Aumentando a Expressividade da Descrição de Cargas de Falhas de Comunicação para Testes com Injetores de Falhas

Ruthiano S. Munaretti, Taisy S. Weber, Sérgio L. Cechin

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{rsmunaretti,taisy,cechin}@inf.ufrgs.br

Abstract. *Fault injection causes faults in a controlled way on a target system during a test experiment to investigate the behavior of this system under fault conditions. The faultload description format used in an experiment influences directly the viability of emulation of this faultload and the relevance/validity of the results obtained with the experiment. This paper presents an approach, based on Java programming language, to describe communication faultloads. The main goal of this approach is the improvement of flexibility and expressiveness in faultload specification, which makes easier the execution of test experiments, allowing faultloads reuse and its conversion to specific input formats of several communication fault injectors.*

Resumo. *Injeção de falhas visa provocar falhas de forma controlada em uma aplicação alvo durante um experimento de teste, de forma a investigar o comportamento da mesma sob falhas. A forma pela qual uma carga de falhas é descrita para um experimento influi diretamente na viabilidade da emulação desta carga de falha e na relevância e validade dos resultados alcançados com o experimento. Este artigo apresenta uma abordagem, baseada na linguagem de programação Java, para a descrição de cargas de falhas de comunicação. O principal objetivo desta descrição consiste em aumentar a flexibilidade e a expressividade na descrição de cargas de falhas, facilitando a execução dos experimentos de teste, permitindo o reuso de cargas de falhas e sua tradução para formatos de entrada específicos de diferentes injetores de falhas de comunicação.*

1. Motivação

Segundo [Avizienis et al. 2001], dependabilidade indica o nível de confiança que pode ser justificadamente colocada em um sistema. Entre as técnicas existentes para realizar uma avaliação de dependabilidade de um determinado sistema computacional, pode ser destacada a *injeção de falhas* [Hsueh et al. 1997]. Injeção de falhas é uma técnica que tem como objetivo provocar falhas de forma *controlada* em um determinado sistema alvo. A partir desta injeção, pode-se investigar o comportamento do sistema durante a presença de falhas, verificando o seu funcionamento, bem como a eficiência e correção dos mecanismos de tolerância a falhas implementados.

No contexto de sistemas distribuídos, as principais falhas são de *comunicação*. Por falhas de comunicação, entende-se todas as falhas que envolvem, no todo ou em parte, a rede de comunicação no qual um determinado sistema distribuído é executado.

Assim, como exemplos de falhas de comunicação, podem ser destacados o colapso, em um determinado momento, de algum *caminho* na rede de comunicação, assim como o atraso ou perda dos pacotes que trafegam por este caminho, além do colapso de nodos que formam uma rede.

Ao realizar injeção de falhas em uma determinada aplicação, uma característica que deve ser levada em consideração diz respeito a *forma* pela qual uma **carga de falhas** é descrita. Esta característica é importante, visto que a mesma define o *escopo* do experimento, ou seja, que tipos de falhas devem ser injetadas e quais devem ser desconsideradas. Vale ressaltar que um cenário de falhas é dividido em duas partes: uma *carga de trabalho* (que consiste na execução da aplicação em si, com dados de entrada e/ou estímulos adequados) e uma *carga de falhas* (abrangendo a emulação das falhas injetadas). Além disso, outro quesito importante diz respeito a *extensão* de um determinado modelo de falhas utilizado. Esta extensão, por sua vez, permite adaptar o modelo em questão para a realidade do experimento que está sendo realizado.

Neste sentido, um ambiente para criação de cargas de falhas foi previamente modelado e definido [Munaretti and Weber 2008]. O principal objetivo deste ambiente é facilitar a criação de experimentos utilizando diversos injetores, partindo-se do pressuposto que é necessária apenas a descrição de uma carga de falhas genérica. Assim, fica a cargo deste ambiente a conversão para as diversas cargas específicas, referentes a cada injetor que possa vir a ser usado.

No processo de definição do ambiente citado anteriormente, o mecanismo de descrição de cargas de falhas não havia sido modelado para casos complexos de teste. Visando suprir esta demanda, o presente artigo detalha um mecanismo para a descrição das cargas de falhas deste ambiente. Neste sentido, será utilizada uma abordagem baseada em *linguagem de programação*, uma vez que a mesma atende, de forma adequada, aos requisitos de *flexibilidade* e *expressividade*, desejáveis para a criação de cargas de falhas complexas.

