Modelo de um Ambiente para Descrição de Cenários Detalhados de Falhas

Ruthiano S. Munaretti, Taisy S. Weber

[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}@inf.ufrgs.br

Abstract. *In message-based systems, fault injection approach causes faults in a controlled way. Thus, the behavior of these systems could be investigated on the presence of faults. However, each fault injector uses a different approach, causing several difficulties on usability of these tools. In this context, the present paper proposes an environment for detailed fault scenarios description, with the explanation of the main components of this environment, as well as the use of them in some fault injectors in the literature.*

Resumo. *Em aplicações baseadas em troca de mensagens, a técnica de injeção de falhas tem como objetivo provocar falhas de forma controlada. Assim, pode-se investigar o comportamento destes sistemas na presença de falhas. Entretanto, a existência de diversos injetores de falhas relacionados a este fim ocasionam uma certa dificuldade, inerente ao uso destes injetores, por adotarem abordagens distintas de funcionamento. Neste contexto, o presente trabalho apresenta um ambiente para descrição de cenários detalhados de falhas, abordando os principais elementos deste ambiente, bem como a aplicação dos mesmos em injetores de falhas existentes na literatura.*

1. Motivação

A realização de testes em sistemas computacionais é uma tarefa essencial e desafiadora. Para a execução desta tarefa, o uso de injetores de falhas é fundamental, visto que a ocorrência de falhas nestes sistemas é inevitável, o que pode ocasionar conseqüências desastrosas ao funcionamento dos mesmos. Ao mesmo tempo, a ocorrência de falhas em uma execução real do sistema tem uma probabilidade muito pequena de ocorrer, o que justifica ainda mais o uso de injetores de falhas, visando assim acelerar este processo em um determinado experimento.

Desta forma, injeção de falhas é uma técnica que tem como objetivo provocar falhas de forma controlada em um determinado sistema. A partir desta injeção, pode-se investigar o comportamento do sistema durante a presença de falhas, verificando o seu funcionamento, bem como mecanismos de tolerância a falhas implementados. Em comparação com a *modelagem analítica*, a injeção de falhas mostra-se mais adequada para a investigação de sistemas complexos [Arlat et al. 2003].

Considerando-se sistemas baseados em trocas de mensagens, 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 o sistema em questão 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 de mensagens e o colapso de nodos que formam a respectiva rede.

A proliferação de injetores de falhas para aplicações baseadas em troca de mensagens leva a uma dificuldade inerente ao uso dos mesmos. O principal problema está relacionado a *escolha* do injetor a ser utilizado para um dado experimento, visto que os mesmos possuem, em sua maior parte, abordagens distintas de funcionamento. Por este motivo, um esforço adicional é necessário nesta etapa, transcendendo assim a já complexa tarefa de elaboração de experimentos para testes de sistemas computacionais.

Neste contexto, o presente artigo aborda um modelo de ambiente para descrição de cenários detalhados de falhas, com foco em falhas de comunicação, conforme ilustrado na figura 1. O principal objetivo deste ambiente consiste em facilitar e expandir o uso dos diversos injetores de falhas existentes na literatura, aumentando assim a usabilidade dos mesmos. Algumas premissas para a realização deste artigo são abordadas nos itens a seguir.

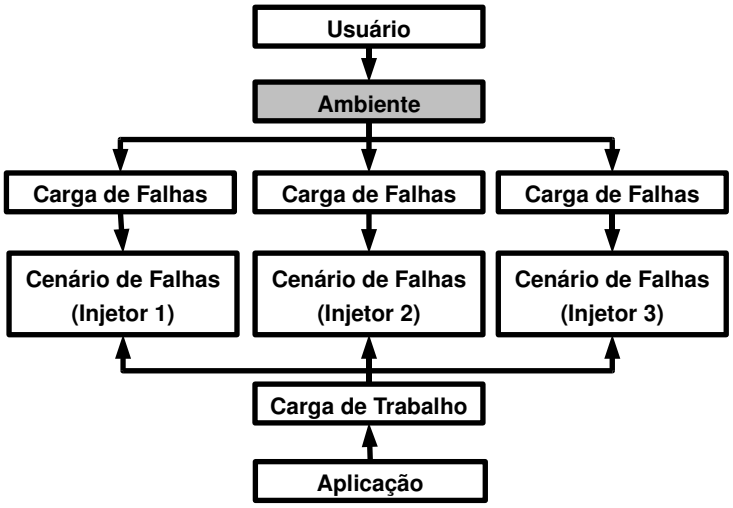

Figura 1. Ambiente para descrição de cargas de falhas.

