui uma máquina de estados que contém códigos responsáveis pela manutenção da visão parcial do estado global do sistema, bem como injeção falhas ao passo que o mesmo atinge um estado específico, e coleta de informações acerca de troca de estados e comportamentos do sistema frente às falhas injetadas. Desta forma, após a execução do experimento é realizada uma análise com base em dados coletados durante a execução, a fim de verificar se falhas foram injetadas adequadamente e inferiram no comportamento esperado do sistema.

A definição do cenário de falhas em LOKI é relativamente complexo. Uma linguagem de alto nível foi implementada para especificação e manutenção do estado local de máquinas participantes do experimento, definição de falhas a serem injetadas e implantação de pontos de verificação na aplicação para detecção de eventos utilizados pelos mecanismos de disparo de falhas. Não obstante, é necessária a descrição de identificadores de falhas para posterior associação com o estado global da aplicação.

LOKI é o primeiro injetor de falhas para sistemas distribuídos com mecanismos de disparo baseados em um estado global da aplicação e verificação da corretude de falhas injetadas. Entretanto, LOKI injeta falhas por instrumentação de código da aplicação-alvo. Além disso, a descrição de cenários é baseada somente em estados globais do sistema, dificultando (ou até mesmo impossibilitando) a construção de cenários mais complexos, como particionamentos de rede, injeção de falhas em cascata, entre outros.

Apesar disso, houve esforços para incorporar o suporte a falhas de particionamento de rede através de regras de *firewall* (LEFEVER et al., 2003), mas por apresentar alta intrusividade, e baixa controlabilidade no que diz respeito ao disparo temporal de falhas, LOKI apresenta abordagens inadequadas para o presente trabalho.

3.2.2.4 VirtualWire

VIRTUALWIRE (NEOGI; DE; CHIUEH, 2003) é uma ferramenta de injeção de falhas e análise de sistemas modelada para facilitar o processo de testes de protocolos de comunicação. Capaz de emular qualquer conjunto de falhas de comunicação a partir de uma abstração para uma rede virtual em cima da rede física usada em um ambiente experimental, a ferramenta visa (i) testar implementações de protocolos de comunicação, (ii) fornecer falhas reais de rede, (iii) simplificar o processo de injeção de falhas, e (iv) automatizar a interpretação de rastreamento de pacotes.

Motivado por dificuldades encontradas em testes de protocolos de comunicação, como instrumentação de códigos e rastreamento exaustivo de pacotes com analisadores de rede, VIRTUALWIRE permite ao engenheiro de testes especificar tipos de falhas a serem injetadas no sistema assim como o seu respectivo evento de disparo através de *scripts* independentes. Neste mesmo *script* responsável pela construção do ambiente de testes é possível determinar filtros para rastreamento de pacotes em tempo real, possibilitando identificar a integridade do sistema. Em outras palavras, consiste na verificação de falhas e o respectivo comportamento do sistema.

Com o objetivo de manter o sistema operacional intacto, VIRTUALWIRE adota uma infraestrutura baseada em módulos. Neste, mecanismos de injeção de falhas e análise de tráfego operam através do *framework* Netfilter (RUSSEL; WELTE, 2002), que responsabiliza-se pelo redirecionamento de pacotes para posterior classificação de acordo com o cenário especificado. Já sua natureza distribuída torna-se efetiva através da implementação de um protocolo de plano de controle que gerencia o injetor presente em múltiplos nodos, caracterizando sua alta escalabilidade.

Por outro lado, assim como ORCHESTRA, VIRTUALWIRE não suporta a descrição de falhas rotineiras em ambientes de comunicação, como o particionamento de rede, alvo deste trabalho. Além disso, mecanismos de disparo de falhas baseados em temporização não estão presentes nesta ferramenta, inviabilizando a realização de experimentos em ambientes móveis.

3.3 Conclusões de Capítulo

Este capítulo apresentou um estudo acerca dos mais variados tipos de injetores de falhas de comunicação encontrados na literatura.

Embora todas as ferramentas caracterizem-se por injetores de falhas de comunicação, este trabalho foca em um tipo muito específico de falhas de comunicação, o de particionamento de rede.

Apesar de ser possível reproduzir este tipo de falha com alguns dos injetores listados neste capítulo, estes não foram desenvolvidos para este propósito, de tal forma que sua extensão para este tipo de falha dificulta, ou até mesmo limita o engenheiro de testes no que diz respeito à descrição de cenários de falhas, e o suporte a protocolos de rede específicos.

Entretanto, algumas metodologias apresentadas na literatura podem ser reaproveitadas no escopo deste trabalho, seja pelos mecanismos de interceptação de mensagens apresentados (ACRI, 2010; DREBES, 2005; NEOGI; DE; CHIUEH, 2003), seja pelos métodos de coordenação de atividades adotados (HOARAU; TIXEUIL, 2005; JACQUES-SILVA et al., 2006; LEFEVER et al., 2003).

O capítulo a seguir descreverá a proposta deste trabalho, que consiste em um mecanismo voltado unicamente para a injeção de falhas de particionamento de rede em aplica-

ções distribuídas.

4 AMBIENTE DE INJEÇÃO DE FALHAS PIE

Para injetar falhas de comunicação corretamente em aplicações-alvo, um injetor de falhas adequado deve obedecer uma série de requisitos. Do contrário, a análise comportamental doa aplicação-alvo frente à sua especificação sob falhas de particionamento pode levar a conclusões preciptadas.

Dentre os principais requisitos, destacam-se:

- Baixo grau de interferência no desempenho da aplicaçã-alvo;
- Independência de topologias e protocolos de rede;
- Suporte a coordenação de experimentos;
- Suporte a testes de caixa preta;
- Suporte a métodos simples, mas poderosos, de descrição de falhas.

Seguindo os requisitos identificados para solução, garante-se que um ambiente de injeção de falhas tenha (*i*) maior aplicabilidade, dada sua independência de topologias de rede e protocolos de comunicação, (*ii*) maior usabilidade, com uma linguagem simples e de fácil aprendizado para descrição de falhas, e (*iii*) maior integridade, uma vez que um baixo grau de interferência, em conjunto com um módulo de coordenação de experimentos, não altera a carga de trabalho da aplicação-alvo. Logo, nenhuma interferência compromete a integridade da campanha de teste em andamento.

Entretanto, as ferramentas estudadas no capítulo anterior não estão adequadas ou oferecem suporte à extensões para injeção de falhas de particionamento de rede. Logo, este capítulo descreve a arquitetura passo-a-passo acerca do desenvolvimento de todos os componentes que integram o novo ambiente de injeção de falhas *PIE – Partitioning Injection Environment*.

4.1 Emulação de Particionamento

Para emular corretamente falhas de particionamento de rede, neste trabalho, foi adotado um modelo de descarte de mensagens a partir do nodo receptor. Em outras palavras, quando um nodo receber uma determinada mensagem, a mesma só será entregue à aplicação-alvo se o emissor desta mensagem estiver presente na mesma partição do nodo receptor.

A figura 4.1 apresenta o sistema adotado para emulação de particionamentos de rede. Assim como em um ambiente real, as tentativas de troca de mensagens entre nodos durante uma falha de particionamento de rede não são identificadas no envio, mas sim no recebimento da mensagem pelo nodo receptor.

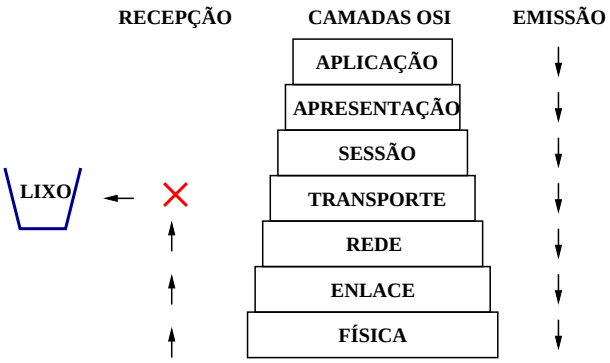

Figura 4.1: Sistema de emulação de particionamentos de rede adotado.

4.2 Descrição de Particionamento de Rede

Existem diversas maneiras para descrição de cargas de falhas. Contudo, cada injetor de falhas tende a focar em um pequeno grupo de falhas, tornando-se mais conveniente a adoção de métodos simples e objetivos.

Apesar de simples, uma linguagem voltada para descrição de cargas de falhas deve ser robusta o suficiente para prover ao engenheiro de testes a criação de um ambiente de acordo com seus requisitos. Do contrário, a má definição de primitivas, seja por erro do injetor de falhas, ou do engenheiro de testes, pode comprometer o sistema de modo a gerar estados inesperados.

Portanto, deseja-se uma linguagem com um alto nível de abstração e sem limitação das ações do engenheiro de testes na especificação da carga de falhas do experimento. Desta forma, para fins de simplicidade e facilidade, este trabalho considera uma visão de nodos em um sistema distribuído como grupos de elementos.

Conforme a figura 4.2, quando ocorre o particionamento de rede, estes grupos de nodos são divididos em diversos subgrupos (partições); quando o particionamento de rede é desfeito, alguns subgrupos unem-se e formam um único grupo (partição). Logo, para este trabalho, foi definida a representação de um sistema distribuído com base na teoria de conjuntos, ou seja, a partir de grupos e subgrupos de nodos, provendo uma forma simples de ler e escrever cargas de falhas.

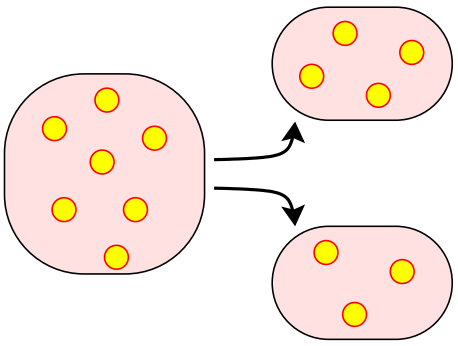

Figura 4.2: Partições representadas como grupos de nodos.

Supondo um sistema distribuído com 7 nodos (N1 até N7) que sofre um particionamento de rede associado a um tempo T e divide a rede em dois subgrupos, um deles contendo 3 nodos (N1, N3 e N6), e o outro com os 4 nodos restantes (N2, N4, N6 e N7). Tal associação temporal ao evento de particionamento permite ao engenheiro de testes

a realização de experimentos com comportamento dinâmico e bem definido, e pode ser simplesmente representado por:

$$(T)\ \{\text{N1, N3, N5}\}\ \{\text{N2, N4, N6, N7}\}$$

Conforme apresentado, o disparo de falhas de particionamento de rede segue uma linha temporal bem definida pelo engenheiro de testes, onde este pode definir quando (tempo) e qual estado o ambiente passa a assumir. Apesar de outros injetores adotarem outros métodos para o disparo de falhas, seja de acordo com o conteúdo de uma mensagem, seja de acordo com o número de mensagens recebidas, tais métodos não se aplicam em falhas de particionamento.

Para emulação adequada de falhas de particionamento de rede, os nodos integrantes do experimento devem apresentar a mesma visão da rede durante o disparo de eventos pois, caso contrário, mensagens podem chegar ao seu destino indevidamente, e podem comprometer uma injeção de falha de particionamento de rede efetiva. Em outras palavras, quando um evento é disparado com base em atributos atemporais, o comportamento de um nodo durante o disparo de eventos pode não ser o comportamento esperado por outro nodo no mesmo instante, caracterizando uma violação. Logo, este trabalho assume características temporais com o objetivo de reduzir o impacto destes intervalos entre nodos para o disparo de eventos durante a realização de experimentos.

A figura 4.3 ilustra as dificuldades mencionadas em utilizar características não baseadas em tempo para o disparo de eventos de falhas. Suponha dois nodos, *N1* e *N2*, que executam injetores de falhas que disparam suas ações de acordo com o número de mensagens recebidas (*m1*, *m2* e *m3*). Dentro de um intervalo de tempo qualquer, o nodo *N1* recebeu três mensagens e o injetor de falhas associado ao mesmo identifica e efetua o disparo do evento *E0*, particionando ambos nodos. A partir de então nenhuma mensagem enviada pelo nodo *N1* (*m2*, *m3* e *m4*) será entregue ao nodo *N2*, e o mesmo comportamento se espera do nodo *N2* em relação à *N1*. Entretanto, o nodo *N2* ainda não satisfez as condições necessárias para o disparo do evento *E0* e permanece entregando mensagens (*m4*) ao nodo *N1*, violando propriedades para uma injeção de falha de particionamento adequada. Daí por diante o problema seria se manteria até o momento em que todos os eventos sejam satisfeitos e disparem suas ações, dificultando a estimativa de tempo de duração da campanha de testes antes de sua execução.

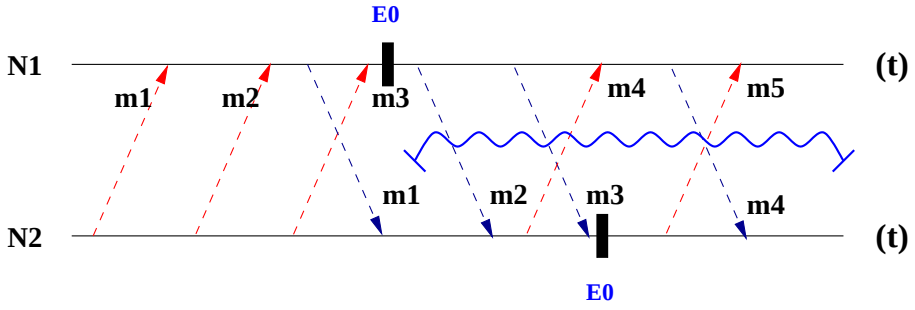

Figura 4.3: Disparo de falhas atemporal.

Portanto, tais abordagens apresentam-se insuficientes para a emulação íntegra de particionamentos de rede, sustentando a adoção de eventos baseados em tempo.

Definida a premissa para descrição de eventos de particionamento de rede, resta a declaração dos nodos que compõe a rede. Tal declaração estática implica na proibição de

entrada e saída de nodos durante a realização do teste, podendo limitar o uso do ambiente de injeção de falhas *PIE* em ambientes dinâmicos, como redes móveis. Entretanto, tratar este tipo de comportamento do ambiente foge ao escopo deste trabalho e será focado em trabalhos futuros.

4.3 Arquitetura do Ambiente de Injeção de Falhas *PIE*

A figura 4.4, proposta por Hsueh, Tsai e Iyer (1997), apresenta um ambiente de injeção de falhas ideal composto de uma aplicação-alvo de testes, um injetor de falhas, uma biblioteca de falhas, um gerador de carga de trabalho, um coordenador de experimentos, um monitor de execução e, por fim, um coletor/analisador de *logs*.

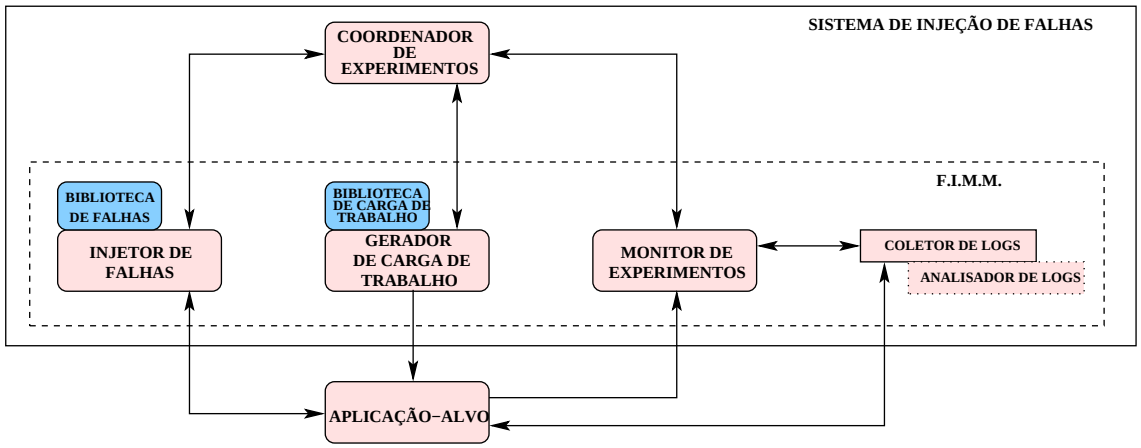

Figura 4.4: Componentes básicos de um ambiente de injeção de falhas proposto por Hsueh, Tsai e Iyer.

A aplicação-alvo refere-se diretamente à aplicação sob testes que opera em um ambiente de injeção de falhas. Este ambiente compõe-se basicamente de 5 módulos: (*i*) um injetor de falhas, responsável pela injeção de falhas na aplicação-alvo de acordo com as falhas suportadas em sua biblioteca de falhas, (*ii*) um gerador de carga de trabalho, que dispõe de tarefas (*benchmarks*, entre outros) que são executadas pelo injetor de falhas, (*iii*) um monitor de experimentos, para fins de rastreamento das atividades executadas pelo injetor de falhas e, (*iv*) um coletor de logs, para armazenar qualquer tipo de informação que o módulo de monitoramento julgar necessário, e (*v*) um analisador de logs, para análise e avaliação das informações coletadas.

Existe ainda um 6^o módulo nesta arquitetura. Trata-se do coordenador de experimentos, uma peça importante em um ambiente de injeção de falhas, podendo operar no sistema sob teste, ou ainda em um nodo separado. Tal abordagem, além de facilitar o entendimento, a separação de componentes permite maior flexibilidade e portabilidade do ambiente para outros sistemas.

Dada a importância da arquitetura proposta e sua aceitação pela comunidade na última década, neste trabalho segue-se o mesmo padrão de Hsueh, Tsai e Iyer com pequenas adaptações, pois *PIE* volta-se para ambientes distribuídos. Suponha os módulos destacados entre o retângulo pontilhado na figura 4.4, exceto o módulo responsável pela análise de *logs*, como um único componente, chamado *F.I.M.M.* (*Fault Injection and Monitoring Module*). O módulo de análise de *logs*, por sua vez, é agregado ao módulo de coordenação de experimentos. Isto, pois, a análise completa de *logs* da campanha de testes é

centralizada e realizada após o encerramento dos testes, ou seja, quando cada nodo envia seus *logs* para o coordenador.