O restante do artigo está organizado da seguinte maneira: a seção 2 apresenta trabalhos relacionados, enquanto que a seção 3 faz uma revisão do ambiente no qual a linguagem descrita neste artigo está inserida. A seção 4, por sua vez, realiza uma descrição da linguagem apresentada e, em seguida, a seção 5 apresenta alguns exemplos da mesma. A seção 6 finaliza o artigo, com conclusões a respeito do trabalho realizado.

2. Injetores de Falhas de Comunicação

Referente a descrição de carga de falhas, os injetores de falhas existentes fazem uso de abordagens distintas. A seção atual visa descrever injetores que utilizam abordagens facilitadas para esta descrição, de forma a compará-las com o ambiente proposto. Neste sentido, serão abordados dois injetores locais (FIONA e FIRMAMENT), além de três injetores conhecidos na literatura (DOCTOR, Mendosus e FAIL/FCI).

FIONA [Jacques-Silva et al. 2004] (*Fault Injector Oriented to Network Applications*) é um injetor de falhas em sistemas distribuídos, com foco no protocolo UDP. A abordagem usada no mesmo consiste na instrumentação de código, utilizando-se para isso de uma ferramenta de instrumentação em Java, denominada JVMTI (*Java Virtual Machine Tool Interface*). O modelo de falhas adotado neste injetor é o proposto por Birman [Birman 1996], com suporte a particionamento de rede. Para a criação de cargas de

falhas, FIONA utiliza um *arquivo de configuração*, instanciando-se um objeto para cada falha especificada, o que torna simplificada a descrição das falhas a serem injetadas.

FIRMAMENT [Drebes 2005] (*Fault Injection Relocatable Module for Advanced Manipulation and Evaluation of Network Transports*) é um injetor de falhas que trabalha no nível de sistema operacional, injetando falhas a partir de módulos do kernel Linux. Para a descrição de cargas de falhas, FIRMAMENT utiliza a abordagem de *faultlet*, uma linguagem tipo *assembler* utilizada para emular comportamentos de falhas. Apesar de existir uma certa complexidade na aprendizagem da ferramenta (devido à linguagem adotada pelos *faultlets*), esta abordagem proporciona um grande poder de expressividade à ferramenta.

DOCTOR [Han et al. 1995] consiste em um ambiente de injeção de falhas, com foco em sistemas distribuídos de tempo real. Quanto ao modelo de falhas adotado, DOCTOR é capaz de emular falhas de hardware tradicionais, bem como falhas de comunicação. Referente aos quesitos de carga de falhas e extensibilidade, DOCTOR apresenta um mecanismo adequado para a elaboração de carga de falhas, assim como pontos de extensão bem definidos. Entretanto, a extensibilidade em si é limitada, devido ao foco específico do injetor (em sistemas de tempo real), além da complexidade inerente ao mesmo.

Mendosus [Li et al. 2002] é uma ferramenta para injeção de falhas em redes do tipo *SAN* (System-Area Networks). Assim como FIRMAMENT, este injetor aplica as falhas no nível de kernel, através da extensão de classes do kernel Linux. Diferentemente de FIRMAMENT, Mendosus utiliza chamadas de alto nível para a descrição de carga de falhas. Entretanto, o modelo de falhas adotado por Mendosus é limitado, admitindo apenas falhas relacionadas a colapso.

A ferramenta FAIL/FCI [Hoarau and Tixeuil 2005] é usada para injeção de falhas em aplicações de *cluster* e *peer-to-peer* (P2P). Por utilizar uma abordagem de *linguagem de autômatos* para a descrição de carga de falhas, esta ferramenta se apresenta como a mais próxima à abordagem proposta neste artigo. Entretanto, assim como o injetor Mendosus, esta ferramenta possui um modelo de falhas limitado apenas às falhas de colapso.

De todas as ferramentas citadas, as descrições de carga de falhas de FIRMAMENT e FAIL/FCI são as que mais se aproximam ao trabalho proposto. Isto pode ser constatado ao se considerar os requisitos de *flexibilidade* e *expressividade*. Enquanto FIRMAMENT apresenta vantagens em relação ao poder de expressividade, a linguagem de alto nível adotada por FAIL/FCI proporciona uma maior flexibilidade na sua utilização. Neste sentido, a descrição adotada visa atingir estes dois requisitos simultaneamente, algo ainda escasso nas ferramentas atuais.