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

adaptado para a construção de novos cenários de falhas, de acordo com a necessidade do projetista da aplicação.

A organização do artigo foi realizada da seguinte forma: a seção 2 apresenta uma visão geral sobre os injetores de falhas com foco em sistemas baseados em troca de mensagens, com ênfase aos injetores que mais se aproximam a este trabalho. A seção 3, por sua vez, descreve detalhadamente o modelo do ambiente, abrangendo seus elementos principais. Conclusões e detalhes sobre o andamento do trabalho são apresentados na seção 4.

2. Trabalhos Relacionados

Esta seção visa apresentar alguns dos principais injetores de falhas existentes na literatura. Primeiramente, serão descritos dois injetores desenvolvidos no Grupo de Tolerância a Falhas da UFRGS: FIONA e FIRMI, nas seções 2.1 e 2.2, respectivamente. Em seguida, serão abordados outros dois injetores conhecidos na literatura: DOCTOR (seção 2.3) e FAIL (seção 2.4).

2.1. FIONA

FIONA [Jacques-Silva et al. 2004] é um injetor de falhas com foco em sistemas distribuídos de *larga escala*. A abordagem utilizada no mesmo consiste na **instrumentação de código**, utilizando-se para isso da ferramenta JVMTI. O foco do injetor é o protocolo UDP, muito embora estejam previstas extensões para outros protocolos, tais como o TCP, por exemplo [Gerchman and Weber 2006].

A arquitetura *distribuída* de FIONA, assim como a local, também é constituída por três elementos: o **injetor principal** (responsável pelo gerenciamento do experimento), o **injetor de site** (que gerencia o conjunto de máquinas integrantes do respectivo site) e o **injetor local** (que aplicam as falhas propriamente ditas na aplicação alvo). A figura 2 ilustra esta arquitetura distribuída.

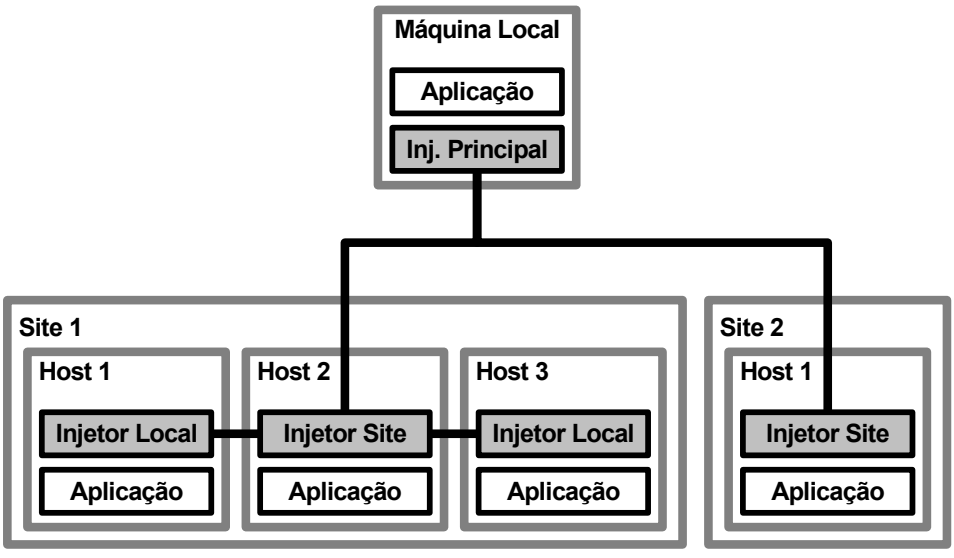

Figura 2. Arquitetura distribuída de FIONA [Jacques-Silva et al. 2004].

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

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

2.2. FIRMI

FIRMI [Vacaro and Weber 2006] é uma ferramenta de injeção de falhas que emula cenários de falhas envolvendo JavaRMI (Remote Method Invocation). JavaRMI, um sistema baseado em objetos distribuídos, possui diversos pontos suscetíveis a falhas, principalmente se considerarmos a rede de comunicação no qual o mesmo é executado. Assim, a ferramenta FIRMI é utilizada para avaliar as aplicações alvo que são construídas sobre JavaRMI, além de avaliar também os mecanismos de tolerância a falhas existentes nestas aplicações.

A arquitetura de FIRMI é definida através de um *diagrama de classes*, ilustrado na figura 3. Como é possível visualizar nesta figura, os tipos de falhas são implementados através de *exceções*. Estas exceções são ativadas na aplicação alvo através de uma técnica de *instrumentação*. Esta instrumentação de código, por sua vez, é realizada a partir do *stub* e do *skeleton*, ambos componentes do sistema de comunicação JavaRMI, referentes ao cliente (que solicita as requisições) e ao servidor (que atende as requisições) da aplicação alvo, respectivamente.

