Construção de uma Ferramenta de Injeção de Falhas Simuladas para Avaliação de Sistemas Distribuídos

Ruthiano S. Munaretti, Marinho P. Barcellos

[1]PIPCA - Programa Interdisciplinar de Pós-Graduação em Computação Aplicada
Unisinos - Universidade do Vale do Rio dos Sinos
Av. Unisinos, 950 - São Leopoldo, RS - 93022-000

ruthiano@gmail.com, marinho@acm.org

Abstract. *The union of simulation with fault injection allows the evaluation of a complex system when it is subjected to faults, allowing the use of an abstract model of this system. Simmcast is a simulation framework with great extensibility, and SimmFI is an extension that aims to provide support for fault injection in the simulated distributed system. This paper discusses architectural and implementation aspects of SimmFI, with emphasis on the fault injection mechanisms as well as activation and deactivation of faults. A test set was run to evaluate the proper behaviour of the implementation.*

Resumo. *A união da simulação com a injeção de falhas possibilita a avaliação de um sistema complexo quando o mesmo é sujeito a falhas, permitindo a utilização de um modelo abstrato do sistema em questão. O* Simmcast *é um framework de simulação com grande extensibilidade, e o* SimmFI *é uma extensão com o objetivo de oferecer suporte à injeção de falhas no sistema distribuído simulado. O presente artigo discute aspectos da arquitetura e implementação do SimmFI, enfatizando os mecanismos de injeção de falhas e de ativação e desativação de falhas. Um conjunto de testes foi realizado com o objetivo de avaliar o correto funcionamento da implementação.*

1. Introdução

A avaliação de um sistema distribuído consiste basicamente na verificação de uma ou mais propriedades do respectivo sistema, a fim de constatar a *satisfabilidade* destas propriedades no cenário proposto. A importância desta avaliação decorre da própria natureza distribuída deste tipo de sistema, formada a partir de uma série de características não determinísticas, tais como nodos heterogêneos e interconexões de velocidades variadas. Para estes casos, a *simulação* apresenta um ambiente determinístico e controlado, o que torna esta técnica bastante adequada em sistemas complexos, pela possibilidade de utilização de um protótipo para esta avaliação.

Devido aos serviços oferecidos, um sistema distribuído pode necessitar adicionalmente de características como confiabilidade e disponibilidade. Nas aplicações tolerantes a falhas, o serviço fornecido deve possuir uma certa garantia de funcionamento, mensurada através da noção de *dependabilidade* (*dependability*) [A. Avizienis et al. 2001]. Neste contexto, a *injeção de falhas* é uma importante metodologia de avaliação da dependabilidade, que permite um melhor entendimento do comportamento do sistema perante

a ocorrência de falhas. Ela é complementar à *modelagem analítica*, e não possui a desvantagem de se tornar mais restritiva a medida que a complexidade do sistema aumenta [Arlat et al. 2003]. Atualmente, a injeção de falhas é uma técnica amplamente utilizada na avaliação de sistemas computacionais, onde podem ser destacadas as aplicações espaciais [Ambrosio 2005], bem como as aplicações baseadas em *clusters* [Nagaraja et al. 2003].

Neste raciocínio, a união da simulação com a injeção de falhas possibilita a avaliação de um sistema complexo durante a presença de falhas, permitindo utilizar, para este propósito, um modelo abstrato do sistema em questão. Com relação à simulação, o framework Simmcast [H. H. Muhammad and M. P. Barcellos 2001] apresenta-se como uma importante ferramenta, cuja extensibilidade já foi demonstrada [M. P. Barcellos et al. 2004]. Na injeção de falhas, por sua vez, o SimmFI (*Simmcast with Fault Injection*) objetiva oferecer o suporte a falhas no framework de simulação existente.