A figura 4.5 apresenta uma visão do modelo proposto por este trabalho, destacando *F.I.M.M.* e sua instanciação em cada nodo que compõe a aplicação-alvo.

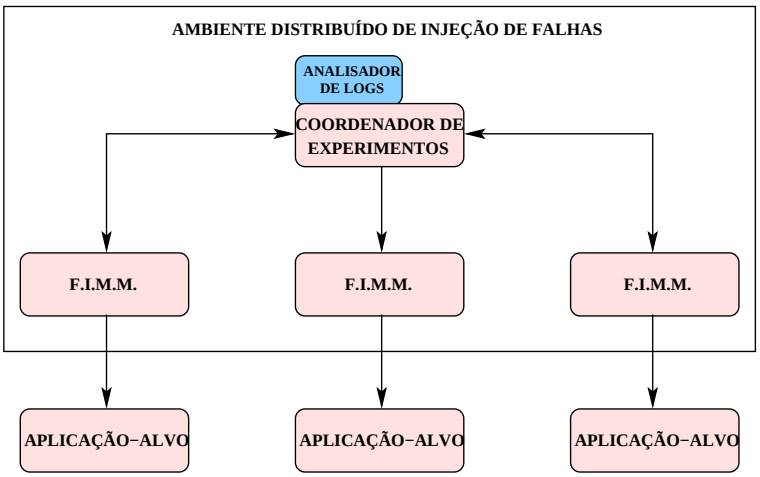

Figura 4.5: Proposta de ambiente distribuído de injeção de falhas.

Apresentadas os módulos que compõem a arquitetura do ambiente de injeção de falhas *PIE*, as próximas seções descrevem as especificações de cada componente e suas justificativas para a adoção dos métodos empregados na construção do protótipo proposto neste trabalho.

4.3.1 Biblioteca de Falhas

O módulo de biblioteca de falhas agrega 3 funções principais:

- Análise sintática: verifica a estrutura da carga de falhas, construída pelo engenheiro de testes, em busca de possíveis erros de sintaxe não previstos pelo interpretador de comandos;
- Análise semântica: verifica a consistência da carga de falhas, desde a declaração dos nodos sob teste, intervalos de tempo entre eventos, até particionamentos inconsistentes (um nodo particionado com ele mesmo, por exemplo).
- Interpretação: a única tarefa realizada pela biblioteca de falhas em tempo de execução do experimento, e mais importante. Esta função identifica instantes em que eventos devem ser disparados e suas respectivas ações.

Para prover suporte a estas funções, os princípios estudados neste trabalho, para uma descrição adequada de falhas, levaram ao desenvolvimento de uma linguagem para descrição de cargas de falhas voltadas para falhas de particionamento de rede.

A figura 4.6 apresenta um exemplo de carga de falhas descrita a partir da linguagem desenvolvida para o ambiente de injeção de falhas *PIE*. Sua descrição é bastante simples e inicia com o mapeamento estático dos nodos participantes do experimento, na linha 1.

As linhas 2 e 15 definem, respectivamente, cláusulas que demarcam o início (*START*) e encerramento de atividades (*STOP*). As linhas 3 até 5 formam blocos, que descrevem quando (em segundos decorridos após o início da campanha de testes) e qual visão de rede o nodo apresentará, particionada (linhas 4, 7 e 10), ou em *merge* (linha 13).

```
1:  @declare { N1, N2, N3, N4, N5, N6, N7}          mapeamento estático dos nodos
2:  START:                                          demarca o início da execução
3:     after (60s) do
4:        partition {N1, N2, N3} {N4, N5, N6, N7};
5:     end
6:     after (120s) do                              tempo para disparar evento
7:        partition {N1, N3} {N2, N4, N6} {N5, N7}; visão de rede
8:     end                                          fim de evento
9:     after (180s) do
10:       partition {N1, N7} {N2, N3, N4} {N5, N6};
11:    end
12:    after (240s) do
13:       partition {N1, N2, N3, N4, N5, N6, N7};
14:    end
15: STOP:                                           demarca o término da execução
16:    after (300s);                                tempo de duração do experimento
```

Figura 4.6: Exemplo de aplicação da linguagem desenvolvida.

Ressalta-se que, durante a descrição de falhas de um experimento, o engenheiro de testes tem livre arbítrio para definir se o sistema sofre o *merge* para retornar ao seu estado inicial, onde todos os nodos comunicam-se entre si. Entretanto, não realizar o *merge* não implica na permanência dos nodos particionados de acordo com sua última visão da rede, pois após o tempo definido para o término do experimento, o injetor não atua mais sobre o canal de comunicação e os nodos voltam a comunicar-se normalmente.

4.3.2 Injetor de Falhas

Particionamentos de rede não diferem de outros tipos de falhas de comunicação, como descarte e atraso de mensagens, exceto por sua complexidade, uma vez que ocorrem em sistemas distribuídos e atingem muitos nodos. Logo, para injeção de falhas de particionamento de rede, previamente especificados na carga de falhas pelo engenheiro de testes, alguns requisitos devem ser considerados.

Conforme visto, para aumentar sua aplicabilidade, um injetor de falhas deve suportar uma grande variedade de protocolos de rede, bem como ser independente da linguagem em que a aplicação-alvo de testes fora desenvolvida. Isto, pois, muitos SWIFI voltam-se para sistemas específicos, seja pelo protocolo, seja pela linguagem de desenvolvimento da aplicação-alvo. Além disso, o impacto sobre a aplicação-alvo deve ser mínimo. Em outras palavras, não se deve causar um aumento de carga sobre a aplicação-alvo, evitando interferências que podem levá-la a comportamentos inesperados.

Para atender tais requisitos, diversas técnicas para interceptação de mensagens no canal de comunicação podem ser identificadas na seção 2.3. A interceptação de chamadas ao sistema, por exemplo, permite a emulação de situações de falhas através da sobreposição da chamada padrão, forçando o sistema a operar sob situações críticas. Entretanto, esta abordagem apresenta um alto grau de intrusividade no que se refere à degradação

de desempenho da aplicação-alvo, uma vez que para cada chamada ao sistema o injetor deveria verificar se corresponde à chamada desejada.

Outra possibilidade de injeção de falhas de particionamento é através de um *firewall*, que identifica o fluxo de mensagens e atua de acordo com as ações estabelecidas na carga de falhas. Contudo, além da baixa portabilidade e intrusividade gerada no sistema ao ficar manipulando a tabela de filtros do *firewall*, a complexidade do injetor de falhas também seria elevada, uma vez que cuidados especiais devem ser considerados ao manipular a sua tabela de filtros, entre outros.

É possível também encontrar trabalhos que utilizem camadas extras na pilha de protocolos do sistema operacional. Esta abordagem consiste na inserção de uma camada entre duas camadas da pilha de protocolos. Ou seja, uma camada de injeção de falhas é inserida abaixo da camada onde reside o protocolo alvo e ativa filtros de manipulação com o objetivo de reproduzir as ações previstas na carga de falhas.

Portanto, dentre os mecanismos buscados para solução desta proposta, *FIMM* adota uma técnica que consiste em operar dentro do núcleo do sistema operacional como um módulo de interceptação de mensagens. Assim como *FIRMAMENT* e *VirtualWire*, *PIE* possui um módulo, presente em cada nodo participante do experimento, que se registra na pilha de protocolos do sistema para ter acesso aos fluxos de entrada e saída de pacotes.

Desta forma, o protocolo utilizado pela apliação-alvo, ou sua linguagem de desenvolvimento não limitam a aplicabilidade do ambiente *PIE*, visto que as mensagens chegam ao seu destino após serem (possivelmente) manipuladas. E, além de possibilitar a execução de testes de caixa preta, reduz a perturbação sobre o desempenho da aplicação-alvo, pois intercepta mensagens diretamente na pilha de protocolos do sistema operacional por meio de funções privilegiadas.

4.3.3 Carga de Trabalho

O módulo de carga de trabalho, mais especificamente, refere-se a procedimentos de automação de testes e devem ser criados pelo engenheiro de testes e agregados ao ambiente de injeção de falhas para operarem simultaneamente.

A carga de trabalho propriamente dita, diz respeito a todo fluxo de troca de mensagens entre nodos durante a execução de uma campanha de testes.

Tratando-se de falhas de comunicação, normalmente um gerador de carga de trabalho responsabilizaria-se pela geração desta carga. Entretanto, alguns experimentos requerem cargas de trabalho específicas, seja pela interação humana com a aplicação-alvo, seja com a adição de *scripts* de automação, e foge do escopo deste trabalho agregar tal funcionalidade.

4.3.4 O Coordenador de Experimentos

Tratando-se de sistemas distribuídos, uma das grandes dificuldades encontradas pelos engenheiro de teste é a distância física entre nodos do sistema. Além disso, injeção de falhas de particionamento de rede não depende de apenas um nodo, mas de todos sob testes.

Partindo desta premissa, e a necessidade de disparar ações baseadas em tempo, faz-se necessária uma coordenação entre nodos para que todos possuam a mesma visão de rede simultaneamente. Em outras palavras, em um certo tempo t, o nodo *N1* está particionado do nodo *N2* que, por sua vez, deve estar particionado do nodo *N1* no tempo t.

Dada a necessidade de coordenação de atividades entre nodos, uma solução baseada em um centralizador de experimentos apresenta-se bastante satisfatória, visto que o enge-

nheiro de testes não percorrerá nodo a nodo do sistema para interagir com o experimento. Com isso, toda e qualquer ação pode ser realizada a partir de um único nodo e, quando necessário, repassada aos demais.

Da mesma forma dos outros módulos, a inserção de um módulo de coordenação de experimentos visa auxiliar o engenheiro de testes a um baixo custo para a campanha de testes, seja não causando degradação de desempenho, seja alterando a carga de trabalho da aplicação-alvo.

Entretanto, adotar um coordenador de atividades requer alguns cuidados, uma vez que a delegação de tarefas em tempo de execução podem ser descartadas indistintamente pelos nodos, durante um particionamento de rede.

A figura 4.7 ilustra o modelo descrito, em que um dos 3 nodos do experimento assumiria o controle sobre o experimento e delegaria funções aos demais nodos (*N1* e *N2*). O experimento inicia com a troca de mensagens entre nodos (setas pontilhadas), proveniente da carga de trabalho da aplicação-alvo. Em um determinado tempo *t*, o coordenador de atividades identifica um evento a ser disparado e o repassa para os demais nodos. A partir de então o ambiente assume o estado *s1*, fragmentando a rede em três partições isoladas (separadas por cosenoides horizontais), de forma que nenhum nodo está ao alcance de outro. Quando o nodo coordenador do experimento identificar um novo evento, suas mensagens de controle aos demais nodos serão interceptadas pelo módulo *F.I.M.M.* e serão descartadas, assim como as mensagens integrantes da carga de trabalho da aplicação-alvo. Por fim, a troca de estado desejada (*s1–>s2*) não produz efeito algum, visto que as mensagens de controle não chegaram ao seu destino, deixando o ambiente de testes no estado *s1* permanentemente.

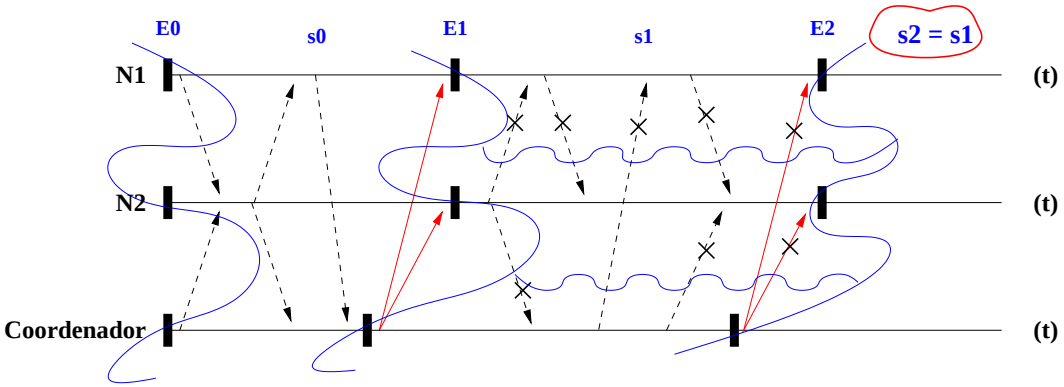

Figura 4.7: Interceptação de mensagens do coordenador de atividades.

Desta forma, *PIE* adota um modelo onde não há alteração da carga de trabalho da aplicação-alvo. Após iniciado o experimento, cada nodo irá reproduzir seu comportamento especificado na carga de falhas de acordo com o seu relógio local até o término da execução. Entretanto, sabe-se que a ocorrência de eventos em sistemas distribuídos pode comprometer a integridade dos testes em virtude das diferentes variações de relógio entre cada nodo, inferindo no disparo de ações em intervalos de tempo distintos dos previstos na carga de falhas.

A variância de relógios em tempo de execução poderia ser facilmente sanado a partir da inclusão de mecanismos de sincronização de relógios. Uma alternativa seria centralizar o experimento em um nodo coordenador do ambiente, onde a troca de estados do sistema seria delegada em tempo de execução por apenas um nodo, mas acarreta em outras limitações, conforme visto na figura 4.7, onde ocorre interferência na carga de trabalho da

aplicação-alvo.

Para contornar este cenário, *PIE* propõe um modelo parcialmente coordenado. Em outras palavras, antes do início da execução de testes, o módulo de coordenação assume que o estado atual de cada nodo é também o seu respectivo estado inicial, objetivando definir um ponto de sincronia entre os nodos. Desta forma, é possível diminuir o impacto das variações de relógio durante o disparo de eventos descritos na carga de falhas.

O modelo adotado por *PIE* pode ser visto na figura 4.8, onde os nodos (*N1*, *N2* e *N3*) iniciam o experimento em seu estado *s0*. Durante um intervalo de tempo *t* os nodos seguem trocando mensagens referentes a carga de trabalho da aplicação-alvo. Entretanto, ao se aproximar dos intervalos de tempo marcados para disparo de eventos do sistema, cada nodo identifica sua troca de estado em intervalos de tempo potencialmente diferentes em virtude das variações de relógio.

Após a identificação e, posterior, disparo do evento *E1*, o nodo *N1* assume o estado *s1*, seguido pelo nodo *N3* e, ainda mais adiante o nodo *N2*. Neste estado, o ambiente é particionado de forma que o nodo *N3* encontra-se isolado em uma partição, tendo suas mensagens da carga de trabalho descartadas pelo injetor de falhas (setas pontilhadas).

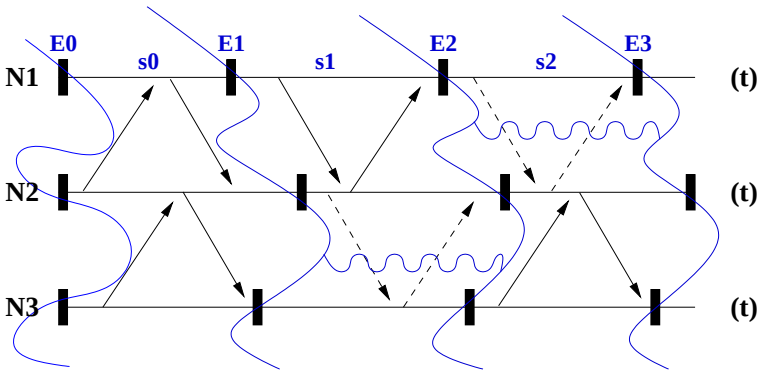

Figura 4.8: Injeção de falhas com processamento local.

Ao contrário do modelo anterior (figura 4.7), embora em instantes diferentes de tempo, os nodos seguem seu processamento local e, quando identificado, um novo evento (*E2*) é disparado e o ambiente passa para o estado *s2* e os nodos seguem sob testes e processam a carga de falhas até o disparo que identifica seu encerramento (*E3*).

Concluída a execução do experimento, o injetor passa para a etapa de coleta de *logs*. Nesta, o módulo de coordenação de testes volta a efetuar a troca de mensagens com os demais nodos para efetuar a coleta e análise das informações obtidas.

Apesar de prover um ambiente sem sincronização entre nodos para que as aplicações-alvo possam ser testados em ambientes o mais realísticos possíveis, ressalta-se que este injetor não limita a realização de testes de aplicações que, em sua implementação, adotem mecanismos de sincronização de relógio ou afins.

4.3.5 Monitor de Experimentos e Coletor de *Logs*

Visto que um coordenador de dados não pode delegar tarefas para os nodos em tempo de execução, os eventos de particionamento de rede são disparados localmente por cada nodo. Desta forma, o módulo de monitoramento de execução, presente em cada nodo, visa identificar a ocorrência de eventos. Uma vez identificado, a informação é passada para o módulo de coleta que, por sua vez, registra tais informações para posterior identificação de características importantes acerca do experimento.

As principais informações coletadas incluem (*i*) o índice de variação de relógio do início ao fim do experimento, bem como no intervalo entre o disparo de eventos, e (*ii*) o número de mensagens trocadas ao longo do experimento.

Com estas informações, é possível identificar se, durante a execução do experimento, a injeção de particionamento foi correta.

4.3.6 Analisador de *Logs*

Ao fim de cada experimento, todos os dados coletados pelo módulo de coleta são submetidos a um módulo central de análise de dados. Esta análise objetiva identificar a corretude das falhas de particionamento injetadas e, caso sejam detectados problemas, cabe ao engenheiro de testes julgar a necessidade de realizar um novo teste.

Os problemas mencionados podem existir em decorrência das diferentes variações de relógio. Em outras palavras, um evento disparado em um tempo *t* no nodo *N1* pode ser disparado no nodo *N2* em um tempo *t+1*. Esta divergência de estados, em um intervalo de milissegundos, poderia resultar em uma violação de propriedade do particionamento de rede, uma vez que podem haver trocas de mensagens entre nodos particionados.

Logo, *PIE* apresenta propostas para identificação deste problema, que será retomado no próximo capítulo.