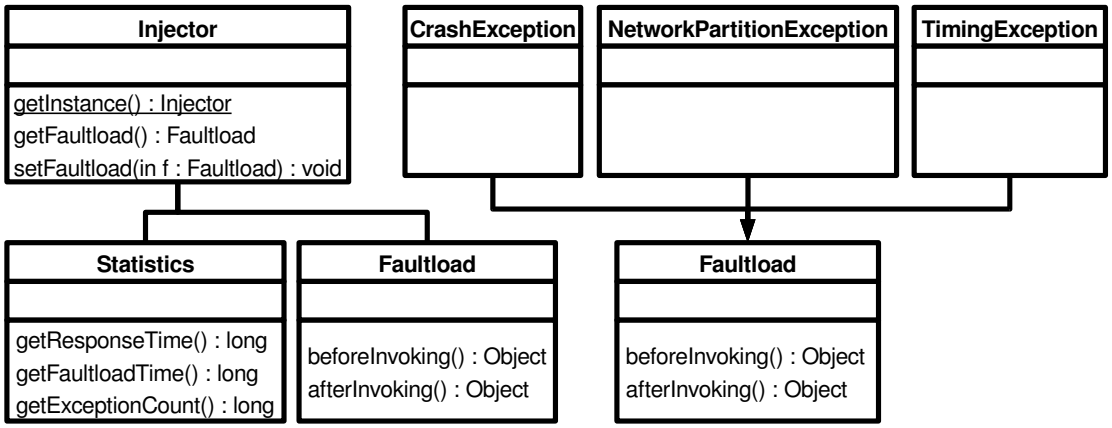

Figura 3. Diagrama de classes de FIRMI [Vacaro and Weber 2006].

Para a especificação de carga de falhas, FIRMI utiliza a classe `Faultload`. Assim, ao especificar uma carga, esta classe deve ser estendida, sendo a funcionalidade específica da mesma implementada através dos métodos `beforeInvoking()`

e `afterInvoking()`. Estes métodos são responsáveis pelas implementações das funcionalidades que ocorrem *antes* e *depois* da carga de falhas do injetor em si, respectivamente.

2.3. DOCTOR

A ferramenta DOCTOR [Han et al. 1995] consiste em um ambiente de injeção de falhas, com foco em sistemas distribuídos de tempo real. Seu objetivo inicial era injetar falhas na aplicação de tempo real HARTS [Shin 1991], aplicação em que DOCTOR foi utilizada de forma extensiva. Além do injetor em si, DOCTOR era formado por uma série de ferramentas auxiliares, tais como coletor de dados, gerador de cargas sintéticas de trabalho e interfaces gráficas (neste último, para interação com o usuário do injetor).

Referente à arquitetura, DOCTOR define uma série de componentes. Os principais componentes integrantes desta arquitetura são descritos nos parágrafos a seguir. A figura 4 ilustra a arquitetura esquematizada de DOCTOR.

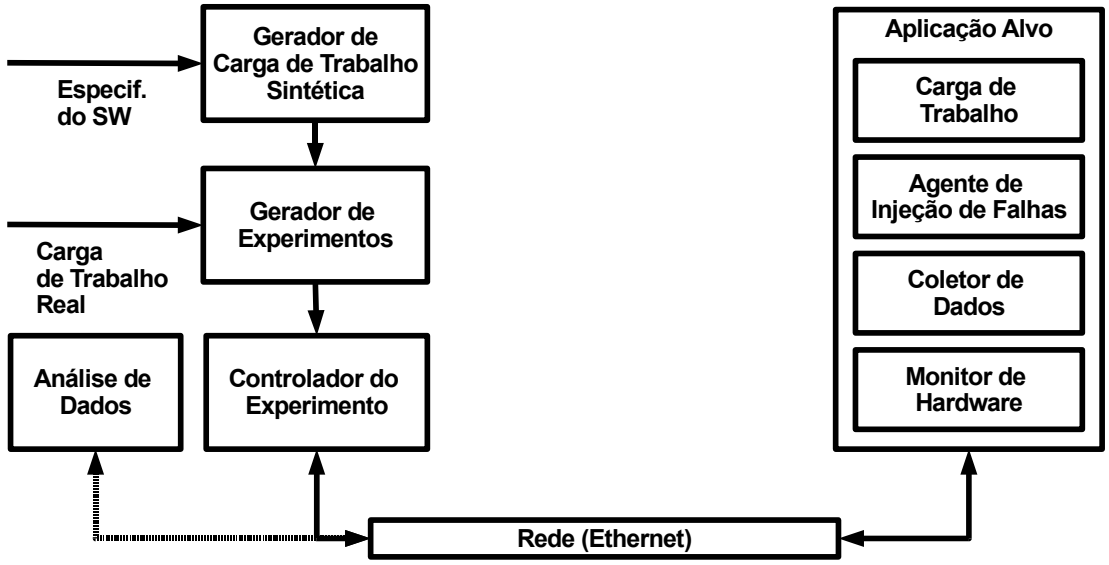

Figura 4. Arquitetura de DOCTOR [Han et al. 1995].