3. Ambiente para Criação de Cargas de Falhas

Visando uma utilização abrangente e efetiva de testes por injeção de falhas, um ambiente para criação de cargas de falhas deve permitir descrever a maior parte dos casos possíveis de falhas. A figura 1 ilustra uma instância do modelo de framework construído para o ambiente de descrição e geração de cargas de falhas deste trabalho. Como é possível visualizar na figura, os componentes e sub-componentes do ambiente são divididos em camadas, enquanto que a seta à direita do ambiente indica o nível de cada camada (neste caso, do mais alto para o mais baixo nível).

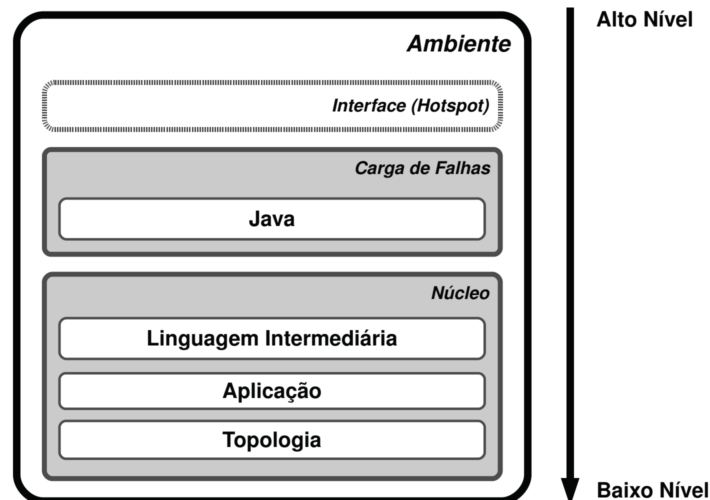

Figura 1. Instância do framework do ambiente.

O *núcleo*, por ser o componente de mais baixo nível, define todas as operações básicas implementadas pelo modelo. Logo, o núcleo será o responsável pela delimitação das falhas a serem emuladas, criando-se assim o *escopo* das falhas que poderão ser injetadas em um determinado experimento. Por questões de desempenho, o núcleo é modelado utilizando-se de uma abordagem simplificada, formado por três construções: **Topologia**, **Aplicação** e **Linguagem Intermediária**. Estas construções são explicadas nos itens que seguem.

- **Topologia:** como o nome indica, a topologia é a responsável por todos os elementos de rede que integram o respectivo experimento a ser realizado. Além disso, a mesma abrange também todas as interações necessárias para uma efetiva comunicação destes elementos. No contexto do modelo do ambiente, a topologia é composta por três elementos: *Nodos*, *Caminhos* e *Pacotes*. Estes elementos são refletidos diretamente na linguagem proposta neste trabalho. Por este motivo, a seção 4.2 explica em detalhes cada um destes elementos.
- **Aplicação:** representa a aplicação alvo sob teste, que é executada na topologia informada anteriormente. Além das configurações referentes a *chamada* desta aplicação, este componente também é responsável pelas configurações de inicialização da respectiva aplicação, além de outras informações relevantes (como em quais nodos da topologia executar a aplicação, por exemplo).
- **Linguagem Intermediária:** consiste em uma linguagem de baixo nível, cujo principal objetivo é facilitar a transformação de uma carga de falhas descrita em Java (foco deste trabalho) para as cargas de falhas correspondentes aos injetores escolhidos. Para isso, esta linguagem possui uma construção simplificada, utilizando-se apenas de uma *tabela de símbolos* (para armazenamento de dados) e seis comandos: INSERT (adiciona um valor na tabela de símbolos), RUN (executa a aplicação alvo especificada), RAND (gera um número aleatório, a partir de um valor "semente" pré-configurado), EVAL (avalia uma expressão), GOTO (direciona a execução para uma determinada linha da linguagem intermediária), DROP (descarta um pacote) e DELAY (atrasa um pacote).