4.4 Conclusões de Capítulo

Este capítulo apresentou a arquitetura utilizada pelo novo ambiente de injeção de falhas *PIE*, justificando cada um dos módulos integrantes e suas limitações.

Os estudos realizados para definição desta proposta, junto às experiências exploradas por outros membros da comunidade permitiram o desenvolvimento de um novo ambiente de injeção de falhas. *PIE*, objetiva avaliar aplicações distribuídas sem interferir na carga de trabalho e no desempenho de aplicações-alvo. Além disso, a modularidade da arquitetura proposta, bem como dos mecanismos utilizados para o desenvolvimento deste ambiente, facilitam a realização de testes de caixa preta.

O capítulo a seguir apresenta especificações do protótipo *PIE*, ilustrando seu fluxo de operação, e características importantes de implementação.

5 O PROTÓTIPO PIE

Este capitulo aborda características de implementação e funcionamento do ambiente de injeção de falhas *PIE*. Atualmente, *PIE* pode ser utilizado em plataformas **nix* com versão de núcleo igual ou superior a 2.4, pois a partir desta que a interface de programação *Netfilter* foi disponibilizada. Além disso, é necessária a instalação de dois *softwares* sobre os quais a biblioteca de falhas foi implementada, *lemon* (HIPP, 2008) e *re2c* (BUMBULIS; COWAN, 1993).

Salienta-se também que, tratando-se do desenvolvimento de um ambiente para emulação de injeção de falhas de particionamento de rede, é necessário avaliar a corretude das falhas injetadas. Isto, pois, a implementação de um emulador está suscetível à condições inexistentes em particionamentos de rede em um ambiente real. Logo, as seções deste capítulo que abordam tópicos de violação em particionamentos tratam explicitamente das dificuldades encontradas para injetar falhas de particionamento, e não das possíveis violações causadas pelo impacto da propagação da falha.

5.1 Fluxo de Atividades do Protótipo PIE

A realização de experimentos utilizando o protótipo *PIE* é dividida em três etapas: *configuração*, *operação* e *coleta de dados*.

A figura 5.1 ilustra a primeira etapa, onde o engenheiro de testes inicia a construção da carga de falhas, ou seja, descreve o ambiente desejado sobre o qual a aplicação-alvo será testada. Em outras palavras, é nesta etapa em que se define quantos e quais nodos estarão particionados em um determinado instante de tempo (figura 4.6). Concluída esta etapa, resta ao engenheiro de testes submeter sua carga de falhas para que o ambiente *PIE* possa operar automaticamente.

Antes de iniciar a preparação do ambiente para execução da campanha de testes, a carga de falhas é analisada sintática e semanticamente, pelo *parser* (biblioteca de falhas), a fim de evitar que erros sejam detectados somente em tempo de execução do experimento, o que pode causar efeitos colaterais no comportamento do protótipo *PIE*. Caso existam problemas na carga de falhas, o erro é reportado ao engenheiro de testes para correção, caso contrário a etapa de configuração segue adiante.

Com a carga de falhas pronta, o módulo de coordenação de experimentos realiza a distribuição desta carga para todos os nodos integrantes do experimento, referenciados por círculos (figura 5.1), e aguarda até que todos confirmem o recebimento completo das informações. Após a confirmação, o coordenador envia uma instrução em *broadcast* para os nodos informando acerca do início da execução do experimento, garantindo que o os nodos iniciem sua execução praticamente ao mesmo tempo.

Ainda nesta etapa, juntamente com a carga de falhas definida pelo engenheiro de

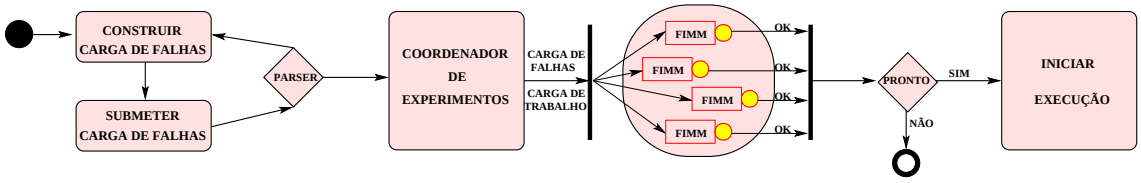

Figura 5.1: Etapa de configuração do ambiente de injeção de falhas *PIE*.

testes, deve ser definida uma carga de trabalho para ser distribuída entre os nodos do experimento. Normalmente, esta carga de trabalho trata-se de uma instância da aplicação-alvo executando localmente em cada nodo, ou ainda instruções remotas solicitadas por um nodo centralizador.

Já na etapa de operação, a execução do experimento dá-se de forma automatizada até seu encerramento, visto que não há necessidade de interação entre o engenheiro de testes e o protótipo *PIE*. Entretanto, isto não quer dizer que esta etapa possa ser ignorada pelo engenheiro de testes, pois é nela que a aplicação-alvo apresentará comportamentos anômalos, ou soluções de contorno para a ocorrência de falhas de particionamento de rede. Por tratar-se de uma seção crítica do experimento, é desejável que a aplicação-alvo possa operar livre de mensagens externas à ele, ou seja, sem a presença de mensagens que não compõem a carga de trabalho original da aplicação-alvo.

O módulo de injeção de falhas atua sobre o fluxo de mensagens da aplicação trocado entre os nodos, visando reproduzir os diferentes comportamentos do ambiente anteriormente descritos na carga de falhas. Para ter acesso à carga de trabalho da aplicação-alvo, *PIE* implementa um módulo específico (*F.I.M.M.*), presente em cada nodo do experimento, que intercepta as mensagens e age de acordo com as ações especificadas em sua carga de falhas. Logo, é possível definir *quando* um evento de particionamento irá ocorrer e *quais* nodos pertencerão a cada partição, conforme ilustrado na figura 5.2.

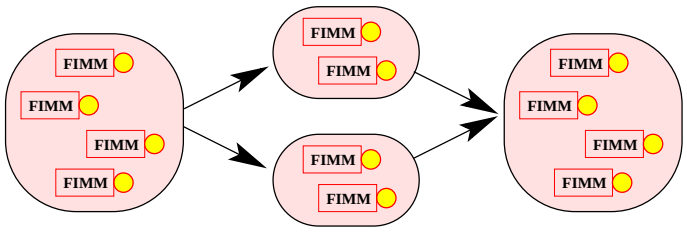

Figura 5.2: Etapa de operação do ambiente de injeção de falhas *PIE*.

Quando o módulo *F.I.M.M.* identificar o fim da execução de testes, *PIE* segue para a etapa de coleta de dados (figura 5.3). Nela, todos os registros de log gravados pelo módulo de monitoramento em cada nodo são enviados para o módulo de coordenação de experimentos que, por sua vez, submete tais informações a um módulo de análise de logs. Este módulo de análise tem por objetivo encontrar violações que comprometam a integridade de resultados experimentais e, consequentemente, da avaliação do engenheiro de testes.

Estas violações caracterizam-se pelo fluxo de mensagens em pontos críticos do disparo de falhas. Em outras palavras, as diferentes taxas de variação de tempo nos nodos implicam, em uma determinada altura do experimento, em um disparo de falhas não simultâneo. Durante este pequeno intervalo existente entre os disparos de falhas de um nodo e outro, podem trafegar mensagens indevidas. Logo, é possível que este comporta-

mento, inexistente em um particionamento de rede real, afete o experimento e prejudique o resultado do mesmo.

Entretanto, nem sempre este tipo de violação invalida o experimento e, por essa razão, o protótipo *PIE* responsabiliza-se pela identificação destas violações e geração de relatórios para o engenheiro de testes, que decidirá o grau de criticidade e necessidade de uma nova execução da campanha.

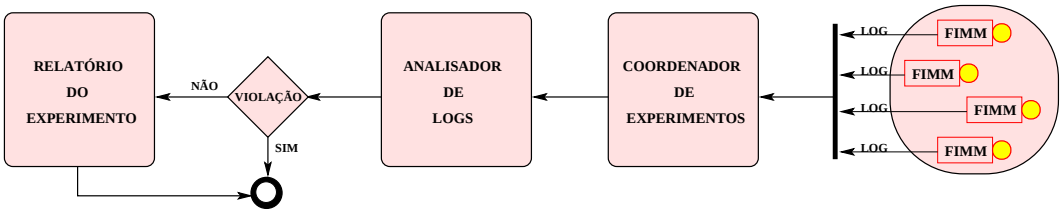

Figura 5.3: Etapa de coleta de dados do ambiente de injeção de falhas *PIE*.

Tratando-se de um componente bastante crítico no ambiente *PIE*, seu desenvolvimento será detalhado nas próximas seções.

5.2 Consistência de Particionamentos de Rede

A injeção de falhas de particionamento de rede deve ser tratada cuidadosamente. Para que seja possível emular este comportamento é necessário garantir que algumas propriedades sejam satisfeitas. Um particionamento de rede caracteriza-se pela divisão da rede em sub-redes. Logo, a tentativa de troca de mensagens entre nodos presentes em sub-redes distintas deve ser negada explicitamente. Além disso, em um ambiente real, sabe-se que a visão de rede dos nodos não são detectadas ao mesmo tempo. Ou seja, um nodo *N1* pode estar particionado de um nodo *N2* enquanto um nodo *N2* não está particionado de um nodo *N1*.

Para satisfazer tais propriedades, *PIE* adota características baseadas em tempo para efetuar o disparo de eventos de falha. Embora as diferentes variações de relógio de cada nodo possam inferir em disparos de eventos em instantes diferentes de tempo, a adoção de disparo de falhas baseados em tempo visa diminuir o intervalo entre o disparo de falhas propostos por outros mecanismos.

Nestes intervalos de tempo entre o disparo de falhas é possível que mensagens possam ser entregues indevidamente. Em outras palavras, uma mensagem enviada durante um determinado estado é recebida pelo nodo receptor em um estado distinto. Este comportamento caracteriza uma violação de integridade do experimento, visto que as mensagens trocadas nestes intervalos de tempo podem comprometer a análise de resultados experimentais.

Apesar de ser praticamente impossível evitar a ocorrência de comportamentos desta natureza, foi desenvolvido para o ambiente de injeção de falhas *PIE* um mecanismo de detecção desta violação. Tal abordagem implementa um sistema de contadores, onde a quantidade de mensagens enviadas de um nodo *N1* para um nodo *N2* deve ser igual ao número de mensagens recebidas de um nodo *N2* por um nodo *N1*. Do contrário, a emulação não foi bem sucedida.

5.2.1 Detecção de Violações

Técnicas voltadas para a identificação de violações são importantes em sistemas distribuídos, pois identificam pontos-chave onde o sistema apresenta comportamentos que fogem de sua especificação.

Definir condições de violação não é uma tarefa trivial, visto que estas alteram de acordo com a técnica utilizada para o monitoramento da execução, como também podem não cobrir todos os tipos de violações desejáveis. Para chegar até a solução adotada neste trabalho, diversas técnicas foram identificadas e estudadas. Entretanto, nem todas se aplicam a sistemas distribuídos. Em Elnozahy et al. (2002) são apresentadas técnicas para *rollback-recovery* em sistemas baseados em troca de mensagens. Contudo, apenas as baseadas em logs permitem modelar a execução de testes como uma sequência de intervalos de estado determinísticos (STROM; YEMINI, 1985).

Para este trabalho, optou-se por uma simplificação das técnicas baseadas em logs apresentados em Juang e Venkatesan (1991). Logo, o conceito de contadores de mensagens para recuperação de *crashes* de *link* foi adotado. Em um primeiro momento, pensou-se em integrar contadores de estados em campos não utilizados dos pacotes TCP/IP trafegantes na rede, mas esta solução tornou-se inviável, dada a necessidade de alteração da carga de trabalho, bem como a existência de aplicações e sistemas distribuídos que utilizam os mesmos.

Desta forma, um novo modelo de contadores, para verificação de violação, foi definido com o propósito de diminuir a intrusividade. Assim como citado acima, a idéia é baseada em um algoritmo clássico de recuperação (JUANG; VENKATESAN, 1991), onde a violação pode ser identificada através da quantidade de mensagens trocadas entre os nodos participantes do experimento. Logo, cada nodo mantém um contador de mensagens enviadas para cada receptor distinto no sistema. Além disso, cada nodo também mantém um contador de mensagens recebidas de cada nodo emissor no sistema. Assim, os contadores de cada nodo são gravados no log a cada vez que um novo estado é identificado.

Nesta proposta, conforme apresentado anteriormente, a coordenação da transição de estados entre os nodos é feita (localmente) através do disparo de instruções de acordo com o relógio de cada nodo participante do experimento. A figura 5.4 ilustra um ambiente ideal, onde os relógios apresentam taxa de variação idêntica e estão sincronizados no evento de disparo da campanha de testes (*E0*). Portanto, pode-se assumir que as próximas ocorrências de eventos (*E1*, *E2* e *E3*) serão processadas simultaneamente entre os nodos do experimento. Entretanto, não se pode assumir um ambiente ideal, visto que relógios apresentam diferentes taxas de variação, o que requer políticas para compensar tais diferenças ou, pelo menos, identificá-las.

Para compensar as diferentes taxas de variação dos relógios bastaria, simplesmente, implementar um algoritmo de sincronização de relógios. Todavia, não é considerada uma boa prática, tendo em vista o alto custo para sincronização de relógios em tempo de execução, bem como a alta intrusividade sobre a carga de trabalho.

Focando-se na baixa intrusividade, optou-se por implementar mecanismos para detectar possíveis influências destas diferentes taxas de variação dos relógios. Em outras palavras, ao fim da etapa de testes é necessária a presença de um analisador de integridade dos resultados obtidos, que permita avaliar e identificar possíveis mensagens que violaram alguma propriedade do sistema durante o experimento.

A figura 5.5, por sua vez, ilustra um cenário real onde os nodos vão perdendo sua sincronia com os demais após o início do experimento. No instante de disparo da campanha de testes (evento *E0*), o módulo coordenador do experimento envia uma instrução

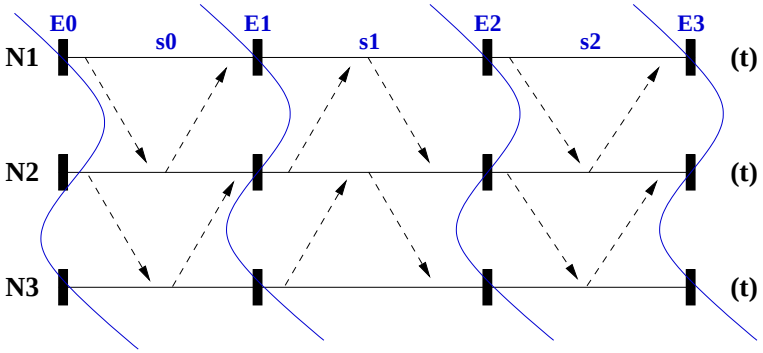

Figura 5.4: Situação ideal em um ambiente sincronizado, onde o número de mensagens enviadas e recebidas é consistente em todos os nodos.

simultânea (*broadcast*) para todos os nodos participantes do experimento para dar início à troca de mensagens.

No decorrer do experimento, o relógio de cada nodo perde sua sincronia com relação aos demais em virtude das diferentes taxas de variação. Desta forma, a transição entre eventos pode ocorrer em tempos distintos nos nodos *N1*, *N2* e *N3*, acarretando no envio de mensagens durante um estado *s0* por *N2*, mas sua entrega só é identificada por um nodo *N3* em um estado *s1*.

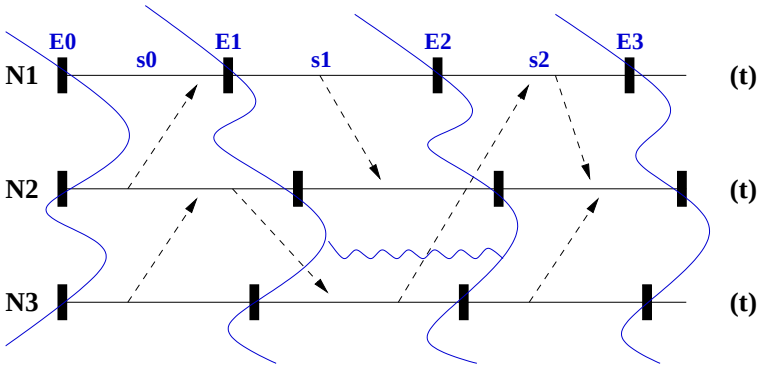

Figura 5.5: Mensagens em trânsito em um ambiente assíncrono.

Este cenário apresentado, sem um algoritmo de sincronização de relógios, dá margem para a ocorrência de violações, ou seja, há a possibilidade de nodos trocarem mensagens em estados distintos. Analisando o comportamento da comunicação entre os nodos na figura 5.5, e construindo uma tabela de *logs* seguindo o modelo de contadores locais, tem-se:

A tabela 5.1 mostra uma visão do controlador ao fim da campanha de teste; Nesta etapa, o módulo de avaliação de integridade do efetua a comparação entre contadores de cada nodo em seus respectivos intervalos. A primeira linha, por exemplo, apresenta o comportamento do nodo *N1* frente ao nodo *N2*. Nela, diz-se que o mesmo recebeu uma mensagem advinda do nodo *N2* durante o estado *s0*, mas não enviou nenhuma. Tendo isto, o módulo de análise de *logs* confronta com as informações presentes na linha 3, que contém informações do comportamento do nodo *N2* frente ao nodo *N1*. A linha 3, por sua vez, mostra que o nodo *N2* não recebera nenhuma mensagem advinda do nodo *N1*, contudo, o mesmo registra o envio de uma mensagem ao nodo *N1* durante o estado *s0*. Logo, as informações registradas por ambos nodos estão iguais e nenhuma violação foi

Tabela 5.1: Tabela de logs do controlador

	Estado s0		Estado s1		Estado s2	
Nodos	M. Rec.	M. Env.	M. Rec.	M. Env.	M. Rec.	M. Env.
N1 – N2	1	0	0	1	0	1
N1 – N3	0	0	0	0	1	0
N2 – N1	0	1	1	0	0	0
N2 – N3	1	1	0	0	1	0
N3 – N1	0	0	0	1	0	0
N3 – N2	0	1	1	0	0	1

detectada nesta comunicação, assim como nas linhas 2 e 5, referentes ao comportamento do nodo *N1* frente ao nodo *N3* e vice-versa.