Primeiramente, o **Gerador de Experimentos** é o responsável pela obtenção da carga de trabalho da aplicação. Neste caso, podem ser usadas aplicações reais ou "artificiais" (obtidas a partir do **Gerador de Carga de Trabalho Sintética**). Outra função do Gerador de Experimentos é a de realizar a leitura do arquivo que descreve o experimento, que possui um formato próprio e contém informações sobre os tipos de falhas a serem consideradas, além do tempo em que as mesmas devem ser injetadas.

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

2.4. FAIL/FCI

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

A arquitetura da ferramenta, ilustrada na figura 5 como um exemplo de um nodo em um sistema distribuído, é composta por três componentes: um **compilador**, uma **biblioteca** e um **daemon**. Ambos são descritos nos itens a seguir.

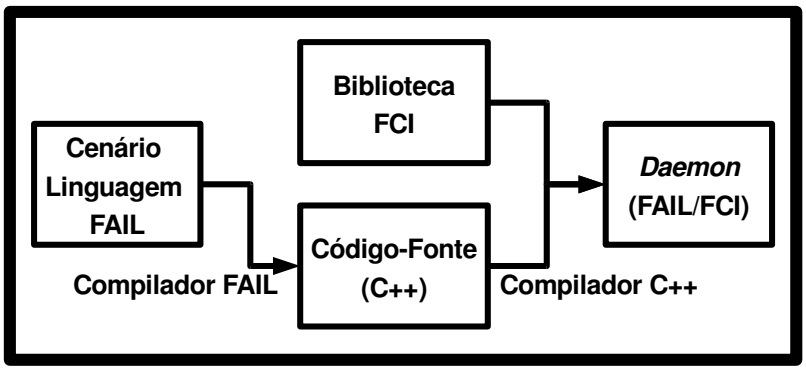

Figura 5. Arquitetura do injetor FCI [Tixeuil et al. 2006].

- **Compilador FCI:** responsável pela pré-compilação dos cenários de falhas escritos em FAIL, gerando códigos-fontes (no caso do FCI, em C++) que serão utilizados pela biblioteca FCI (descrita no próximo item), bem como arquivos de configuração para a realização do experimento.
- **Biblioteca FCI:** a partir dos arquivos gerados pelo compilador, a biblioteca FCI realiza a distribuição destes arquivos pelos nodos do sistema distribuído. Vale ressaltar que esta biblioteca distribui os arquivos como código-fonte *não compilado*, de forma a manter a *heterogeneidade* entre os nodos que formam o *cluster* do experimento.
- **Daemon FCI:** componente presente em cada nodo do *cluster*, o daemon é encarregado de realizar as compilações dos fontes em cada nodo, bem como realizar a instrumentação de código, que trata-se da injeção de falhas em si na aplicação alvo.

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

limitação é relacionada ao modelo de falhas implementado pela ferramenta, envolvendo apenas falhas relacionadas a colapso.

3. Modelo do Ambiente

Um ambiente de cenários de falhas deve refletir todos os casos possíveis de falhas, de acordo com o tipo de aplicação alvo desejado. Neste sentido, o modelo descrito neste artigo foi elaborado visando a divisão em elementos, modularizando as funcionalidades do ambiente, bem como promovendo a extensibilidade do mesmo. A figura 6 ilustra uma visão macro da arquitetura do ambiente, sendo cada elemento descrito nos itens a seguir.