Funcionando como uma entrada de dados ao ambiente, a *carga de falhas* visa especificar *quais* falhas estarão ativas em um dado experimento, bem como qual a *configuração* de cada falha no respectivo experimento. Esta carga de falhas é genérica, não sendo assim restrita às peculiaridades de uma determinada ferramenta. Este componente é formado pelo módulo de apoio a **Java**, foco deste artigo. A seção 4 descreve em detalhes como este módulo de apoio, bem como características adicionadas a esta linguagem, foram incluídas e adaptadas ao ambiente.

Representando o *ponto de extensão* do framework, a *interface* tem por objetivo finalizar a transformação da carga de falhas genérica para a carga de falhas específica do injetor utilizado. Para isso, o ambiente modela a interface como uma outra entrada de dados definida pelo usuário, seguindo-se a abordagem de *plugin*, através de um script de mapeamento. Neste script, cada linha representa um mapeamento, separado pelo caractere ":", correspondente a uma falha primitiva/composta do ambiente ("<FaultLabel>"), seguido do respectivo comando a ser chamado no injetor específico ("<Command>"). A sintaxe deste script pode ser visualizada na figura 2.

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

Figura 2. Mapeamento - falha primitiva/composta : descrição do injetor

Finalmente, a figura 3 ilustra o funcionamento deste ambiente. As especificações da *topologia* e *aplicação alvo* são fornecidas como entradas do ambiente, assim como a própria *carga de falhas* (definida neste trabalho e descrita na linguagem Java). Estas entradas são submetidas a uma série de compiladores (correspondentes aos componentes e construções descritos anteriormente), de forma que, ao final do processamento, várias cargas de falhas são geradas, sendo uma para cada injetor de falhas que venha a ser utilizado pelo engenheiro de testes na condução do experimento de injeção de falhas.

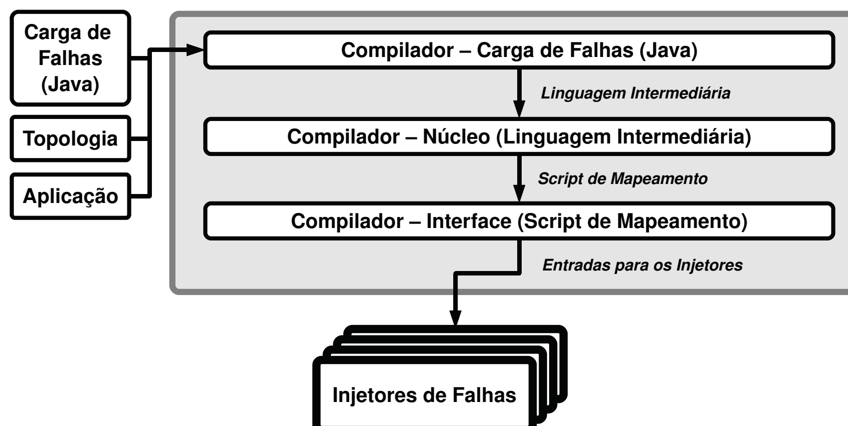

Figura 3. Funcionamento do ambiente

4. Escolha da Forma de Descrição de Falhas

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

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

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

Assim, esta seção visa descrever a linguagem utilizada no ambiente modelado [Munaretti and Weber 2008]. Esta linguagem é de fundamental importância para o ambiente, visto que ela possibilita a criação de cargas de falhas de forma direta pelo criador de um experimento de testes. Além disso, a linguagem proporciona o atendimento aos requisitos de flexibilidade e expressividade, definidos anteriormente.

A linguagem escolhida é um subconjunto da linguagem de programação Java. As principais decisões que justificaram a adoção de Java para a composição deste ambiente são explicitadas nos itens que seguem.

- **Popularidade da linguagem:** Java é amplamente utilizada, tanto no meio acadêmico como profissional, para a realização de tarefas nos mais diferentes níveis de complexidade. Isto facilita a curva de aprendizado da descrição de cargas de falhas e, por consequência, permite que o foco do usuário do ambiente possa estar mais voltado às tarefas realmente relevantes, relativas ao experimento de testes.
- **Orientação a objetos:** a orientação a objetos permite uma maior *modularização* das tarefas referentes às cargas de falhas, possibilitando um futuro *reuso* das mesmas.

- **Atendimento aos requisitos de *flexibilidade* e *expressividade:*** a linguagem Java atende, de forma apropriada e precisa, o requisito de flexibilidade, por permitir várias formas de criação de cargas de falhas. Da mesma forma, a demanda de expressividade também é preenchida de forma adequada, pela liberdade e o nível de detalhe que Java possibilita na criação de uma determinada carga de falhas.