Já nas linhas 4 e 6 é possível identificar uma possível violação na comunicação entre os nodos *N2* e *N3*. No estado *s0*, o nodo *N2* envia uma mensagem para o nodo *N3* que somente detecta sua chegada quando o mesmo encontra-se no estado *s1*. Logo, o módulo de análise de integridade identifica os pontos de violação e as reporta ao engenheiro de testes que, se julgar conveniente, descarta o experimento e inicia uma nova campanha.

5.2.2 Violação em Ambientes Particionados

Definida as condições para emulação adequada de particionamento de rede e detecção de possíveis violações, cabe , resta aplicá-la em ambientes onde ocorrem particionamentos de rede, foco desta proposta. Antes disso, para melhor entender o problema, a figura 5.6 ilustra um cenário ideal assíncrono (sem violações) onde ocorre um particionamento de rede.

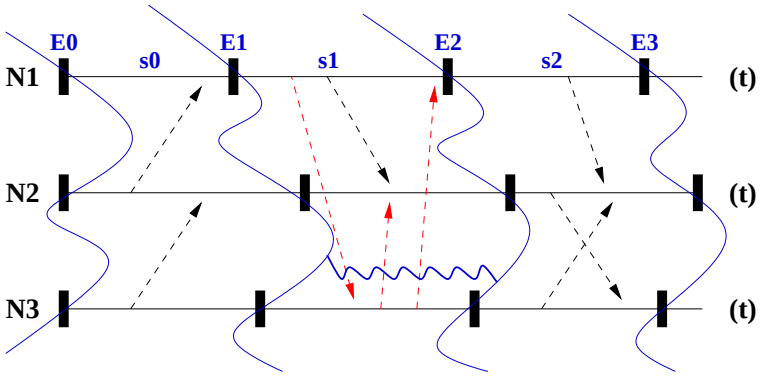

Figura 5.6: Situação ideal em um ambiente particionado assíncrono.

Dado o início do experimento (evento *E0*), os nodos *N1*, *N2* e *N3* trocam mensagens normalmente em seu estado *s0*, até que em um instante de tempo *t* é identificado e disparado um evento (*E1*) de particionamento de rede. Este particionamento, referenciado por uma linha curvada horizontal, dá origem a 2 novas partições, isolando o nodo *N3* dos nodos *N1* e *N2*. Logo, durante o estado (particionado) *s1*, todas trocas de mensagens entre as partições são descartadas. Contudo, por estarem agrupadas em uma mesma partição, os nodos *N1* e *N2* comunicam-se normalmente. Ao fim desta etapa, quando disparado um evento *E2*, os nodos assumem o estado *s2*, onde o particionamento é desfeito e a comunicação é restabelecida. Por fim, o experimento se encerra no instante de disparo do evento *E3* e é validado pelo módulo de análise de *logs*, pois não apresenta nenhuma

inconsistência nas tabelas de contadores dos nodos.

A exploração desta abordagem adotada por este trabalho, para avaliação da integridade de resultados, apresenta um ponto vulnerável, que, se não tratado, pode resultar em situações indesejadas ao fim do experimento. A figura 5.7 ilustra tal vulnerabilidade.

O experimento é iniciado e a troca de mensagens entre os nodos ocorre normalmente. Ao passo que os nodos alcançam o estado *s1*, a rede sofre um particionamento similar ao apresentado na figura 5.6, isolando o nodo *N3* dos demais nodos participantes do experimento. Nesta etapa, as mensagens de *N3* para os nodos *N1* e *N2* são descartadas, conforme o esperado. E, durante a transição do evento *E1* para *E2*, a rede ainda está particionada e duas mensagens (setas acompanhadas de um asterisco) são enviadas do nodo *N2*, uma para *N1* e outra para *N3*.

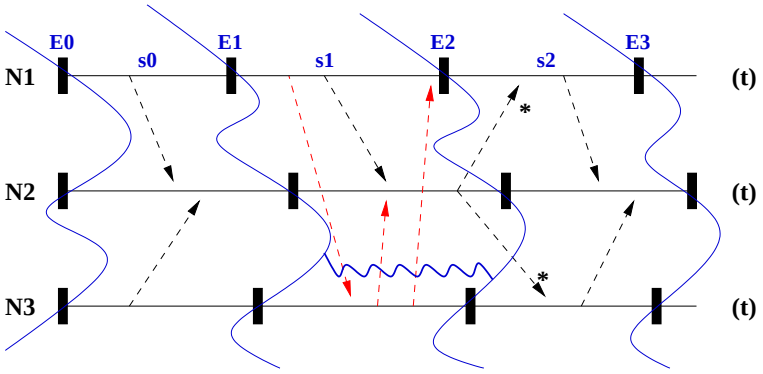

Figura 5.7: Violação em um ambiente particionado assíncrono.

A primeira delas, enviada ao nodo *N3*, só identifica seu recebimento após o disparo do evento *E2*, quando a rede não se encontra mais particionada. Esta mensagem em trânsito caracteriza uma violação e, se não identificada, pode comprometer a emulação de um particionamento de rede apropriado. Já a segunda mensagem enviada ao nodo *N1*, apesar de representar o envio de uma mensagem de um nodo para outro em estados distintos, não produz efeito colateral no ambiente de testes, visto que durante a ocorrência dos eventos de particionamento e *merge* (*E1* e *E2*) os nodos *N1* e *N2* encontram-se na mesma partição. Desta forma, o módulo de avaliação de integridade de resultados deve tratar estas condições para não caracterizar um falso positivo para violações.

Por fim, o comportamento da comunicação entre os nodos na figura 5.7 pode ser visto na tabela 5.2.

Assim como no processo de avaliação no cenário anterior (tabela 5.1), o módulo de análise de *logs* inicia confrontando as informações recebidas de cada nodo. Em outras palavras, a quantidade de mensagens enviadas e recebidas do nodo *N1* para o nodo *N2*, no estado *s0* (linha 1), é comparada com as mesmas informações coletadas do nodo *N2* para o nodo *N1*, no mesmo estado *s0* (linha 3). Logo, a quantidade de mensagens enviadas pelo nodo emissor deve ser igual ao número de mensagens recebidas pelo nodo receptor. O mesmo processo é executado para todas as demais linhas da tabela. Contudo, o módulo de avaliação de integridade deve ser capaz de identificar uma violação efetiva, quando uma mensagem é entregue ao seu destino indevidamente (causando efeitos colaterais), e uma violação não efetiva, quando uma mensagem em trânsito é entregue sem causar efeitos colaterais.

Neste caso, ao verificar a corretude das informações dos nodos *N1* e *N2* no estado *s1*, o módulo de avaliação de integridade identifica o próximo estado (*s2*), onde a mensagem

Tabela 5.2: Tabela de Logs do Controlador

	Estado s0		**Estado s1**		**Estado s2**	
Nodos	M. Rec.	M. Env.	M. Rec.	M. Env.	M. Rec.	M. Env.
N1 – N2	0	1	0	1	1*	1
N1 – N3	0	0	1	1	0	0
N2 – N1	1	0	1	1*	1	0
N2 – N3	1	0	1	1*	1	0
N3 – N1	0	0	0	1	0	0
N3 – N2	0	1	0	1	1*	1

foi entregue, e verifica se os nodos ainda encontram-se na mesma partição. Se sim, uma violação não efetiva é identificada e o experimento não é invalidado incorretamente. Caso contrário, como por exemplo o comportamento entre os nodos *N2* e *N3* no estado *s1* (linhas 4 e 6), o módulo de avaliação de integridade identifica se no estado em que a mensagem foi entregue ao receptor (estado *s2*), os mesmos permanecem com o mesmo comportamento, ou seja, particionados. Como neste caso o comportamento se altera a partir do instante *t2*, onde a comunicação é restabelecida, uma violação efetiva é detectada e o experimento é invalidado.

Como pode ser visto, a adoção e adaptação dos modelos baseadas em logs identificam corretamente os pontos cruciais onde a taxa de variação do relógio de cada nodo influencia na execução do experimento e, tendo as condições adequadas a serem avaliadas, situações indesejadas também podem ser evitadas e não invalidar testes incorretamente.

5.3 Convenções do Protótipo *PIE*

Definida as dependências de outras ferramentas para aplicabilidade do protótipo *PIE*, seu fluxo de atividades, entre outras características, esta seção trata de peculiaridades adotadas durante seu desenvolvimento a fim de facilitar a compreensão e extensão da ferramenta por terceiros.

Neste protótipo, toda comunicação entre os nodos do experimento é feita através de *bsd sockets* via protocolo *UDP*, uma vez que o ambiente desejado para execução de testes sugere *LANs*, fazendo-se desnecessária a preocupação com ordenamento de mensagens, garantia de entrega, entre outros atributos proporcionados pelo protocolo *TCP*.

Após enviar a carga de falhas para os demais nodos, o módulo *F.I.M.M.* responsabiliza-se pela leitura da carga de falhas recebida pelo respectivo nodo e prepara o ambiente. Conforme proposto, eventos de falha ocorrem em intervalos de tempo bem definidos pelo engenheiro de testes. Tais intervalos de tempo são programados no sistema a partir de temporizadores no *kernel* do sistema baseado em *jiffies*, e esta abordagem comportou-se dentro do esperado durante o processo de desenvolvimento.

Dados os intervalos de tempo em que os eventos ocorrerão, resta executar as atividades descritas na carga de falhas. Para isso, o injetor de falhas *PIE* utiliza os ganchos, ou *hooks*, na pilha de protocolos do sistema operacional através da interface de programação *Netfilter*. Este gancho consiste em uma função de retorno (*callback*) que é ativada sempre que uma mensagem cruza pela pilha de protocolos. Quando executada, esta função processa as condições impostas pela carga de falhas de modo que a mensagem seja processada de acordo com a especificação desta carga.

Logo, quando alcançado o tempo que ativa o evento de encerramento do experimento,

todos os nodos enviam suas informações coletadas durante os testes para o módulo de coordenação do experimento. A partir daí, o mesmo analisa, automaticamente, os dados dos nodos em busca de violações. Caso sejam detectadas violações, o coordenador envia uma mensagem para o engenheiro de testes que deverá descartar o experimento e efetuar uma nova execução.

O modelo adotado para avaliação de integridade de experimentos, teoricamente, apesar da baixa intrusividade sobre o sistema, poderia ser aprimorado. Isto, pois, o modelo adotado requer a atuação sobre a pilha de protocolos do sistema operacional de cada nodo, deixando cada nodo responsável pelo processamento de seus próprios contadores.

Um modelo menos intrusivo é apresentado na figura 5.8. A diferença para o modelo anterior é a inserção de um N-ésimo nodo (E), não participante do experimento, com interface de rede em modo promíscuo. Logo, seria possível manter o controle sobre as mensagens trafegantes nos nodos sob teste (A, B, C e D) sem gerar *overhead* para o controle dos contadores.

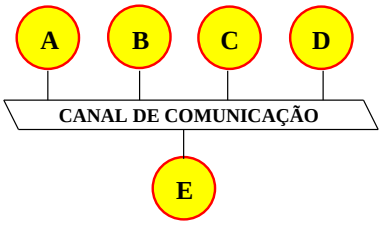

Figura 5.8: Aprimoramento do sistema de contadores.

Contudo, a implantação prática deste modelo não foi concretizada, uma vez que o *framework Netfilter* não provê mecanismos de interceptação de mensagens em modo promíscuo, exceto a partir da aplicação de *patches*. Desta forma, tal alternativa será retomada em trabalhos futuros com o objetivo de adicionar novas funcionalidades ao protótipo *PIE*, como o suporte a avaliação de integridade em tempo de execução.

5.4 Extensão do Protótipo *PIE*

Considerando a existência de outros tipos de falhas de comunicação, o projeto de desenvolvimento do protótipo *PIE* inclui, em sua biblioteca de falhas, suporte à descrição de outras falhas recorrentes na transmissão de mensagens, como descarte, atraso, duplicação de mensagens, entre outros. Entretanto, tais especificações não foram implementadas no módulo *F.I.M.M.*.

Emular outros modelos de falhas de comunicação, ao contrário do particionamento de rede, não requer coação entre nodos, ou ainda o disparo simultâneo de eventos. Diante destas condições, o disparo de falhas pode ser: *(i)* de acordo com o número de mensagens recebidas do canal de comunicação, *(ii)* de acordo com um intervalo de tempo bem definido, ou ainda *(iii)* permanente, onde a aplicação-alvo inicia e encerra sua execução em um estado específico definido pelo engenheiro de testes.

A declaração dos nodos que participarão do experimento também sofre alterações, de modo que a cláusula *@declare* é extinta e cada nodo passa a ser declarado separadamente no cabeçalho de cada comportamento especificado. A figura 5.9 apresenta as três sintaxes suportadas pela biblioteca de falhas do protótipo *PIE*.

A primeira delas refere-se ao disparo temporal de falhas, onde o injetor de falhas passa a atuar sobre a aplicação-alvo a partir do momento em que o limite de tempo,

estipulado em segundos, é alcançado. No segundo caso, o disparo de ações dá-se através da quantidade de mensagens recebidas na interface de comunicação de um determinado nodo (identificado pela letra *p* na cláusula *after*). O terceiro e último caso, por sua vez, não aguarda nenhuma condição para o disparo de falhas, ou seja, suas ações são executadas logo após o início do experimento. Desta forma, o nodo permanece no mesmo estado até que o experimento seja encerrado através de comandos enviados ao injetor de falhas pelo engenheiro de testes.

```
1: @nodo_A              @nodo_B              @nodo_C
2: START:               START:               START:
3:     after (60s) do       after (1000p) do     ações;
4:         ações;           ações;           STOP:
5:     end                  end                  manual;
6: STOP:                STOP:
7:     after (300s);        after (15000p);
```

Figura 5.9: Modelos de disparo de falhas suportado pelo interpretador de comandos.

Apresentado os diferentes métodos para o disparo de falhas suportados pela biblioteca de falhas do protótipo *PIE*, resta salientar as primitivas que descrevem os tipos de falhas comuns em ambientes de comunicação. Qual seja, a representação das mesmas dá-se através da seguinte forma:

Tabela 5.3: Outras nomenclaturas suportadas pelo interpretador de comandos

Tipo de Falha	Ação	Nomenclatura	Exemplo de Uso
DELAY	Atraso de mensagens	protocolo DELAY probabilidade (%) FOR tempo (*ms*)	*tcp* **delay** 5% **for** 2*ms*;
DROP	Descarte de mensagens	protocolo DROP probabilidade (%)	*udp* **drop** 3%;
DUPLICATE	Duplicação de mensagens	protocolo DUPLICATE probabilidade (%)	*sctp* **duplicate** 7%;

Com isto, é possível construir diversos cenários de falhas de uma maneira bastante simples e explícita ao mesclar estas primitivas com as diferentes maneiras de se disparar falhas em um experimento. Sua aplicabilidade pode ser vista na figura 5.10.

De acordo com a carga de falhas mostrada na figura, o cenário é composto de três nodos (*A*, *B* e *C*). Tanto o nodo *A*, quanto o nodo *B*, apresentam 3 eventos, sendo 2 para comportar-se de maneira específica, e 1 demarcando o fim de suas execuções. O nodo *C*, por sua vez, apresenta apenas 2 eventos, 1 para definir seu comportamento na rede, e outro também para demarcar o fim de sua execução.

Nota-se que além dos nodos apresentarem mecanismos de gatilho de falhas distintos, suas ações em cada evento também são distintas, tanto o protocolo utilizado pela aplicação-alvo, quanto o tipo de falha associado. Apesar de ser um cenário bastante improvável, ressalta-se a diversidade de cenários passíveis de reprodução a partir do interpretador de comandos do injetor de falhas *PIE*. Entretanto, conforme mencionado anteriormente, não é possível construir um cenário de falhas mesclando este grupo de falhas de comunicação com falhas de particionamento de rede.

```
1:  @nodo_A                  @nodo_B                  @nodo_C
2:  START:                   START:                   START:
3:      after (60s) do           after (1000p) do         sctp drop 3%;
4:          tcp drop 5%;             udp duplicate 3%;    STOP:
5:      end                      end                      manual;
6:      after (180s) do          after (5000p) do
7:          tcp duplicate 4%;        udp delay 3% for 2ms;
8:      end                      end
9:  STOP:                    STOP:
10:     after (300s);            after (15000p);
```

Figura 5.10: Exemplo de carga de falhas mesclando as nomenclaturas suportadas pelo interpretador de comandos.

5.5 Conclusões de Capítulo

Este capítulo apresentou características do funcionamento do ambiente de injeção de falhas *PIE*. Além disso, foram descritas características de sua implementação, em especial do módulo de análise de dados, que visa identificar possíveis pontos de violação em campanhas de teste.

Outras características para facilitar a expansão do ambiente *PIE* foram ressaltadas, a fim de dar continuidade no protótipo para cobrir um número maior de testes de comunicação.

O próximo capítulo aborda a etapa experimental deste trabalho, com o objetivo de avaliar o impacto que falhas de particionamento podem causar a uma determinada aplicação, como também o funcionamento do protótipo *PIE* e seu grau de intrusividade sobre a aplicação-alvo.

6 EXPERIMENTOS DE INJEÇÃO DE FALHAS E ANÁLISE DE RESULTADOS

Este capítulo descreve as campanhas de teste realizadas com o objetivo de validar o protótipo *PIE*, buscando demonstrar sua utilidade e viabilidade para avaliação de aplicações distribuídas.