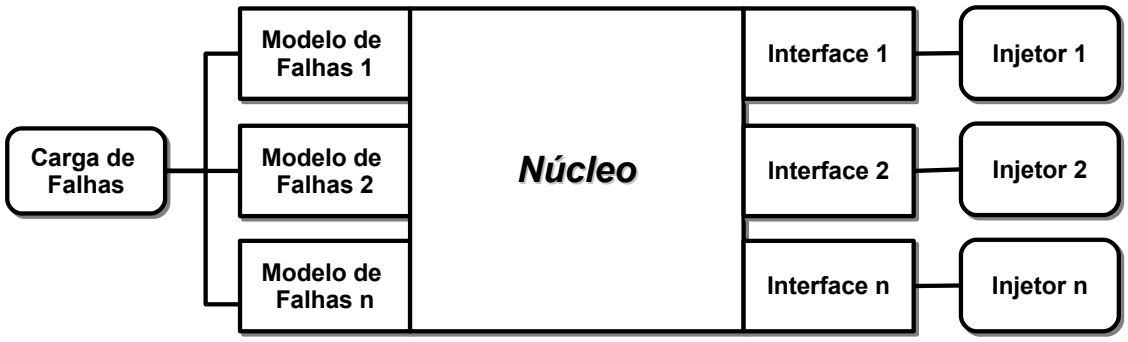

Figura 6. Arquitetura do Ambiente

- **Carga de Falhas:** representando uma *entrada de dados* ao ambiente pelo usuário, 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. É análoga às cargas de falhas realizadas em injetores específicos, com a vantagem de ser genérica, ou seja, não ser restrita às peculiaridades de uma determinada ferramenta.
- **Modelo de Falhas:** define um conjunto de falhas admitidas por uma determinada aplicação. Assim, são especificados os tipos de falhas que estarão disponíveis ao usuário em seu experimento. Os tipos definidos neste modelo delimitarão a funcionalidade da carga de falhas. Assim como a carga de falhas, esta também é uma entrada de dados definida pelo usuário, sendo que o ambiente fornece uma forma *intuitiva* para que o usuário especifique modelos de falhas com facilidade.
- **Núcleo:** sendo o elemento de mais baixo nível e, conseqüentemente, o mais importante do ambiente, o *núcleo* define uma série de *falhas primitivas* existentes em sistemas baseados em troca de mensagens. Logo, o principal objetivo consiste em mapear a carga de falhas, definida pelo usuário, às falhas primitivas existentes, iniciando-se assim a transformação da carga de falhas genérica para a carga de falhas específica de cada injetor utilizado.
- **Interface:** elemento externo ao núcleo, 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 define a interface como uma outra entrada de dados definida pelo usuário e, assim como o modelo de falhas, é fornecida uma forma *intuitiva* para que o usuário especifique a *forma* pelo qual o seu injetor específico trabalha.

- **Injetor:** finalmente, o elemento *injetor* representa um injetor de falhas específico que o usuário deseja utilizar no ambiente. O ambiente permite a utilização de vários injetores simultaneamente, sendo que os mesmos recebem como entrada suas respectivas cargas de falhas específicas, geradas pelas interfaces descritas no item anterior.

Desta forma, pode-se observar que o ambiente possui três entradas de dados, correspondentes a *Carga de Falhas*, ao *Modelo de Falhas* e à *Interface*. Referente a estes dois últimos, percebe-se que os mesmos são integrados ao *núcleo* do ambiente, sendo descritos pelo usuário utilizando a abordagem de *plugins*. Assim, o ambiente torna-se modularizado e extensível, facilitando ainda mais o seu uso.

Nas subseções seguintes, estes três elementos, juntamente com o núcleo, serão descritos em detalhes, abrangendo o funcionamento interno de cada um deles. Para esta descrição, será utilizada uma abordagem *bottom-up*. Assim, será descrito primeiramente o *núcleo*, por ser o elemento de mais baixo nível. Logo após, será abordado o *Modelo de Falhas*, a *Interface para Injetores Específicos* e, finalmente, a descrição da *Carga de Falhas*.

3.1. Núcleo

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

O foco do ambiente, conforme descrito na seção 1, está voltado a falhas de comunicação. Sistemas baseados em troca de mensagens, por terem a sua execução realizada em uma rede de computadores, são compostos basicamente por três componentes fundamentais, a saber: **nodos**, **caminhos** e **pacotes**. Estes componentes, conceituados nos itens que seguem, serão utilizados no núcleo do ambiente para a realização efetiva da injeção de falhas. Uma ilustração dos mesmos, de forma simplificada, é mostrada na figura 7.

- **Nodos:** formam as *unidades de execução* do sistema em questão, possuindo estado interno próprio. Assim, dependendo do contexto, um nodo pode representar um **emissor**, um **receptor** ou **ambos**.
- **Caminhos:** representam as *unidades de comunicação* entre dois nodos quaisquer do sistema. Desta forma, a troca de dados entre os nodos do sistema só pode ser realizada através de caminhos. Além disso, caminhos possuem uma velocidade de propagação específica, que indica o tempo no qual um determinado dado leva para ser transportado de um nodo do sistema para outro.
- **Pacotes:** os pacotes correspondem às *unidades de dados* existentes no sistema. Portanto, quando um determinado nodo do sistema deseja transmitir ou receber conteúdos de algum outro nodo, estes conteúdos são armazenados em um ou mais pacotes, para assim serem transportados a partir do caminho existente. Logo, um determinado pacote, durante a execução do sistema, pode estar no contexto do

nodo, do caminho ou em ambos (neste caso, representando um pacote *em envio* ou *em recebimento*), como ilustrado na figura 7.