Entre as abordagens similares, pode-se citar como mais relevantes o Neko [Urbán et al. 2001] e o Network Simulator (NS-2) [ns 2006]. Tal como a presente proposta, o Neko é um framework de simulação para estudo de algoritmos distribuídos. No entanto, o Neko é limitado a medida que não possui suporte a uma gama de falhas diferentes (apenas omissão de mensagens), nem possibilita a criação de cenários complexos de teste. Já o NS-2 é um simulador de redes cujo uso é amplamente disseminado. Em [R. de M. Trindade et al. 2002], foram propostas alterações no NS-2 para que o mesmo pudesse ser usado na avaliação de sistemas distribuídos em cenários com falhas. Naquele estudo, detectou-se que o projeto do NS-2 não se adapta bem à modelagem e simulação de sistemas distribuídos.

O projeto conceitual do SimmFI foi apresentado em [M. P. Barcellos et al. 2005], com foco na interface e comportamento esperado para os mecanismos fundamentais de **ativação de falhas** e o **comportamento de falhas**. O presente artigo trata de questões arquiteturais e de implementação do SimmFI. A ferramenta resultante oferece lições de projeto e uma experiência que espera-se possam contribuir para o projeto de outros artefatos de software similares, bem como servir a outros esforços de pesquisa na área.

O artigo está organizado da seguinte maneira. A Seção 2 apresenta uma versão mais detalhada da arquitetura do Simmcast, em função de seus componentes, e contendo o estritamente necessário para o entendimento do SimmFI. A Seção 3 mostra uma visão geral do SimmFI, com ênfase nas decisões de projeto tomadas. As Seções 4 e 5 abordam o desenvolvimento dos mecanismos de ativação/desativação de falhas e atuação de falhas, respectivamente. A Seção 6, por sua vez, ilustra um conjunto de testes realizados com a ferramenta e os resultados obtidos. Finalmente, a Seção 7 tece conclusões sobre o trabalho realizado e indica passos futuros.

2. Arquitetura

A arquitetura do Simmcast, conforme [M. P. Barcellos et al. 2004], pode ser representada em camadas. Esta representação permite dividir os componentes por níveis de complexidade, onde os componentes de níveis inferiores oferecem serviços para os componentes de níveis superiores. As camadas que formam a arquitetura do Simmcast são ilustradas[1] na Figura 1.

[1] A arquitetura em camadas descrita a seguir representa uma versão aperfeiçoada e substancialmente detalhada em relação à [M. P. Barcellos et al. 2004].

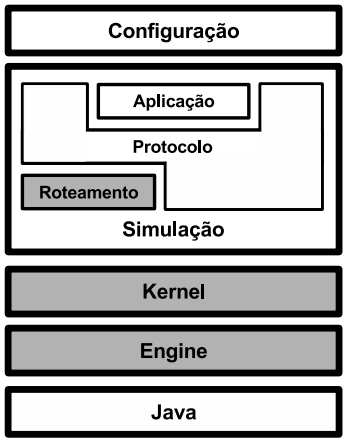
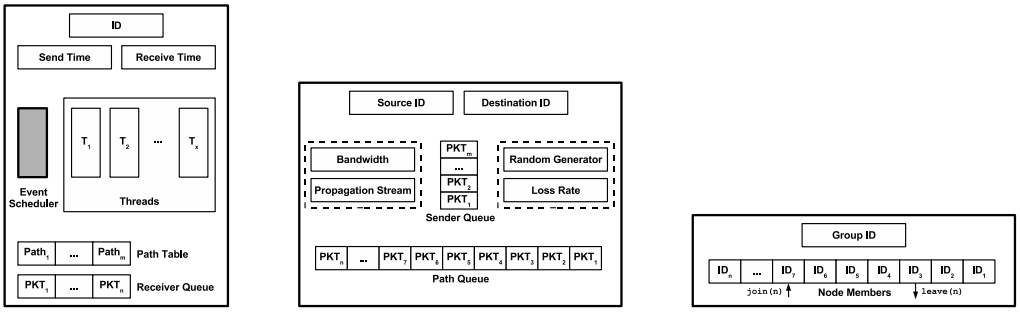


Figura 1. Camadas que formam a arquitetura do Simmcast.