6.1 Cenário Experimental

Para a realização da campanha de testes foram utilizados quatro máquinas Intel Core 2 duo disponíveis no laboratório do Instituto de Informática da UFRGS. O sistema operacional adotado para a realização de experimentos foi a distribuição *Zenwalk Linux* 6.2, com versão de núcleo *vanilla* 2.6.30.5. Outras configurações são descritas abaixo:

Tabela 6.1: Informações de nodos utilizados nos experimentos

Nodo	Clock (MHz)	Memória (kB)	Disco de Armazenamento
dkw	2200.013	2067088	SAMSUNG SP0802N
fusca	2200.392	2067088	MAXTOR STM325031
passat	2199.594	2067088	MAXTOR STM325031
porsche	2199.974	2067088	MAXTOR STM325031

Ressalta-se que não há necessidade de um ambiente homogêneo, conforme utilizado nos experimentos. A escolha do sistema operacional, por outro lado, deu-se de forma bastante simples. Em virtude do uso de componentes do *kernel* do Linux foi escolhida uma distribuição livre de *patches*, que alteram o núcleo do Linux, a fim de garantir que o ambiente de testes não apresente nenhuma configuração especial e, possivelmente, desconhecida.

Para encontrar uma aplicação-alvo para a etapa experimental deste trabalho foram pesquisadas diversas técnicas de comunicação de grupo presentes na literatura (BIRMAN; JOSEPH, 1985; GÄRTNER, 1999; CHOCKLER; KEIDAR; VITENBERG, 2001). Entretanto, poucas implementações funcionais foram encontradas para explorar as funcionalidades do injetor de falhas *PIE*. Desta forma, optou-se pelo uso de um *framework* já explorado em trabalhos anteriores do grupo de tolerância a falhas da UFRGS, o qual é explicado neste capítulo.

A tabela 6.2 apresenta a relação das principais aplicações utilizadas durante as campanhas de teste e suas respectivas versões.

Tabela 6.2: Informações dos aplicativos utilizados nos experimentos

Aplicativo	Versão	Propósito
bash	3.1.17(2)	scripts de automação
JGroups	2.10.0.Alpha4	Aplicação-alvo
time	1.7	controle de tempo de execução

6.2 Considerações Sobre Campanhas de Teste

Para avaliação do protótipo *PIE* foram realizadas quatro campanhas de injeção de falhas. A primeira delas consiste em demonstrar o impacto falhas de particionamento de rede na aplicação-alvo.

A segunda campanha de testes consiste nos mesmos métodos da primeira campanha, exceto pela aplicação-alvo que, desta vez, é utilizada uma aplicação-alvo tolerante a falhas de particionamento de rede. Em outras palavras, as falhas de comunicação são percebidas pela aplicação que se adapta ao ambiente e, quando a comunicação é restabelecida, esta se recupera e mantém um estado consistente entre os nodos.

A terceira campanha visa avaliar a intrusividade do protótipo *PIE* através de uma aplicação-alvo com alta taxa de troca de mensagens, e é dividida em três etapas: (*E1*) experimentos com falha, cenário com até 4 partições, (*E2*) experimentos com falha, cenário com uma única partição (melhor caso), e (*E3*) sem falhas, cenário sem a atuação de *PIE* sobre a aplicação-alvo.

Todas campanhas de teste descritas se deram em partida quente, consistindo em 20 execuções cada. Todavia, não foi possível ativar o módulo de verificação de inconsistências em todos os experimentos, visto que foram realizados sobre a rede aberta do Instituto de Informática da UFRGS em virtude da indisponibilidade de um equipamento adequado para construir uma rede fechada. Isto, pois, o mecanismo utilizado para controle de violação leva em consideração apenas os nodos do experimento para alocação da memória necessária que armazenará as tabelas de *logs*. Logo, com a rede aberta, todas as mensagens do barramento da rede acabaram influenciando no funcionamento adequado deste atributo, que foi retirado da avaliação inicial do protótipo e avaliado isoladamente em uma quarta campanha.

6.3 Sistema-Alvo

Para avaliação de todas as características ressaltadas do ambiente de injeção de falhas *PIE*, foi utilizado o *framework* de comunicação de grupo confiável escrito em Java, Jgroups (RED HAT, 2009). Seu funcionamento dá-se a partir da inserção de uma camada de agrupamento sobre um protocolo de transporte que, internamente, manterá uma lista dos nodos participantes da comunicação. Esta camada, dentre outras coisas, responsabiliza-se por:

- permitir que a aplicação enxergue todos os demais nodos em escuta;
- permitir a entrega ordenada e confiável de mensagens;
- permitir transmissão atômica de mensagens, ou seja, todos os nodos recebem a mensagem ou nenhum;

Desta forma, JGroups pode ser utilizado para criar grupos de nodos cujos membros troquem mensagens entre si. Apesar de prover suporte a desenvolvedores de software

para a criação de aplicações baseadas em comunicações de grupo confiáveis, este trabalho adota 3 das inúmeras aplicações de demonstração disponibilizadas pelo próprio *framework*.

A escolha de tais aplicações-alvo levou em consideração três aspectos: (*i*) a importância de tratar falhas de particionamento de rede (*Whiteboard*), (*ii*) a demonstração do funcionamento do protótipo (*Topology*), e (*iii*) a avaliação da intrusividade do protótipo sobre a aplicação-alvo (*Draw*).

6.4 Primeira Campanha: Aplicação que não Trata Particionamento

Para dar início aos experimentos com injeção de falhas será mostrado o comportamento sob falhas de particionamento de rede em uma aplicação que não leva em consideração o tratamento deste tipo de falhas.

A aplicação *Whiteboard* é o primeiro dos aplicativos de demonstração do *framework* JGroups. Com ele, o objetivo deste trabalho é alertar sobre os impactos que falhas de comunicação de particionamento de rede podem causar.

Este aplicativo consiste em pegar as informações do nodo local e apresentá-las na tela. À medida que os demais nodos executam suas instâncias, a janela de informações do nodo local é atualizada e passa a mostrar também as informações dos demais nodos conectados.

Considerando a inexistência de mecanismos de tolerância a falhas de comunicação nesta aplicação-alvo, quando o protótipo *PIE* atua sobre o ambiente, a aplicação *Whiteboard* atualiza as informações apresentadas na tela do nodo local com a sua visão de rede particionada.

Quando ocorre o *merge*, a aplicação consegue atualizar sua visão de rede com a volta dos nodos que estavam particionados. A aplicação não contém técnicas para verificar consistência entre estados e, após o *merge*, as informações apresentadas na tela de cada nodo são distintas e desatualizadas, embora com a mesma visão rede, caracterizando uma inconsistência entre estados.

```
#                    fusca          passat          dkw          porsche
1: @declare {143.54.10.141, 143.54.10.126, 143.54.10.101, 143.54.10.42}
2:  START:
3:     after (5s) do
4:        partition {143.54.10.141, 143.54.10.101} {143.54.10.126, 143.54.10.42};
5:     end
6:     after (20s) do
7:        partition {143.54.10.141, 143.54.10.126, 143.54.10.101, 143.54.10.42};
8:     end
9:     after (30s) do
10:       partition {143.54.10.141} {143.54.10.101} {143.54.10.126} {143.54.10.42};
11:    end
12:    after (45s) do
13:       partition {143.54.10.141, 143.54.10.126, 143.54.10.101, 143.54.10.42};
14:    end
15:  STOP:
16:    after (60s);
```

Figura 6.1: Carga de falhas utilizada com a aplicação *Whiteboard*.

A figura 6.1 descreve a carga de falhas utilizada neste experimento, que divide-se em dois eventos de particionamento de rede e 2 eventos de *merge*. O primeiro evento, aos

5 segundos após o início da execução, divide a rede em duas partições com dois nodos cada. A seguir, aos 20 segundos, é identificado um evento de *merge*, situação na qual os nodos estão agrupados em uma única partição. Aos 30 segundos um novo evento de particionamento de rede afeta o ambiente que, por sua vez, é dividido em 4 partições isoladas com um nodo cada. O último evento ocorre aos 45 segundos, onde a rede é normalizada e os nodos voltam a agrupar-se em uma única partição. Por fim, aos 60 segundos, o experimento se encerra.

Descrito o cenário de falhas para este experimento, inicia-se o processo de execução da aplicação que será submetida aos eventos acima. A figura abaixo (figura 6.2) apresenta a fase inicial do experimento, ou seja, onde cada nodo executa sua instância da aplicação *Whiteboard*. Nota-se que o indicador do número de membros na rede, e em comunicação é destacado ao lado do botão *exit*, e os demais retângulos mostram as informações de cada um dos nodos, conforme explicado neste capítulo.

Ressalta-se que a estrutura do modelo apresentado na figura será seguido até o fim deste capítulo para fins de facilitar a visualização do ambiente experimental como um todo. Desta forma, a primeira instância, a partir da esquerda, diz respeito ao nodo *dkw*, seguido pela instância dos nodos *fusca*, *passat* e *porsche*.

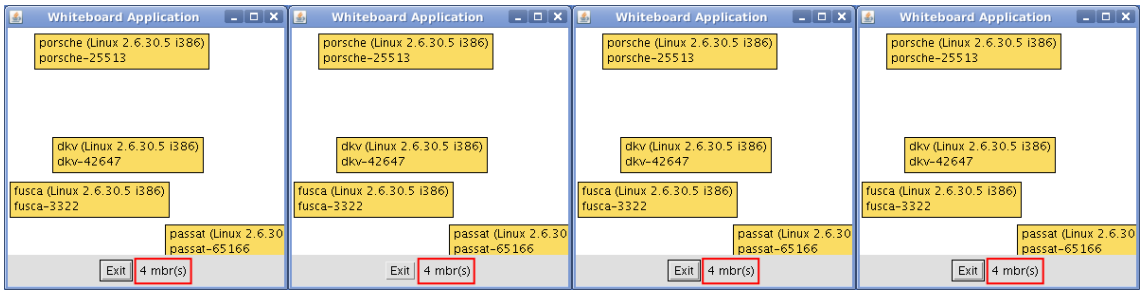

Figura 6.2: Aplicação *Whiteboard* após instanciação de cada um dos nodos

A figura 6.3, por sua vez, reflete o ambiente após o primeiro evento de particionamento, aos 5 segundos. Neste, os nodos *dkw* e *passat* ficam isolados em uma partição, e os nodos *fusca* e *porsche* isolados em outra.

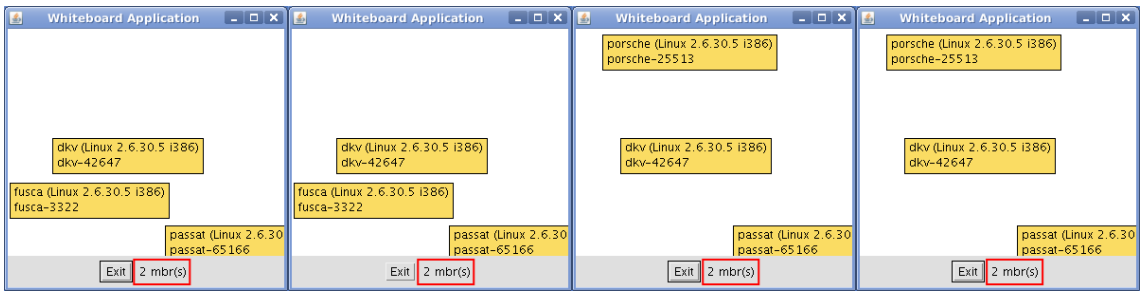

Figura 6.3: Aplicação *Whiteboard* após particionamento aos 5 segundos

Visto que os intervalos entre falhas de particionamento e a restauração da comunicação ocorrem em intervalos muito curtos, é possível notar que a aplicação *Whiteboard* não atualizou completamente a janela de informações de todos os nodos. Entretanto, nota-se uma diminuição no número de nodos integrantes do grupo de comunicação daquela instância. Comportamento este, em decorrência do particionamento previsto pela carga de falhas.

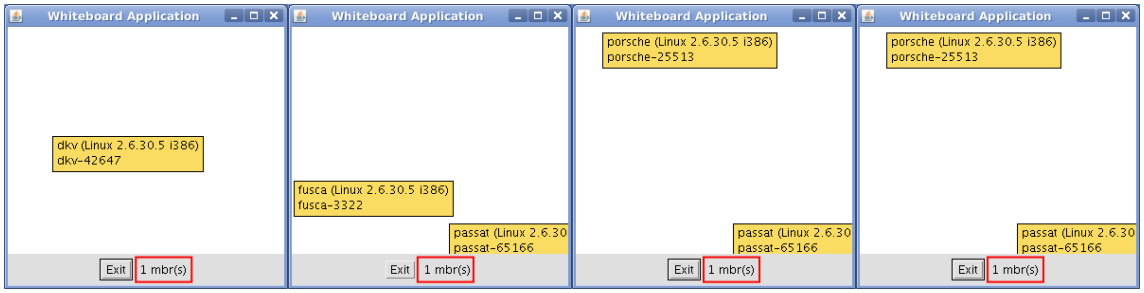

Figura 6.4: Aplicação *Whiteboard* após particionamento aos 30 segundos

Já na figura 6.4, o comportamento anterior se repete, desta vez com o ambiente sendo particionado em quatro grupos isolados. Todas as instâncias detectaram sua nova visão de grupo. Entretanto, somente o nodo *dkw* atualizou sua janela de informações, enquanto as demais instâncias permaneceram com a janela de informações referente ao evento de particionamento anterior, com 2 nodos em cada grupo.

Por fim, a figura 6.5 apresenta o ambiente após o fim do experimento, cujo estado atual é de *merge*. E, conforme o esperado, todas as instâncias detectaram a presença dos demais nodos do experimento de acordo com o número de nodos presente na partição. Apesar de identificar a restauração da comunicação entre os nodos, a aplicação *Whiteboard* não tratou a inconsistência entre os estados de cada instância.

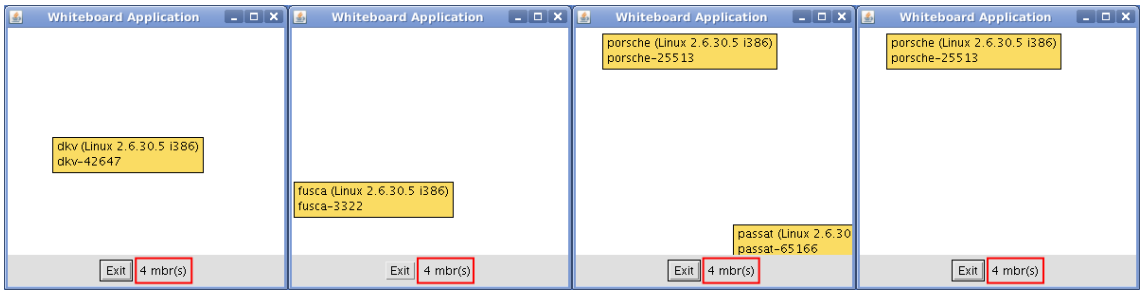

Figura 6.5: Aplicação *Whiteboard* após o fim do experimento

Além disso, considerando um ambiente em produção, a entrada de novos nodos no grupo de comunicação seria detectada por todos os componentes atuais, e suas janelas de informação também seriam atualizadas. Entretanto, novos particionamentos entre esses nodos comprometeriam completamente a integridade dos dados, tendo em vista os estados distintos apresentados após estes eventos de falha.

6.4.1 Conclusões Parciais

A primeira campanha consistiu, basicamente, em salientar a importância da implementação e testes dos mecanismos de tolerância a falhas presentes no ciclo de desenvolvimento de um projeto de *software*. Caso contrário, as consequências da propagação de falhas não toleradas (ou previstas) na aplicação podem ser catastróficas, visto que os resultados apresentados pelo sistema podem estar comprometidos e transparentes ao usuário final.

Apesar de não apresentar mecanismos de tolerância a falhas de particionamento de rede, a aplicação-alvo *Whiteboard* permite validar parcialmente a atuação do protótipo PIE. Isto, pois a aplicação-alvo não apresenta cobertura de falhas, identificando apenas as quedas de *link*, e não estando apto a tratar as inconsistências geradas após o *merge*.

6.5 Segunda Campanha: Aplicação que Trata Falhas de Particionamento

Esta seção apresenta a segunda campanha de testes, onde avalia-se a corretude das falhas injetadas pelo protótipo *PIE* através da aplicação-alvo *Topology*, sensível a falhas de particionamento de rede, ou seja, capaz de recuperar-se de falhas e tratar inconsistências entre os novos estados.

Topology é uma aplicação-alvo disponibilizado pelo *framework* JGroups que, diferentemente do primeiro, apresenta técnicas de tolerância a falhas para cobertura de falhas de comunicação.

Após cada nodo executar sua instância da aplicação, será apresentada na tela a visão de cada nodo, representada por um retângulo contendo suas respectivas informações. Após definida a visão de grupo com os nodos desejados, a aplicação *Topology* define um nodo coordenador para gerenciar a lista de membros do grupo. Este coordenador pode ser identificado nos experimentos por apresentar sua janela de execução com fundo escuro.

A partir de então, a cada nova entrada/saída de membros do grupo todos os membros do grupo atualizarão sua janela com as mesmas informações presentes na janela do nodo coordenador, garantindo um estado consistente mesmo na presença de falhas de comunicação.

Cabe salientar que, quando o ambiente sofre particionamento de rede, cada partição existente conterá um novo nodo coordenador.

A carga de falhas utilizada nesta campanha está descrita na figura 6.6, e é similar à carga de falhas utilizada na campanha de testes anterior, com os mesmos 2 eventos de falha de particionamento de rede e 2 eventos de *merge*, diferenciando-se apenas pelos instantes de tempo em que as ações devem ocorrer, identificadas na cláusula *after*.

```
#                    fusca          passat          dkw          porsche
1:  @declare {143.54.10.141, 143.54.10.126, 143.54.10.101, 143.54.10.42}
2:  START:
3:     after (10s) do
4:        partition {143.54.10.141, 143.54.10.101} {143.54.10.126, 143.54.10.42};
5:     end
6:     after (20s) do
7:        partition {143.54.10.141, 143.54.10.126, 143.54.10.101, 143.54.10.42};
8:     end
9:     after (30s) do
10:       partition {143.54.10.141} {143.54.10.101} {143.54.10.126} {143.54.10.42};
11:    end
12:    after (40s) do
13:       partition {143.54.10.141, 143.54.10.126, 143.54.10.101, 143.54.10.42};
14:    end
15: STOP:
16:    after (60s);
```

Figura 6.6: Carga de falhas utilizada com a aplicação *Topology*.