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

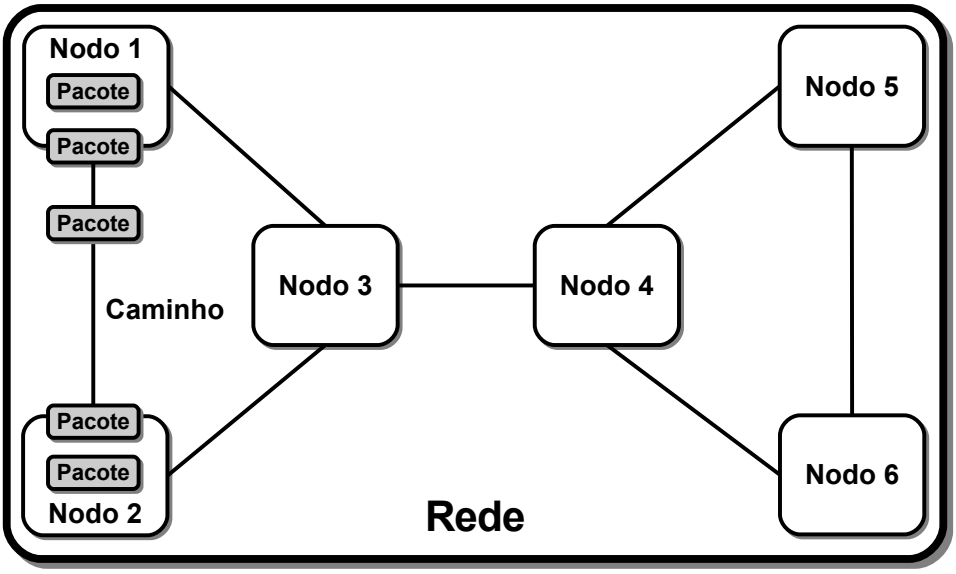

Figura 7. Componentes fundamentais de sistemas baseados em troca de mensagens

Além disso, outra abordagem adotada pelo núcleo do ambiente referem-se às *falhas primitivas* suportadas pelo ambiente. Por falha primitiva, entende-se um tipo de falha comumente encontrada em sistemas baseados em troca de mensagens, podendo levar o respectivo sistema a um estado *inconsistente*. Desta forma, o componente *Modelo de Falhas* (que será descrito na subseção 3.2) poderá fazer uso destas falhas primitivas, deixando-se assim o ambiente *independente* de um modelo de falhas específico.

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

- **Reinício:** o componente em questão é *reiniciado*, tendo assim sua configuração retornada ao seu respectivo estado inicial. Nos contextos de nodo e caminho, isto significa que todos os pacotes serão descartados, sem possibilidade de recuperação dos mesmos.
- **Parada:** o componente tem a sua execução *suspensa*, podendo ser *retomada* em algum momento posterior do experimento. Em nodos, esta falha primitiva corresponde a parada no envio e recebimento de pacotes, enquanto que em caminhos, indica a parada na propagação de pacotes. Ao retomar a execução, os pacotes que

estavam presentes nos respectivos componentes **não** são perdidos, seguindo-se assim os destinos já definidos anteriormente para os mesmos.

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

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

Referente à arquitetura do ambiente, a mesma é definida através de uma *hierarquia de classes*. Esta técnica é adequada para a construção do núcleo, uma vez que a mesma permite delimitar, a partir de um componente genérico, todas as funcionalidades necessárias - neste caso, as falhas primitivas. Desta forma, fica a cargo de componentes específico a implementação destas funcionalidades, que não podem fugir ao escopo definido no componente genérico.

No caso do ambiente, é definida uma interface **Component**, que define os métodos correspondentes às falhas primitivas definidas nos itens anteriores, a saber: **restart()**, **stop()**, **dropPackets()** e **delayPackets()**, respectivamente. No contexto do núcleo, a classe **Network** engloba um conjunto de objetos da classe **Component**. Além disso, temos as classes específicas **Node** e **Path**, que implementam os métodos de **Component**, além da classe **Packet**, abrangendo assim todos os componentes fundamentais abordados. O diagrama UML desta estrutura de classes pode ser visualizado na figura 8.