4.1. Construções de Java Implementadas

O tempo de aprendizado dispensado em uma linguagem recém-definida é comumente elevado. Logo, com o intuito de reduzir ao máximo este tempo, pode-se contar com o reuso de construções provenientes de linguagens amplamente conhecidas. Além de facilitarem, de forma decisiva, a utilização da linguagem, estas construções reutilizadas são de extrema importância para o aumento de produtividade do usuário, levando à elaboração de experimentos mais complexos em um menor tempo.

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

4.2. Topologias

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

Considerando o ambiente utilizado neste artigo, bem como sua respectiva linguagem, o suporte a definição de topologias é realizado a partir de um conjunto de classes. Estas classes, explicadas nos parágrafos que seguem, são agrupadas em um pacote denominado *env* (um acrônimo de *environment*, indicando as personalizações do ambiente proposto na linguagem Java utilizada). A figura 4 ilustra uma topologia envolvendo as classes citadas, de forma a ilustrar como as mesmas interagem entre si.

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

- *id:* corresponde ao identificador único do pacote;
- *data:* representam os dados transportados pelo pacote;
- *protocol:* indica o protocolo utilizado;
- *state:* mostra o estado em que se encontra o pacote, podendo ser um dos seguintes: *em envio*, *em transmissão* ou *em recebimento*).

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

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

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

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

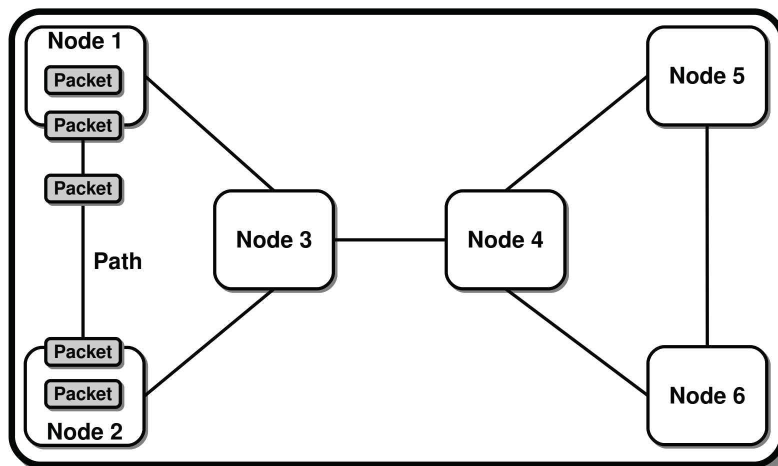

Figura 4. Interação existente entre as classes *Node*, *Path* e *Packet*

Além dos atributos, cada uma das classes acima possui a definição de um conjunto de métodos. Tais métodos referem-se ao conjunto de falhas que podem ser injetadas em um experimento. Este conjunto, por sua vez, delimita o *modelo de falhas* do ambiente proposto. As definições destes métodos são elaboradas nos itens seguintes, classificados por *níveis*. Na tabela 1, pode-se visualizar em quais classes estes métodos estão implementados, indicando-se assim os tipos de falhas que podem ser injetadas em cada componente da topologia.

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

- *drop():* realiza o *descarte* do pacote indicado. Assim, a transmissão do respectivo pacote passa a não ser realizada, sem a possibilidade de recuperação posterior do mesmo.
- *delay(<x>):* efetua um *atraso* no pacote sob falhas. Neste caso, a transmissão ainda é realizada, mas após o intervalo especificado.

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

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

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

- *Colapso:* refere-se a interrupção *total* do serviço fornecido pelo componente. Assim como acontece com a falha "*drop()*" (descrita anteriormente), não existe a possibilidade de um retorno posterior deste serviço.
- *Omissão:* ocasiona uma interrupção *parcial* do serviço oferecido pelo componente. Neste caso, o nível desta interrupção pode ser configurado pelo usuário do experimento.
- *Temporização:* proporciona um *atraso* no fornecimento do serviço pelo componente. Assim como a falha anterior, o tamanho deste atraso também é configurável pelo usuário.
- *Aleatório (Bizantino):* uma falha aleatória (também conhecida como *bizantina*) apresenta um comportamento imprevisível. Em comparação com os demais tipos de falhas, esta é a considerado a mais complexa, considerando-se os aspectos de injeção de falhas, podendo englobar todas as demais e ainda alteração do conteúdo de pacotes, duplicação e geração espontânea de novos pacotes.