A figura 6.7 mostra o ambiente logo após a instanciação da aplicação *Topology* pelos nodos participantes do experimento. Neste, por ser o nodo mais antigo a ter entrado no grupo, o nodo *passat* torna-se o coordenador da aplicação, identificado pelo fundo escuro, e repassa sua visão de grupo para os demais nodos do ambiente.

Figura 6.7: Aplicação *Topology* após instanciação de cada um dos nodos

Após o particionamento de rede, aos 10 segundos, são criadas duas partições isoladas, uma contendo os nodos *dkw* e *fusca*, e outra com os nodos *passat* e *porsche*. Quando este evento ocorre, é identificado pela aplicação o isolamento do coordenador em uma partição. Logo, os nodos da outra partição elegem um novo coordenador, ou seja, o nodo mais antigo do grupo, e pode ser visto na figura 6.8.

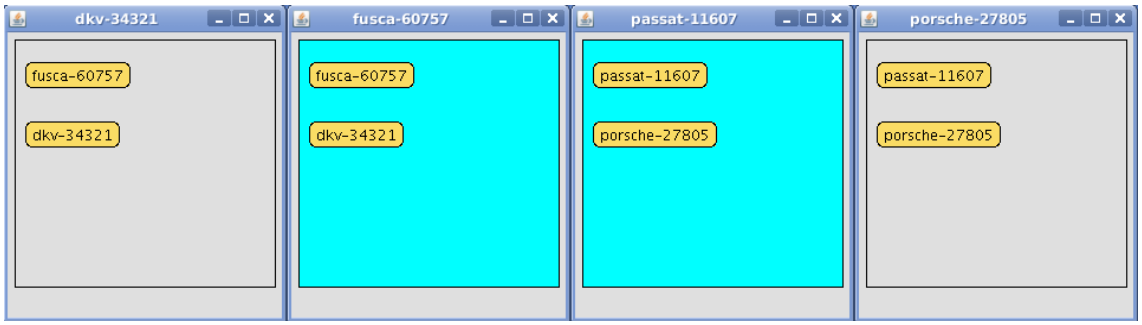

Figura 6.8: Aplicação *Topology* após particionamento aos 10 segundos

Aos 30 segundos ocorre um evento de *merge*. Entretanto, os 10 segundos restantes para o próximo evento não foram suficientes para que a aplicação se recuperasse do particionamento, visto que a desconexão de um nodo do grupo, no *framework JGroups*, é tratada de maneira muito menos complexa que a entrada de um novo integrante no grupo.

Desta forma, o evento de particionamento aos 30 segundos divide a rede em 4 partições isoladas, cada uma com um único nodo. Assim como no evento de falha anterior, a aplicação identifica a existência de mais de um grupo de comunicação sem um nodo coordenador. Logo, tendo em vista o cenário atual, cada nodo vira o coordenador de sua partição até que o sistema se recupere da falha (figura 6.9).

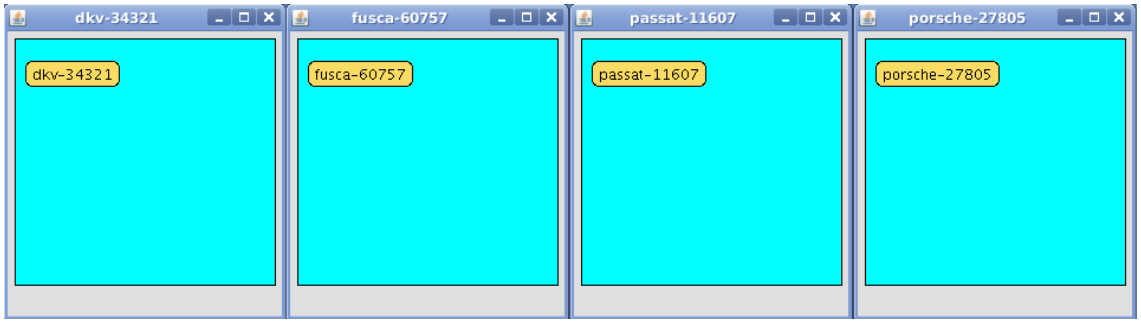

Figura 6.9: Aplicação *Topology* após particionamento aos 30 segundos

Por fim, o ambiente sofre um último evento aos 45 segundos onde o particionamento é desfeito e todos os nodos voltam a comunicar-se. A partir daí o cenário permanece nesse estado até o fim do experimento, aos 60 segundos. A figura 6.10 mostra o comportamento da aplicação *Topology* após o experimento, recuperando-se dos particionamentos e voltando a ter um único grupo de nodos em que todos estão acessíveis a todos diretamente. E, conforme a imagem, a aplicação volta a ter um único coordenador.

Figura 6.10: Aplicação *Topology* após o fim do experimento

6.5.1 Conclusões Parciais

Esta segunda campanha de testes teve por objetivo mostrar a corretude das atividades exercidas pelo protótipo *PIE*. Logo, utilizou-se a aplicação *Topology* para permitir a visualização de um cenário reproduzido a partir das descrições contidas na carga de falhas do engenheiro de testes.

Conforme o esperado, a aplicação *Topology* reproduziu fielmente o comportamento especificado na carga de falhas. Desta forma, foi possível acompanhar passo-a-passo o processo de quebra do *link* de comunicação entre os nodos, bem como o processo de tratamento de falhas da aplicação, tanto nos eventos de particionamento, quanto nos eventos de *merge*.

Este comportamento torna o ambiente de injeção de falhas *PIE* útil para quaisquer engenheiros de testes com o intento de avaliar as técnicas de dependabilidade de uma determinada aplicação. Isto, pois, o uso deste ambiente anula a necessidade de métodos pouco ortodoxos para testes de comunicação, que vão desde a criação de regras no *firewall* até a desconexão manual de cabos de comunicação, dando lugar a possibilidade de criação de um ambiente controlado e com maior variedade de cenários de falha.

6.6 Terceira Campanha: Viabilidade do Injetor de Falhas PIE

Conforme falado anteriormente, esta última campanha é dividia em três experimentos. As duas primeiras com a atuação do protótipo *PIE* sobre o ambiente, e a última sem sua presença. Com isso, será possível analisar o comportamento da aplicação-alvo, bem como o grau de intrusividade do protótipo, ou seja, determinar se houve degradação de desempenho da aplicação com a atuação do ambiente de injeção de falhas *PIE*.

Draw é outra aplicação-alvo disponível a partir do *framework Jgroups*. Trata-se de um quadro compartilhado entre nodos de um mesmo grupo e, assim como as aplicações anteriores, cada nodo que executa sua instância entra no mesmo grupo que as demais.

Após executar uma instância, automaticamente é escolhida uma cor aleatória que, ao mover o mouse sobre o quadro, um rastro sobre o caminho percorrido é registrado de

acordo com a cor eleita por esta instância. Além disso, cada nova movimentação do cursor é replicada pelas demais instâncias através de mensagens *broadcast* para os membros do grupo.

Em conjunto com *Draw*, que depende de interação humana, foi desenvolvido para este trabalho um algoritmo para manipulação dos movimentos do cursor. Desta forma será possível medir o nível de intrusividade do ambiente de injeção de falhas *PIE* sobre a aplicação-alvo.

A figura 6.11 apresenta a descrição do cenário de falhas. Assim como nos experimentos anteriores, são 4 eventos divididos em um intervalo de 60 segundos, sendo 2 responsáveis por particionar a rede (aos 20 e aos 40 segundos), e os outros 2, aos 22 e 43 segundos, responsáveis pelo restabelecimento da conexão entre os nodos.

```
#                    fusca          passat        dkw          porsche
1: @declare {143.54.10.38, 143.54.10.36, 143.54.10.59, 143.54.10.78}
2:  START:
3:     after (20s) do
4:        partition {143.54.10.38, 143.54.10.36} {143.54.10.59, 143.54.10.78};
5:     end
6:     after (22s) do
7:        partition {143.54.10.38, 143.54.10.36, 143.54.10.59, 143.54.10.78};
8:     end
9:     after (40s) do
10:       partition {143.54.10.38} {143.54.10.59} {143.54.10.36} {143.54.10.78};
22:    end
11:    after (43s) do
12:       partition {143.54.10.38, 143.54.10.36, 143.54.10.59, 143.54.10.78};
13:    end
14:  STOP:
15:    after (60s);
```

Figura 6.11: Carga de falhas utilizada com a aplicação *Draw*.

A figura 6.12 apresenta a aplicação *Draw* logo após sua instanciação e início da movimentação do cursor. Para fins de facilitar a visualização do cenário, optou-se pela reprodução dos mesmos movimentos por todos os nodos, diferenciando-se apenas pelas cores. Logo, as cores das próximas imagens nesta seção seguem a seguinte ordem: amarelo (*passat*), magenta (*fusca*), verde (*dkw*), e vermelho (*porsche*).

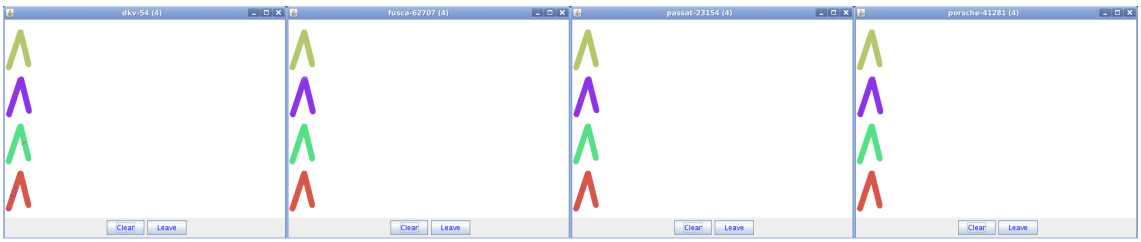

Figura 6.12: Aplicação *Draw* após instanciação de cada um dos nodos

.

A seguir, a figura 6.13 reflete o estado do ambiente no instante após a ocorrência da primeira falha de particionamento de rede, aos 20 segundos. Neste, os nodos *dkw* e *fusca*

ficam isolados em uma partição e passam a não receber mensagens dos outros nodos, de maneira que os movimentos dos nodos *passat* e *porsche* não são reproduzidos. O mesmo ocorre na outra partição, que não recebe as mensagens advindas dos nodos *dkw* e *fusca* e o ambiente entra em um estado inconsistente.

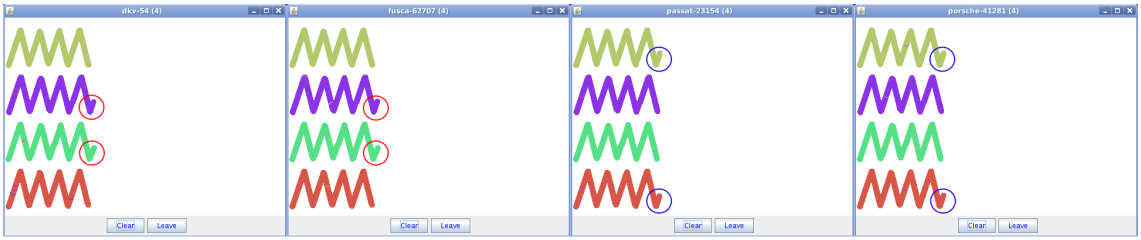

Figura 6.13: Aplicação *Draw* após particionamento aos 20 segundos

Logo após, aos 22 segundos, o ambiente sofre o *merge*, de modo que os nodos retomam sua comunicação e a aplicação inicia o tratamento da inconsistência gerada pela falha de particionamento de rede. É o que mostra a figura 6.14, o instante de 20 segundos recuperado completamente, visto que nenhum intervalo pode ser identificado neste ponto. A figura também reflete o segundo evento de particionamento, aos 40 segundos, quando são formadas 4 partições isoladas e estão destacadas na figura.

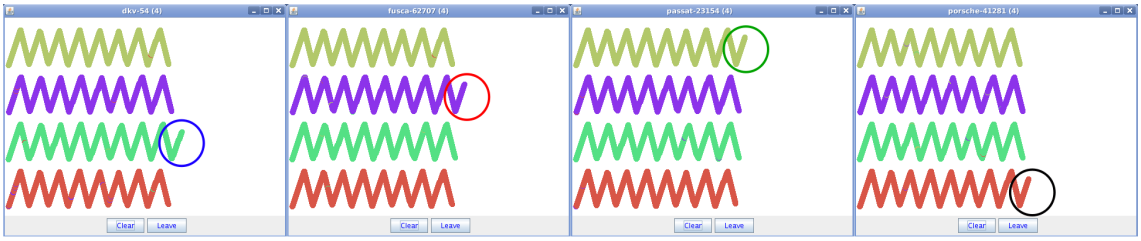

Figura 6.14: Aplicação *Draw* após particionamento aos 40 segundos

Passados 60 segundos o experimento se encerra e, novamente, o instante de falha anterior é recuperado com sucesso e a aplicação consegue manter um estado consistente entre todos os nodos, conforme apresenta a figura 6.15.

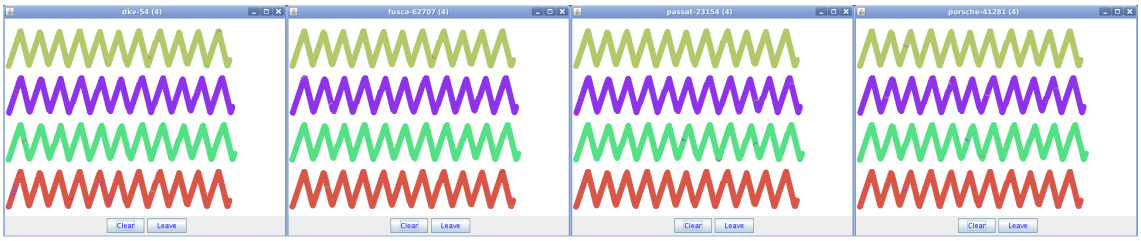

Figura 6.15: Aplicação *Draw* após o fim do experimento

O segundo experimento (*E2*), por sua vez, diferencia-se da anterior apenas pelo número de partições nos eventos de falha. Em outras palavras, buscou-se avaliar o grau de intrusividade do protótipo *PIE* definindo uma única partição (somente eventos de *merge*), do início ao fim do experimento, contendo todos os nodos.

Já a etapa final desta campanha consiste na verificação do tempo de execução da aplicação-alvo (*Draw*) sem a atuação de *PIE* no ambiente, de modo que se possa comparar com os tempos de execução das etapas anteriores.

6.6.1 Conclusões Parciais

Esta campanha procurou avaliar o grau de intrusividade de *PIE* sobre a aplicação *Draw*, ou seja, o impacto na degradação de desempenho da aplicação-alvo. Para a obtenção de tais medidas foi elaborado um *shell script* que prepara o ambiente automaticamente. Este preparo consiste em executar uma instância da aplicação em cada um dos nodos, ativar o módulo *F.I.M.M.* nos mesmos e, por fim, ativar o algoritmo responsável pela produção da carga de trabalho (movimentos do cursor).

Após ativado, o módulo *F.I.M.M.* atua sobre o ambiente por 60 segundos, conforme sua carga de falhas. O algoritmo responsável pela carga de trabalho, por sua vez, foi desenvolvido para encerrar sua execução dentro do espaço de tempo em que o módulo *F.I.M.M.* atua. Logo, avalia-se seu tempo de execução, o número de movimentos do cursor completos até o fim do experimento, bem como a quantidade aproximada de mensagens trafegadas na rede do Instituto de Informática UFRGS, e de mensagens da aplicação-alvo.

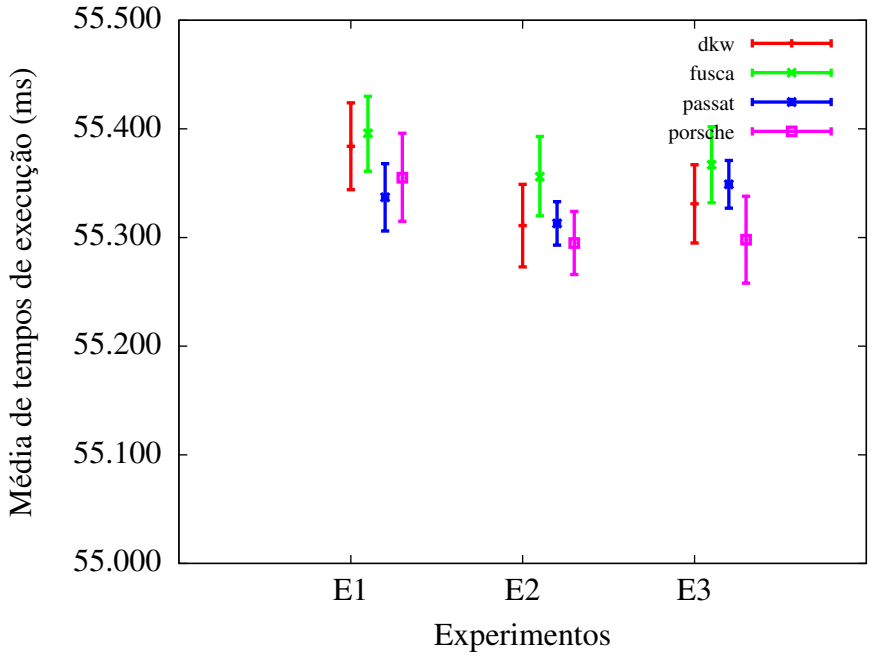

Figura 6.16: Média dos tempos de execução de cada nodo durante as etapas.

O gráfico 6.16 mostra a média dos tempos de execução de cada nodo durante as três etapas de avaliação de intrusividade. Conforme o previsto os experimentos da primeira etapa (*E1*), que consistiam em um cenário de até 4 partições, marcaram os maiores tempos dentre as etapas.

As outras duas etapas (*E2* e *E3*), mostram resultados inesperados, visto que as médias dos tempos de execução da etapa *E3*, onde o injetor de falhas não atua, foi superior em relação à média apresentada na etapa *E2*.