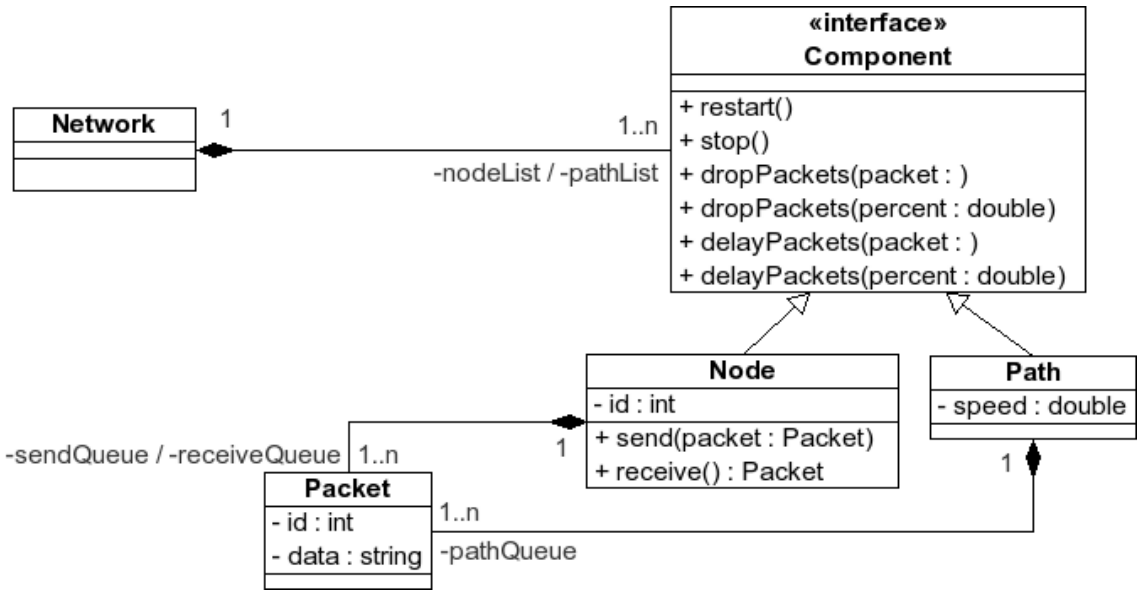

Figura 8. Diagrama UML do núcleo do ambiente

3.2. Modelo de Falhas

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

Considerando sistemas baseados em troca de mensagens, existem diversos modelos de falhas disponíveis na literatura, dentre os quais podem ser destacados os modelos definidos por Cristian [Cristian et al. 1986] e Birman [Birman 1996]. Cristian define as falhas de colapso, omissão, temporização e bizantinas. Birman, por sua vez, define as falhas de colapso, parada segura (*fail-stop*), omissão de envio, omissão de recepção, rede, particionamento de rede, temporização e bizantinas.

No contexto do ambiente, é utilizada uma abordagem simplificada para a especificação de um modelo de falhas, baseada em *script*. Esta abordagem é ilustrada na figura 9 e explicada nos parágrafos subseqüentes.

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

Figura 9. Abordagem *script* para especificação de modelos de falhas

A linguagem *script* utilizada, conforme figura 9, é formada por três construções. A construção <Component> representa o **componente** para o qual se deseja especificar o tipo de falha, podendo ser Node ou Path. <Fault Label> indica um **rótulo** para o tipo de falha, sendo diretamente especificado pelo usuário do script. Finalmente, <Primitive Fault> especifica uma falha primitiva a ser adicionada no tipo de falha a ser especificado, podendo ser um dos métodos pertencentes a classe **Component**, vista na subseção 3.1.

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

Figura 10. Exemplo para especificação de modelos de falhas

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

e “OU” (execução de apenas *um* das falhas primitivas relacionadas no escopo, escolhido de forma aleatória), respectivamente.

Desta forma, mesmo sendo simplificada, a linguagem criada permite a especificação de modelos elaborados de falhas, refletindo de maneira adequada os conceitos indicados nos mesmos. Para efeito de exemplo, a figura 10 ilustra um exemplo de modelo de falhas especificado a partir desta linguagem - neste caso, é ilustrada a especificação do modelo proposto por Cristian.

3.3. Interface para Injetores Específicos

Este elemento visa mapear um cenário de falhas criado no ambiente no formato de um determinado injetor. Neste formato, o cenário é chamado *cenário específico*, correspondendo assim à carga efetiva de falhas ao injetor específico, servindo como dados de entrada ao mesmo. Desta forma, objetiva-se oferecer a compatibilidade necessária aos injetores de falhas existentes.