Tabela 1. Relação entre classes e métodos para injeção de falhas

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

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

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

- **Valor de variáveis:** este é um mecanismo existente de forma natural na linguagem proposta, visto que tal funcionalidade já era existente na linguagem Java. Assim, considerando o ambiente, falhas podem ser injetadas a partir de um certo valor que a respectiva variável avaliada possa assumir durante a execução do experimento.
- **Tempo pré-determinado:** neste trabalho, é possivel injetar falhas considerando um tempo determinado do experimento, bem como entre intervalos pré-definidos. Para isso, é fornecida uma chamada de método, presente em todas as classes de topologia mencionadas anteriormente, denominada *at(x[,y],<classe.falha>)*. Esta chamada indica que uma falha (especificada no parâmetro *<classe.falha>*) pode ser injetada a partir do tempo *x* ou, alternativamente, entre os intervalos *x* e *y*.
- **Distribuições aleatórias:** neste mecanismo, distribuições podem ser utilizadas para a realização de injeção de falhas. Neste sentido, como está se trabalhando com a linguagem Java, qualquer biblioteca disponível para a mesma pode ser utilizada neste contexto. Assim, o usuário do ambiente pode ter a liberdade de utilizar a biblioteca de distribuições mais adequada para a sua respectiva realidade.

5. Exemplos

Esta seção descreve alguns exemplos de cargas de falhas que podem ser descritas e geradas no ambiente mencionado, através da linguagem definida na seção anterior. Para isso, foram definidas três cargas de falhas, onde cada carga ilustra um determinado caso de teste, como ilustrado na tabela 2. Os parágrafos seguintes explicam em detalhes como estas cargas podem ser descritas e geradas no ambiente.

Tabela 2. Exemplos de cargas de falhas, escritas em Java

Omissão

```
import env.Node;

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

Temporização

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

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

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

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

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

Crash/Drop

```
import env.*;