Tendo em vista a escala apresentada pelo gráfico, é relativamente difícil explicar tal comportamento, visto que com os instrumentos utilizados para mensurar o grau de intrusividade no desempenho da aplicação-alvo não permitiram obter resultados com maior precisão. Entretanto, as diferenças entre as médias dos tempos de execução de cada etapa foram inferiores a 100 *ms*, o que caracteriza uma perturbação praticamente nula do injetor de falhas *PIE* sobre a aplicação-alvo.

Ainda na média dos tempos de execução, salienta-se que o nodo *fusca* manteve-se

como o nodo mais lento em todas as campanhas. Apesar das diferenças de tempo para os demais nodos ser insignificante, o nodo *fusca* foi o coordenador de todas as campanhas realizadas. Ou seja, todas as descrições de cenário de falhas foram enviadas a partir deste que, ao final de cada experimento, recebia o relatório de execução de cada um dos demais nodos, acarretando em uma maior demanda por processamento.

O gráfico 6.17, por sua vez, mostra a atividade da rede do Instituto de Informática da UFRGS, que se manteve constante durante os experimentos, e da aplicação *Draw*.

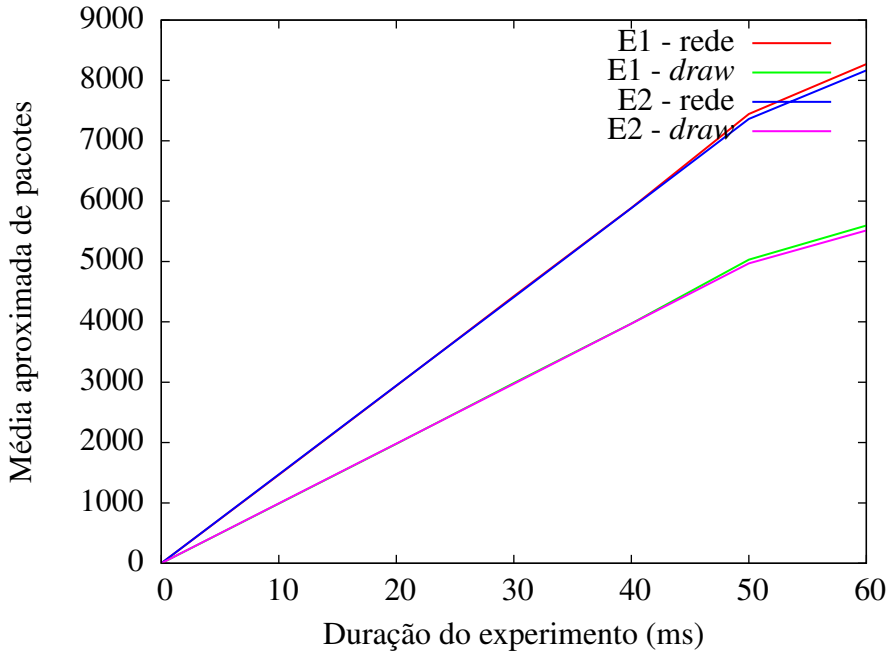

Figura 6.17: Média aproximada de pacotes durante o experimento.

O gráfico apresenta apenas as medidas referentes aos primeiros experimentos (*E1* e *E2*), pois os dados são coletados através de mecanismos provenientes do protótipo *PIE* e a etapa *E3* não considera a atuação do mesmo sobre o ambiente. Outro fator a ser destacado é o valor aproximado de pacotes, de modo que foram definidos dois contadores. O primeiro é um contador incondicional de pacotes, que marca todo e qualquer pacote na rede do Instituto de Informática da UFRGS. O segundo é um contador com filtro pelo protocolo de comunicação *UDP*, visto que, tanto o injetor de falhas, quanto a aplicação *Draw* fazem uso do mesmo para o envio e recebimento de mensagens. Logo, a partir deste mecanismo, o nível de precisão é relativamente baixo, embora os dados tenham sido coletados em horários de pouco movimento.

No intervalo entre 50 e 60 segundos, é possível perceber uma queda do fluxo de mensagens na rede. Esta queda demarca o instante em que o experimento se encerra pois, conforme citado, o algoritmo responsável pela produção da carga de trabalho foi programado para encerrar seu processamento antes do fim da atuação do módulo *F.I.M.M.*.

6.7 Quarta Campanha: Avaliação do Mecanismo de Detecção de Violações

Esta seção descreve passo-a-passo o processo de execução de experimentos levando em consideração a violação de integridade. Em outras palavras, diferentemente das cam-

panhas de teste anteriores, visa avaliar outro componente do protótipo *PIE*, o módulo de verificação de integridade do experimento. Por fim, é ele quem vai reportar ao engenheiro de testes se o resultado obtido do experimento é válido ou não.

Para a realização de experimentos com verificação de integridade foi utilizado um *switch* 10 Mbit/s. Desta forma foi possível criar uma rede isolada entre quatro máquinas. Uma quinta máquina foi designada neste cenário para fins de amostra, visto que anteriormente foi ressaltado que o controlador do experimento não necessariamente deve participar do experimento. Logo, um quinto nodo (N5) foi adicionado para coordenar o experimento.

A carga de trabalho entre os nodos, por sua vez, foi gerada a partir da criação de um *software* de troca de mensagens, baseado em comunicação cliente/servidor. Este, com o intuito de manter controle sobre tal carga, atuará dentro de um intervalo de tempo de 60 segundos. Desta forma, será possível avaliar a solução adotada com diferentes níveis de fluxo de pacotes na rede.

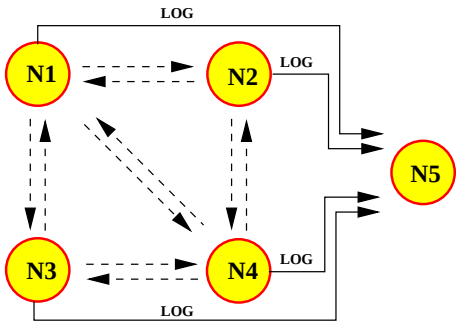

Figura 6.18: Cenário utilizado para avaliação de integridade.

A figura 6.18 ilustra o esquema de funcionamento do *software* desenvolvido. Todos os nodos estão acessíveis entre si, e a carga de trabalho estipulada é dividida em 60 segundos. Cada mensagem é enviada aleatoriamente para um dos demais nodos, ou seja, ao longo do período de 60 segundos, os nodos não necessariamente receberão a mesma quantidade de mensagens, identificadas por setas pontilhadas. Ao fim do experimento, cada nodo participante do experimento envia sua tabela de *log* para o nodo controlador (N5) que, por sua vez, dará início ao processo de verificação da integridade do experimento.

Definido o cenário e seus complementos, foram executados testes com quatro níveis de carga de trabalho: (i) 60 mensagens por nodo, durante 60 segundos, com o objetivo de emular um cenário com baixa carga de trabalho; (ii) 2000 mensagens por nodo, durante 60 segundos, para emular um cenário mais fiel de modo que haja mensagens específicas de algoritmos para o tratamento de inconsistências geradas durante o particionamento de rede; (iii) 5000 mensagens por nodo, durante 60 segundos, para emular um ambiente sobrecarregado; e, por fim, (iv) 10000 mensagens por nodo, durante 60 segundos, para emular um cenário saturado.

Para os diferentes níveis de carga de trabalho, foi utilizada a mesma descrição de cenário de teste descrita na figura 6.19. Aos 10 segundos está previsto um evento de particionamento da rede em duas ilhas distintas que, rapidamente, aos 20 segundos de execução é desfeito e restabelece a comunicação entre todos os nodos. Dos 30 aos 49 segundos ocorre o mesmo comportamento do ciclo anterior, um particionamento de rede seguido de um evento de *merge*. Por fim, aos 50 segundos a rede volta a ser particionada e permanece assim até o fim do experimento.

```
#              fusca      passat     dkw      porsche
1: @declare {10.0.0.1, 10.0.0.2, 10.0.0.3, 10.0.0.4}
2:  START:
3:     after (10s) do
4:        partition {10.0.0.1, 10.0.0.2} {10.0.0.3, 10.0.0.4};
5:     end
6:     after (20s) do
7:        partition {10.0.0.1, 10.0.0.2, 10.0.0.3, 10.0.0.4};
8:     end
9:     after (30s) do
10:       partition {10.0.0.1, 10.0.0.3} {10.0.0.2, 10.0.0.4};
11:    end
12:    after (40s) do
13:       partition {10.0.0.1, 10.0.0.2, 10.0.0.3, 10.0.0.4};
14:    end
15:    after (50s) do
16:       partition {10.0.0.2, 10.0.0.3} {10.0.0.1, 10.0.0.4};
17:    end
18:  STOP:
19:    after (60s);
```

Figura 6.19: Carga de falhas utilizada com a aplicação *Cliente/Servidor*.

A tabela a seguir, 6.3, apresenta uma relação entre a carga de trabalho utilizada por cada nodo ao longo de 60 segundos, e o número total de execuções necessárias para obter-se uma amostra de 20 execuções válidas de um determinado cenário.

Tabela 6.3: Tabela de relação carga de trabalho/execuções válidas

N^o msgs/nodo	**Execuções Necessárias**
60	20
2000	20
5000	23
10000	56

As tabelas 6.4 e 6.5 apresentam a visão do coordenador de experimentos após a coleta de informações de cada um dos nodos envolvidos no teste. Analisando o conteúdo de cada uma das linhas da tabela, também é possível comprovar a variação do fluxo de mensagens de um experimento para outro, mesmo tendo sido designada a mesma carga para ambos (10 mil mensagens por minuto).

As tabelas apresentam 6 estados distintos cada. Para fins de esclarecimento, ressalta-se que o primeiro estado, *s0*, refere-se ao intervalo entre o disparo do experimento por parte do coordenador de experimentos e o primeiro evento definido na primeira cláusula *after* (*s1*). A partir desta, seguem os estados definidos pelas demais cláusulas definidas aos 20 (*s2*), 30 (*s3*), 40 (*s4*) e 50 segundos (*s5*). O evento definido aos 60 segundos não entra para a tabela de *logs* do nodo por tratar-se apenas de um demarcador para o fim do experimento, ou seja, não há informações a serem coletadas após o alcance do estado definido (*STOP*).

A figura 6.20 apresenta as informações mostradas pelo módulo de análise de integriade ao engenheiro de testes após o fim do experimento. Nesta, o componente responsável pela verificação utiliza os dados coletados e, a partir daí, inicia o processo de avaliação da integridade do experimento. A primeira linha da tabela 6.5 (linha *0*, *67108874* –

Tabela 6.4: Exemplo de uma tabela íntegra de logs do controlador

	Estado s0		Estado s1		Estado s2		Estado s3		Estado s4		Estado s5	
Nodos	**Rec.**	**Env.**	**Rec.**	**Env.**	**Rec.**	**Env.**	**Rec.**	**Env.**	**Rec.**	**Env.**	**Rec.**	**Env.**
67108874 – 50331658	698	1196	698	864	698	1030	1428	1262	864	1362	1096	266
67108874 – 33554442	1030	698	764	432	532	532	532	532	1030	200	598	598
67108874 – 16777226	864	366	1262	764	1030	532	930	266	1196	698	1030	1030
33554442 – 50331658	698	864	764	764	532	1196	100	764	532	698	532	366
33554442 – 67108874	698	1030	432	764	532	532	532	532	200	1030	598	598
33554442 – 16777226	366	366	532	532	532	532	432	598	200	532	100	930
50331658 – 33554442	864	698	764	764	1196	532	764	100	698	532	366	532
50331658 – 67108874	1196	698	864	698	1030	698	1262	1428	1362	864	266	1096
50331658 – 16777226	1030	864	100	598	698	1030	698	532	864	864	764	266
16777226 – 50331658	864	1030	598	100	1030	698	532	698	864	864	266	764
16777226 – 33554442	366	366	532	532	532	532	598	432	532	200	930	100
16777226 – 67108874	366	864	764	1262	532	1030	266	930	698	1196	1030	1030

Tabela 6.5: Exemplo de uma tabela inválida de logs do controlador

	Estado s0		Estado s1		Estado s2		Estado s3		Estado s4		Estado s5	
Nodos	**Rec.**	**Env.**	**Rec.**	**Env.**	**Rec.**	**Env.**	**Rec.**	**Env.**	**Rec.**	**Env.**	**Rec.**	**Env.**
67108874 – 33554442	1362	698	764	598	1528	1030	1030	864	698	366	1096	598
67108874 – 50331658	698	8646	1030	1196	1196	1030	1262	1096	1196	1362	1096	1096
67108874 – 16777226	366	698	764	100	864	200	930	100	1196	532	532	200
33554442 – 50331658	1030	532	598	1096	365	366	267	598	697	698	532	532
33554442 – 67108874	698	1362	598	764	1030	1528	864	1030	366	698	598	1096
33554442 – 16777226	1196	366	532	200	366	366	764	266	200	864	432	266
50331658 – 33554442	532	1030	1097	598	366	366	598	266	698	698	532	532
50331658 – 16777226	698	532	764	266	1030	698	366	532	864	366	930	266
50331658 – 67108874	864	698	1196	1030	1030	1196	1096	1262	1362	1196	1096	1096
16777226 – 50331658	532	698	266	764	698	1030	532	366	366	864	266	930
16777226 – 33554442	366	1196	200	532	366	366	266	764	864	200	266	432
16777226 – 67108874	698	366	100	764	200	864	100	930	532	1196	200	532

33554442) é comparada com a sua inversa (linha *4*, *3355442 – 67108874*) estado a estado para verificar a integridade dos contadores de mensagens de entrada e saída. Percorrida a primeira linha, o processo segue até que todas as linhas tenham sido verificadas, exceto em casos onde há constatação de violação da integridade dos dados, conforme mostrado abaixo e destacado na tabela 6.5.

---

verificando **67108874** e **33554442** com linha **4**
ESTADO s0: analisando linha 0 (in/out: **1362/698**) com linha **4** (in/out: **698/1362**)
ESTADO s1: analisando linha 0 (in/out: **764/598**) com linha **4** (in/out: **598/764**)
ESTADO s2: analisando linha 0 (in/out: **1528/1030**) com linha **4** (in/out: **1030/1528**)
ESTADO s3: analisando linha 0 (in/out: **1030/864**) com linha **4** (in/out: **864/1030**)
ESTADO s4: analisando linha 0 (in/out: **698/366**) com linha **4** (in/out: **366/698**)
ESTADO s5: analisando linha 0 (in/out: **1096/598**) com linha **4** (in/out: **598/1096**)
OK
verificando **67108874** e **50331658** com linha **8**
ESTADO s0: analisando linha **1** (in/out: **698/864**) com linha **8** (in/out: **864/698**)
ESTADO s1: analisando linha **1** (in/out: **1030/1196**) com linha**8** (in/out: **1196/1030**)
ESTADO s2: analisando linha **1** (in/out: **1196/1030**) com linha **8** (in/out: **1030/1196**)
ESTADO s3: analisando linha **1** (in/out: **1262/1096**) com linha **8** (in/out: **1096/1262**)
ESTADO s4: analisando linha **1** (in/out: **1196/1362**) com linha **8** (in/out: **1362/1196**)
ESTADO s5: analisando linha **1** (in/out: **1096/1096**) com linha **8** (in/out: **1096/1096**)
OK
verificando **67108874** e **16777226** com linha **11**
ESTADO s0: analisando linha **2** (in/out: **366/698**) com linha **11** (in/out: **698/366**)
ESTADO s1: analisando linha **2** (in/out: **764/100**) com linha **11** (in/out: **100/764**)
ESTADO s2: analisando linha **2** (in/out: **864/200**) com linha **11** (in/out: **200/864**)
ESTADO s3: analisando linha **2** (in/out: **930/100**) com linha **11** (in/out: **100/930**)
ESTADO s4: analisando linha **2** (in/out: **1196/532**) com linha **11** (in/out: **532/1196**)
ESTADO s5: analisando linha **2** (in/out: **532/200**) com linha **11** (in/out: **200/532**)
OK
verificando **33554442** e **50331658** com linha **6**
ESTADO s0: analisando linha **3** (in/out: **1030/532**) com linha **6** (in/out: **532/1030**)

ESTADO s1: analisando linha **3** (in/out: **598/1096)** com linha **6**(in/out: **1097/598)**
FALHOU

***** **EXPERIMENTO INVÁLIDO** *****

Figura 6.20: Saída gerada pelo protótipo *PIE* após o fim do experimento.

Em virtude da invalidez do experimento, o mesmo deve ser descartado e executado novamente pelo engenheiro de testes.

6.7.1 Conclusões Parciais

O último ciclo de experimentos caracterizou a avaliação da efetividade da solução adotada para detectar violações de propriedade durante o experimento, de modo que este não seja aceito pelo engenheiro de testes e interfira em suas avaliações posteriores.

Durante os experimentos realizados com carga de trabalho relativamente baixa (linhas 1 e 2 - tabela 6.3), a solução portou-se de forma bastante eficaz. Isto, pois, todas as execuções, em partida quente, foram executadas com sucesso.

Ao aumentar mais que o dobro do valor da carga de trabalho em cada nodo (5000 mensagens em 60 segundos), o retrospecto permaneceu positivo, visto que foram necessárias apenas 23 execuções para obter-se um total de 20 execuções íntegras do experimento. Entretanto, de uma outra ótica, a intrusividade do ambiente de injeção de falhas *PIE* pode ser muito superior aos valores citados acima, pois há uma perda de produtividade ao forçar 3 execuções além das 20 necessárias. O mesmo ocorre ao dobrar a carga de trabalho, desta vez para 10 mil mensagens por minuto, que foram necessárias 56 execuções para que se obtivesse 20 execuções válidas.

Este aumento do número de violações, detectadas à medida que cresce a carga de trabalho da aplicação-alvo, pode ser resultado da grande demanda de processamento submetido à camada inserida na pilha de protocolos do sistema operacional. Logo, o enfileiramento de mensagens pode resultar no atraso de processamento da mensagem e, por conseguinte, no atraso da atualização dos contadores do módulo de violação de integridade. Outro fator também está vinculado ao aumento do número de mensagens da aplicação-alvo, que pode causar congestionamento no *switch* de comunicação.