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

Figura 11. Script de mapeamento - modelo de falhas/especificação do injetor

Assim como o modelo de falhas, descrito na subseção 3.2, este elemento é descrito pelo usuário seguindo-se a abordagem de *plugin*. Em outras palavras, o usuário do ambiente especifica a *forma* pelo qual a especificação do injetor específico é **mapeada** ao modelo de falhas já definido no ambiente. Para isso, é utilizado um *script de mapeamento*, cuja sintaxe pode ser visualizada na figura 11.

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

```
Node.Crash : UdpCrashFault
Node.Omission : UdpOmissionFault
```

(a) FIONA

```
Node.Crash : halt
Node.Omission : stop, continue
```

(b) FAIL/FCI

Figura 12. Exemplos de scripts de mapeamento

Vale ressaltar que o ambiente permite a inclusão de vários injetores de falhas simultaneamente a uma execução de um dado experimento. Por este motivo, são ilustrados dois exemplos de scripts de mapeamento na figura 12. A figura 12(a) representa um mapeamento para o injetor FIONA [Jacques-Silva et al. 2004] (um injetor de falhas para aplicações de rede), enquanto que a figura 12(b) ilustra um mapeamento para o injetor

FAIL/FCI [Hoarau and Tixeuil 2005] (um injetor para aplicações do tipo *grid*). Em ambos os casos, o modelo de falhas utilizado refere-se ao proposto por **Cristian** e, logo, são utilizados aqui os rótulos definidos na figura 10.

3.4. Carga de Falhas

A partir da carga de falhas, o usuário do ambiente poderá especificar exatamente quais as falhas estarão presentes em seu experimento, bem como as configurações que cada uma delas possuirá. Entre os elementos descritos, este é considerado o de mais alto nível, uma vez que o mesmo faz uso direto do modelo de falhas definido no ambiente.

Seguindo-se a abordagem adotada nos demais elementos do ambiente, a carga de falhas é realizada a partir de um *script* simplificado. A sintaxe deste script busca a *generalidade*, de forma a permitir uma especificação adequada de cenários de falhas, no contexto de falhas de comunicação. A figura 13 ilustra a estrutura utilizada neste script.

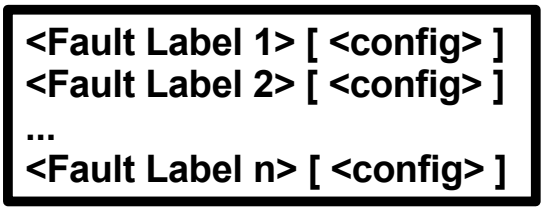

Figura 13. Estrutura utilizada no script para carga de falhas

De acordo com esta estrutura, pode-se visualizar que a mesma permite a definição de uma *seqüência* de falhas (sendo uma ocorrência por linha), caso seja necessário. Vale ressaltar que o usuário do ambiente não precisa, necessariamente, utilizar todos os tipos (estrutura "<FaultLabel>") definidos no modelo de falhas em um experimento, bem como os comandos de configuração (estrutura "<config>", que depende do tipo de falha utilizada). Entretanto, pelo menos um tipo deve ser utilizado, de forma que a injeção de falhas, no experimento, possa ocorrer de forma efetiva.

Para exemplificar a utilização deste script, é exibida na figura 14 uma carga de falhas genérica. Nesta carga, ocorre uma seqüência de duas falhas. Na primeira, ocorre um colapso de nodo, sem a necessidade de configurações adicionais. Já na segunda falha, ocorre uma falha de omissão em um caminho, com uma configuração adicional representada como "0.1", indicando uma probabilidade uniforme de perda em 10% dos pacotes. Em ambos os casos, é assumido o modelo de **Cristian**, conforme figura 10.

Figura 14. Exemplo de carga de falhas genérica

4. Conclusões

Este artigo apresentou um modelo de ambiente para descrição de cenários detalhados de falhas, com ênfase em falhas de comunicação. Foram abordados os elementos principais

deste ambiente, com uma descrição detalhada de cada um deles. Além disso, nos elementos de mais alto nível (ou seja, onde ocorre interação direta com o usuário do ambiente), foram ilustrados exemplos de caso de utilização, envolvendo alguns injetores de falhas existentes na literatura.

A partir do modelo do ambiente, foi constatado que o mesmo apresenta uma arquitetura adequada para a realidade de cenários de falhas, com pontos de extensibilidade bem definidos. Vale ressaltar também a facilidade de uso do ambiente, a partir de linguagens *script* simplificadas. Com isso, o uso do ambiente torna-se adequado para experimentos que envolvam sistemas baseados em troca de mensagens, independente da complexidade dos mesmos.

Referências

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

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

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

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

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

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

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.

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

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

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