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

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

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

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

    c.crash();
  }
}
```

Na coluna **Omissão**, é ilustrada uma carga de falhas que visa injetar uma falha de omissão. Neste caso, o componente alvo desta injeção é um nodo, representado pela variável “n”. Além disso, o método “setApp” permite especificar a aplicação alvo que será executada durante o experimento de injeção de falhas (no contexto do exemplo, o programa “/opt/Prog1”). Finalmente, a chamada à falha de omissão possui um parâmetro

de inicialização, indicando a taxa de omissão de pacotes em trânsito neste nodo (configurada como 5%).

Tabela 3. Scripts intermediários, gerados a partir das cargas de falhas

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

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

Tabela 4. Scripts de mapeamento, aplicados aos injetores utilizados

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

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

Considerando o funcionamento do ambiente a partir destas cargas de falhas, temos como primeiro passo a geração da *linguagem intermediária*. Neste contexto, a tabela 3 apresenta esta primeira fase da tradução, onde cada classe mencionada foi submetida ao compilador Java existente no ambiente, gerando os respectivos scripts de baixo nível. Vale ressaltar que este script não necessita de intervenção humana - neste caso, a única necessidade de contato com estas chamadas ocorre na elaboração dos scripts de mapeamento, discutidos logo em seguida.

Após obter o script intermediário, correspondente à carga de falhas em Java, o ambiente necessita de um script de mapeamento para dar prosseguimento à tarefa de tradução da carga de falhas. Este script é criado pelo instalador do ambiente, demandando assim pouca ou nenhuma manutenção futura. Além disso, este script é orientado *a injetores*, não tendo assim dependência direta de cada carga de falhas elaborada no ambiente. A tabela 4 ilustra os scripts de mapeamento utilizados para os injetores FAIL/FCI, Mendosus e FIRMAMENT.

Tabela 5. Exemplo de entradas para injetores, referente a classe *Omissão*

<table>
<tr><th></th><th>Omissão</th></tr>
<tr><td>FAIL/FCI</td><td><pre>Daemon Logica_Nodo1 {
  node 1:
    int distribution = FAIL_RANDOM(1, 100);
    distribution <= 5   -> stop, goto 2;
  node 2:
}

Computer Nodo1 {
   program = "/opt/Prog1";
   daemon = "Logica_Nodo1";
}</pre></td></tr>
<tr><td>MENDOSUS</td><td><pre>// Configuração da topologia:
set_host Nodo1 10.0.0.1

// Comandos para executar a aplicação alvo:
set_command 0 "/opt/Prog1" EXEC 0 Nodo1

// Falhas a serem injetadas:
set_fault Nodo1 ft_host_freeze_rec TRANSIENT poisson 0.05</pre></td></tr>
<tr><td>FIRMAMENT</td><td><pre>SET 100 R0   ; R0 = 100; // 100%
RND R0 R1   ; R1 = RND(R0); // Obtém número (semente 100)
SET 95 R0   ; R0 = 95; // 5% de probabilidade
SUB R1 R0   ; R0 = R0 - R1;
JMPN R0 DROP ; se R0 é negativo, descarta o pacote

DROP:
     DRP</pre></td></tr>
</table>

Finalmente, a última etapa do processamento do ambiente refere-se, exatamente, à geração da carga *real* de falhas, que pode assim ser utilizada diretamente nos respectivos injetores escolhidos para a realização do experimento. Logo, após a geração desta carga, o experimento de injeção de falhas estará pronto para ser realizado, considerando-se a carga de falhas, bem como os respectivos procedimentos de configuração e inicialização dos injetores. A tabela 5 ilustra a geração de cargas de falhas para os injetores FAIL/FCI, Mendosus e FIRMAMENT, considerando a carga de falhas referente a **Omissão**, mencionada no início desta seção.

6. Conclusões

Este artigo apresentou uma nova abordagem para a descrição de cargas de falhas, baseada em uma linguagem de programação de alto nível. Para isso, foi utilizado como base o modelo/arquitetura de um ambiente já previamente definido pelos autores, uma vez que este ambiente já tinha sido construído com o propósito de utilizar diversos injetores de falhas. Além de uma breve apresentação deste ambiente, foi mostrada uma descrição detalhada desta linguagem, juntamente com exemplos práticos de aplicação da mesma.

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

Considerando os requisitos de *flexibilidade* e *expressividade*, mencionados na seção 1 e discutidos no decorrer de todo o artigo, pode-se dizer que a linguagem adotada neste trabalho atendeu de forma adequada ambos os requisitos. Neste sentido, Java é **flexível**, por permitir a criação de diferentes modalidades de cargas de falhas com possibilidades de expansões em seu modelo, ao mesmo tempo em que é **expressiva**, devido ao alto nível da linguagem, amplamente conhecida nos dias de hoje.

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

Referente ao nível de usabilidade desejado, a abordagem proposta consegue atender, de maneira adequada, aos principais requisitos de usabilidade definidos na literatura desta área [Ferré et al. 2001, Nielsen 1993]. Neste sentido, a linguagem Java é de *fácil aprendizado*, como já mencionado anteriormente. Ao mesmo tempo, esta abordagem permite a descrição de cargas de falhas de uma forma *eficiente* e *relativamente livre de erros*, como é possível visualizar nos exemplos citados na seção 5. Se considerarmos o público-alvo do ambiente (como desenvolvedores e engenheiros de teste, por exemplo), temos que a abordagem é de *uso agradável*, uma vez que a linguagem Java está muito próxima da realidade deste público.

Ao realizar uma comparação com as ferramentas existentes de injeção de falhas (descritas na seção 2), temos que a abordagem descrita neste trabalho emula um modelo de falhas mais detalhado que os injetores MENDOSUS e FAIL/FCI. Além disso, outra característica presente está relacionada à independência de protocolo e aplicação alvo, o que não ocorre nos injetores DOCTOR (focado em sistemas distribuídos de tempo real) e FIONA (com ênfase no protocolo UDP). Finalmente, a abordagem proposta utiliza uma linguagem de alto nível, facilitando o seu aprendizado, o que é uma dificuldade inerente ao injetor FIRMAMENT, uma vez que o mesmo utiliza-se de uma linguagem de baixo nível para a descrição de cargas de falhas.

Referências

Avizienis, A., Laprie, J. C., and Randell, B. (2001). *Fundamental Concepts of Dependability*. In 01145, T. R., editor, *LAAS-CNRS*, Toulouse, France.

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

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

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

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

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

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

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

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

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

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