Apesar de apresentar estes resultados, as violações detectadas permanecem em valores que variam de 1 à 10 mensagens para mais ou para menos em relação ao seu respectivo contador inverso, conforme visto destacado na tabela 6.5, o que é uma margem muito baixa quando levado em consideração o fluxo total de mensagens entre os nodos em teste.

6.8 Observações sobre Experimentos

Durante a realização dos experimentos, uma série de detalhes merecem atenção. Dentre eles, os mais notáveis transcorreram no início dos primeiros testes, quando estavam sendo utilizados dois nodos Intel quad Core em conjunto com os descritos na tabela 6.1.

No decorrer dos primeiros experimentos, os nodos quad Core, estavam visivelmente atrasados em relação aos nodos dual Core, com tempos de execução variando entre 5 e 10 segundos acima da média dos demais nodos sem motivo aparente.

Com pouco tempo para explorar tal anomalia, os nodos quad Core foram descartados da etapa experimental deste trabalho. Entretanto, após definido o processo de automação

de testes, notou-se que o nodo *porsche* apresentou o mesmo comportamento dos nodos quad Core. Logo, é possível perceber que este comportamento estava ocorrendo logo após determinadas alterações de horário dos nodos, para que o processo de automação de testes permanecesse organizado.

O acompanhamento do horário atual é uma das inúmeras tarefas do *kernel* do Linux. Neste nível de abstração, tal tarefa é gerenciada de forma extremamente dependente da arquitetura, visto que cada arquitetura apresenta uma maneira específica para tratar a resolução do tempo, bem como o processamento dos cálculos necessários. Apesar de funcional, este mecanismo vem sendo reutilizado entre novas arquiteturas, tornando-se um ponto frágil, uma vez que os *patches* responsáveis por alterar códigos relacionados ao tempo podem não atualizar corretamente todas as arquiteturas. Por fim, todas as máquinas foram reiniciadas e o problema desapareceu até uma eventual atualização de horário dos nodos.

Outro ponto, menos crítico, é a capacidade de recuperação de falhas da aplicação *Draw* que possui um *buffer* com pequena capacidade para o armazenamento de mensagens. Logo, em eventos de particionamento de rede com intervalos muito grandes, a aplicação não consegue recuperar-se adequadamente da falha, visto que o *buffer* descarta novas mensagens quando está operando acima de sua capacidade. Sendo assim, as descrições das cargas de falhas nos experimentos focaram-se em curtos intervalos entre falhas, para que os resultados pudessem ser melhor visualizados.

6.9 Conclusões de Capítulo

Este capítulo apresentou os processos de validação e avaliação do injetor de falhas *PIE*. Para isso, foram realizadas duas campanhas: (*i*) uma através da aplicação *Topology*, que reproduziu o cenário conforme a descrição da carga de falhas estipulada, e a segunda campanha (*ii*) através da aplicação *Draw*, que serviu de base para avaliação da intrusividade do protótipo *PIE* sobre o ambiente de testes. Por fim, uma terceira campanha foi realizada para fins de prova de conceito do mecanismo de detecção de violações durante os experimentos.

As avaliações do módulo de verificação de integridade, por sua vez, não incluem o controle sobre falso positivo de violações. Desta forma, é possível que o engenheiro de testes obtenha resultados falso positivo. Entretanto, *PIE* continua detectando quaisquer tipos de violação e a extensão para este tipo de verificação será retomada em trabalhos futuros.

Com relação aos dados coletados dos experimentos de avaliação de intrusividade, foi possível perceber que a aplicação-alvo não apresentou atrasos, ou pontos de sobrecarga, que pudessem alterar seu fluxo de execução. Embora sejam voltados para propósitos diferentes, alguns testes foram feitos com a aplicação-alvo operando com o protótipo *PIE* e a ferramenta de análise de protocolos de rede *Wireshark*. Neste caso, foi detectada uma variação da média de tempo de execução que se manteve por volta de 8 segundos. Isto, pois, a execução simultânea das duas ferramentas, *PIE* e *Wireshark*, implicam na passagem do fluxo de pacotes da rede por um número maior de etapas, visto que o *Wireshark* provê uma série de filtros e afins que podem atrasar o processamento dos mesmos.

O capítulo a seguir finda este trabalho com a relação das conclusões, resultados alcançados e trabalhos futuros.

7 CONSIDERAÇÕES FINAIS

7.1 Conclusões

Avaliar o comportamento sob falhas de um sistema computacional de comunicação, com elevados requisitos de dependabilidade, é uma realidade. Para tanto, injeção de falhas de comunicação mostra-se como uma técnica bastante eficaz para a realização de testes sobre as técnicas de tolerância a falhas adotadas no projeto de desenvolvimento de uma ferramenta de comunicação.

Tratando-se de comunicação, diversos sistemas estão fortemente ligados e dependem da participação mútua das demais instâncias em diversas ações, como por exemplo um algoritmo de consenso, um sistema de consistência de réplicas, entre outros. As aplicações-alvo que implementam estes mecanismos, por sua vez, apresentam grandes dificuldades de serem testadas em ambientes específicos de maneira adequada. Dentre as necessidades mais básicas, como a injeção de falhas de comunicação para alteração do fluxo de mensagens sobre o canal de comunicação, tais mecanismos enfrentam um tipo peculiar de falha em sistemas distribuídos, o particionamento de rede.

Quando particionada, a instância de uma aplicação-alvo deixa de se comunicar com as demais instâncias e dá origem às inconsistências entre as mesmas, visto que o processamento continua mesmo na ausência de comunicação. Desta forma, as aplicações-alvo desenvolvem mecanismo para o tratamento desta natureza de falhas que, por conseguinte, precisam ser avaliadas.

Partindo desta premissa, este trabalho contribuiu com o estudo de diversos injetores de falhas de comunicação encontrados na literatura. Seu uso permite a emulação de ambientes específicos de forma controlada, para que a aplicação-alvo possa ser avaliada sob situações anômalas sem a necessidade de utilizar mecanismos pouco ortodoxos, como a desconexão manual de cabeamento, alteração de regras de *firewall*, entre outros. Logo, a opção pela realização de testes de maneira controlada garante alta capacidade de reexecução fiel de experimentos, e obtenção de métricas que podem auxiliar na previsão de falhas que não foram tratadas ou identificadas em tempo de desenvolvimento.

Contudo, injetar falhas de comunicação não necessariamente diz respeito a capacidade do injetor de falhas emular falhas peculiares a ambientes distribuídos, como o particionamento de rede, foco principal deste trabalho. Na maioria dos casos este cenário não se reproduz. Em outros, os injetores possuem potencial para emular falhas de particionamento de rede, mas trazem consigo todas as dificuldades necessárias para tal, demandando grandes esforços do engenheiro de teste. Esta, e outras limitações impostas pelos injetores encontrados na literatura motivaram o desenvolvimento de uma nova solução voltada estritamente para testes de quaisquer aplicações distribuídas baseadas em *IPv4/IPv6* sensíveis ao particionamento de rede.

Como maior contribuição, este trabalho apresentou uma solução para injeção de falhas de particionamento de rede em aplicações distribuídas. Tal solução opera no nível de *kernel*, ou seja, atua sob a pilha de protocolos do sistema operacional, sem o consentimento da aplicação-alvo. Logo, é possível emular falhas de particionamento de rede e coletar dados experimentais para avaliação de integridade dos experimentos com mínima intrusividade. Em um nível mais alto atua um *parser*, responsável pela validação dos cenários de falhas descritos pelo engenheiro de testes, atendendo aos requisitos identificados para o desenvolvimento da solução.

Esta pesquisa deu origem ao protótipo *PIE*, um injetor de falhas de particionamento de rede desenvolvido a partir das experiências retiradas de outros injetores (DAWSON; JAHANIAN, 1995; NEOGI; DE; CHIUEH, 2003; CHANDRA et al., 2004; DREBES, 2005) para o cumprimento dos objetivos definidos ao longo deste trabalho. Além disso, este protótipo incorpora mecanismos de verificação de integridade de experimentos, inexistentes nos injetores de falhas atuais. Para avaliação da solução foram realizados experimentos com diferentes aplicações-alvo e cenários de falha, levando em consideração sua funcionalidade, intrusividade, e integridade dos resultados obtidos. Ao fim dos experimentos, é possível comprovar a viabilidade e utilidade do modelo e da arquitetura de *PIE*.

7.2 Trabalhos Futuros

Apesar de suportar apenas falhas de particionamento de rede, o protótipo *PIE* já suporta a descrever outros tipos de falhas de comunicação, como descarte, atraso, duplicação de mensagens, entre outros. Com isso, visa-se expandir a área de cobertura de aplicações-alvo suportadas pelo injetor de falhas a fim de prover uma única ferramenta de injeção de falhas ao engenheiro de testes que permita avaliar todas as características necessárias da aplicação-alvo.

Outra possível expansão do protótipo *PIE* inclui o suporte ao gerenciamento de ambientes dinâmicos, ou seja, com a entrada e saída de nodos espontaneamente. Logo, será possível avaliar aplicações-alvo operando sobre ambientes móveis e suscetíveis a falhas de particionamento de rede.

Além disso, é prevista a adição de rotinas automáticas para facilitar o processo de descarte de experimentos inválidos e a reexecução dos mesmos por parte do engenheiro de testes, como também prover uma interface amigável para apresentação dos resultados de forma clara e compreensível.

Por fim, estuda-se a necessidade e possibilidade de incorporar mecanismos de injeção de mensagens para geração de fluxo, visto que tal extensão pode ser útil para emulação de ataques de segurança. A necessidade do mesmo em injetores de falhas de comunicação foi apontada em trabalhos que utilizaram *FIRMAMENT* para testes de vulnerabilidades de segurança (MARTINS; MORAIS; CAVALLI, 2008).

REFERÊNCIAS

ACRI, E. **Network manipulation in a hex fashion**: an introduction to hexinject. [S.l.: s.n.], 2010. Disponível em: <http://hexinject.sourceforge.net/hexinject_introduction.pdf>. Acesso em: fev. 2011.

AIDEMARK, J. et al. GOOFI: generic object-oriented fault injection tool. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2001. p.83–88.

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

BIRMAN, K. P. **Building secure and reliable network applications**. Greenwich, CT, USA: Manning Publications Co., 1997.

BIRMAN, K. P.; JOSEPH, T. A. **Reliable Communication in the Presence of Failures**. Ithaca, NY, USA: [s.n.], 1985.

BUCHACKER, K.; SIEH, V. Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects. **IEEE International Symposium on High-Assurance Systems Engineering**, Los Alamitos, CA, USA, p.95, 2001.

BUMBULIS, P.; COWAN, D. D. RE2C: a more versatile scanner generator. **ACM Letters on Programming Languages and Systems**, New York, NY, USA, v.2, n.1-4, p.70–84, 1993.

CARREIRA, J.; MADEIRA, H.; SILVA, J. G. Xception: a technique for the experimental evaluation of dependability in modern computers. **IEEE Transactions on Software Engineering**, Piscataway, NJ, USA, v.24, n.2, p.125–136, 1998.

CARSON, M.; SANTAY, D. NIST Net: a linux-based network emulation tool. **SIGCOMM Computer Communication Review**, New York, NY, USA, v.33, n.3, p.111–126, 2003.

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

CHOCKLER, G. V.; KEIDAR, I.; VITENBERG, R. Group communication specifications: a comprehensive study. **ACM Computing Surveys**, New York, NY, USA, v.33, n.4, p.427–469, 2001.

CLARK, J. A.; PRADHAN, D. K. Fault Injection: a method for validating computer-system dependability. **IEEE Computer**, [S.l.], v.28, n.6, p.47–56, 1995.

DAWSON, S.; JAHANIAN, F. Probing and fault injection of protocol implementations. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1995. p.351.

DAWSON, S.; JAHANIAN, F.; MITTON, T. **ORCHESTRA**: a fault injection environment for distributed systems. [S.l.]: INTERNACIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, XXVI, 1996.

DAWSON, S.; JAHANIAN, F.; MITTON, T. Experiments on six commercial TCP implementations using a software fault injection tool. **Software Practice and Experience**, New York, NY, USA, v.27, n.12, p.1385–1410, 1997.

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

GÄRTNER, F. C. Fundamentals of fault-tolerant distributed computing in asynchronous environments. **ACM Computing Surveys**, New York, NY, USA, v.31, n.1, p.1–26, 1999.

GERCHMAN, J.; WEBER, T. S. Emulando o Comportamento de TCP/IP em um Ambiente com Falhas para Teste de Aplicações de Rede. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, Curitiba, BR. **Anais...** [S.l.: s.n.], 2006.

HEMMINGER, S. Network Emulation with NetEm. In: AUSTRALIA'S NATIONAL LINUX CONFERENCE, VI, Australia. **Proceedings...** [S.l.: s.n.], 2005. Disponível em: <http://devresources.linuxfoundation.org/shemminger/netem/LCA2005_paper.pdf>. Acesso em: fev. 2011.

HIPP, R. **The Lemon Parser Generator**. [S.l.: s.n.], 2008. Disponível em: <http://www.hwaci.com/sw/lemon/lemon.html>. Acesso em: fev. 2011.

HOARAU, W.; TIXEUIL, S. A Language-Driven Tool for Fault Injection in Distributed Systems. In: IEEE/ACM INTERNATIONAL WORKSHOP ON GRID COMPUTING, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2005. p.194–201.

HSUEH, M.; TSAI, T. K.; IYER, R. K. Fault Injection Techniques and Tools. **IEEE Transactions Computers**, Los Alamitos, CA, USA, v.30, n.4, p.75–82, 1997.

JACQUES-SILVA, G. et al. FIONA: a fault injector for dependability evaluation of java-based network applications. In: NETWORK COMPUTING AND APPLICATIONS, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2004. p.303–308.

JACQUES-SILVA, G. et al. A Network-Level Distributed Fault Injector for Experimental Validation of Dependable Distributed Systems. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2006. p.421–428.

JUANG, T.-Y.; VENKATESAN, S. Crash recovery with little overhead. **International Conference on Distributed Computing Systems, XI**, Arlington, TX, USA, p.454–461, 1991.

LEFEVER, R. M. et al. An Experimental Evaluation of Correlated Network Partitions in the Coda Distributed File System. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS. **Proceedings...** John Wiley & Sons, 2003. p.273–282.

LI, X. et al. Mendosus: a san-based fault-injection test-bed for the construction of highly available network services. In: WORKSHOP ON NOVEL USES OF SYSTEM AREA NETWORKS. **Proceedings...** [S.l.: s.n.], 2002.

LOOKER, N.; MUNRO, M.; XU, J. A Comparison of Network Level Fault Injection with Code Insertion. In: COMPUTER SOFTWARE AND APPLICATIONS CONFERENCE, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2005. v.1, p.479–484.

MARTINS, E.; MORAIS, A.; CAVALLI, A. Generating attack scenarios for the validation of security protocol implementations. In: BRAZILIAN WORKSHOP ON SYSTEMATIC AND AUTOMATED SOFTWARE TESTING, Campinas, Brazil. **Anais...** SBC, 2008. p.21–33.

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, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2002. p.483–482.

MURAYAMA, Y.; YAMAGUCHI, S. DBS: a powerful tool for tcp performance evaluations. In: PERFORMANCE AND CONTROL OF NETWORK SYSTEMS. **Proceedings...** [S.l.: s.n.], 1997.

NEOGI, A.; DE, P.; CHIUEH, T. cker. VirtualWire: a fault injection and analysis tool for network protocols. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2003. p.214.

NEVES, N.; FUCHS, W. K. Fault Detection Using Hints from the Socket Layer. In: SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1997. p.64.

OLIVEIRA, G. M.; CECHIN, S. L.; WEBER, T. S. Injeção Distribuída de Falhas de Comunicação com Suporte a Controle e Coordenação de Experimentos. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, João Pessoa, Brazil. **Anais...** SBC, 2009. p.101–114.

OLIVEIRA, G. M.; CECHIN, S. L.; WEBER, T. S. Dependability Evaluation of Distributed Systems through Partitioning Fault Injection. In: LATIN-AMERICAN TEST WORKSHOP, Puntal del Este, Uruguay. **Proceedings...** IEEE Computer Society, 2010. p.7–12.

RED HAT. **Reliable Multicasting with the JGroups Toolkit**. 1.13.ed. [S.l.: s.n.], 2009. Disponível em: <`http://www.jgroups.org/manual/pdf/manual.pdf`>. Acesso em: fev. 2011.

RIZZO, L. Dummynet: a simple approach to the evaluation of network protocols. **SIGCOMM Computer Communication Review**, New York, NY, USA, v.27, n.1, p.31–41, 1997.

RUSSEL, R.; WELTE, H. **Linux netfilter Hacking**. [S.l.: s.n.], 2002. Disponível em: <http://netfilter.org/documentation/HOWTO/ /netfilter-hacking-HOWTO.a4.ps>. Acesso em: fev. 2011.

STOTT, D. T. et al. NFTAPE: a framework for assessing dependability in distributed systems with lightweight fault injectors. In: INTERNATIONAL COMPUTER PERFORMANCE AND DEPENDABILITY SYMPOSIUM, IV, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2000. p.91.

STROM, R.; YEMINI, S. Optimistic recovery in distributed systems. **ACM Transactions on Computer Systems**, New York, NY, USA, v.3, n.3, p.204–226, 1985.

TANENBAUM, A. S.; STEEN, M. v. **Distributed Systems**: principles and paradigms (2nd edition). Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.

TSAI, T. K. et al. Stress-Based and Path-Based Fault Injection. **IEEE Transactions Computers**, Washington, DC, USA, v.48, n.11, p.1183–1201, 1999.

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

VERÍSSIMO, P.; RODRIGUES, L. **Distributed Systems for System Architects**. 1.ed. Boston, USA: Springer, 2001. 648p.

WIERMAN, S. J. **Vajra**: distributed fault injection for dependability evaluation. 2006. 40 f. Dissertação (Mestrado em Ciência da Computação) — Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.