A camada **Java**, representada pela Máquina Virtual Java, oferece o suporte a *threads* (através da classe java.lang.Thread). No Simmcast, a thread é a unidade básica de execução. Na camada superior, esta classe é especializada, para fins de adequação ao modelo de escalonamento proposto.

A camada **Engine** representa o motor de simulação, implementado como uma máquina de eventos discretos baseada em processos. O processo, principal componente desta camada, corresponde a uma especialização da thread Java, retratada na camada anterior. Esta especialização é necessária para a utilização de um modelo de escalonamento próprio, que será visto mais adiante.

A próxima camada é a correspondente ao ***kernel*** do simulador, onde são definidos os componentes fundamentais para simulação de protocolos: Nodo, Caminho e Grupo. Estes componentes são ilustrados na Figura 2 e descritos a seguir.



(a) Nodo. (b) Caminho. (c) Grupo.

Figura 2. Estrutura interna dos componentes Nodo, Caminho e Grupo.

- **Nodo:** principal componente da camada, com objetivo dependente do protocolo a ser simulado: pode representar um *computador* (no caso do sistema físico ser representado por uma rede de computadores) ou um *roteador* (se o grupo Roteamento, descrito logo a seguir, estiver sendo utilizado). Sua estrutura interna, ilustrada na Figura 2(a), abriga uma ou mais threads de execução, bem como uma

thread responsável pelo escalonamento de eventos, uma fila de recebimento, uma lista de caminhos de destino e variáveis numéricas, representando o identificador único (ID) e os tempos de envio e recebimento de um pacote pelo respectivo nodo.

- **Caminho:** canal de comunicação unidirecional entre dois nodos quaisquer, este componente é formado por uma fila de pacotes em propagação e por uma fila de envio, além de variáveis representando identificadores dos nodos origem e destino, largura de banda e taxa de perda, ilustrados na Figura 2(b). Sobre a fila de envio, vale ressaltar que a mesma está localizada conceitualmente no Nodo, mas está implementada no Caminho por uma questão de projeto, a fim de simplificar a implementação do envio de pacotes por um determinado Nodo.

- **Grupo:** componente formado por um conjunto de Nodos, que são diretamente conectados através de Caminhos. A estrutura interna, ilustrada na Figura 2(c), abriga os identificadores do próprio Grupo e dos Nodos que fazem parte do mesmo, além das primitivas join(n) e leave(n), responsáveis por incluir e excluir um determinado Nodo n do Grupo em questão.

Já a camada **Simulação** é a responsável pela montagem de um ambiente de simulação. Este ambiente é elaborado através da extensão dos componentes definidos no kernel, adicionando-se para isso as funcionalidades desejadas pelo usuário. Esta camada é dividida em três grupos, descritos a seguir.

- **Protocolo:** representa a lógica do protocolo a ser simulado. Este é um grupo de presença obrigatória, uma vez que é no protocolo que se concentra a principal atividade de simulação no Simmcast.

- **Roteamento:** grupo opcional, utilizado nas ocasiões em que o protocolo a ser simulado necessita de componentes pertencentes à estrutura física da rede (tais como roteadores e comutadores, por exemplo).

- **Aplicação:** grupo necessário nas situações em que se deseja verificar o comportamento de uma determinada aplicação perante o experimento de um determinado protocolo, de forma separada do mesmo. Assim como o roteamento, o uso deste grupo também é opcional.

Finalmente, a camada **Configuração** corresponde a um mecanismo para criação de um cenário de simulação, baseado no ambiente definido e criado na camada anterior. Este cenário é formado basicamente da topologia em que o protocolo será simulado (nodos, grupos e suas respectivas conexões), bem como de informações adicionais relevantes ao experimento (tais como tempo de processamento de nodos e taxa de perda de um caminho). Este mecanismo é implementado com carga dinâmica de classes, através de reflexão computacional.

Por se tratar de um framework, o Simmcast possui os pontos de extensão definidos nas camadas Kernel e Roteamento. Como é possível visualizar na Figura 1, as camadas destacadas em cinza são aquelas implementadas pelo Simmcast. As demais camadas, com exceção da camada Java, são implementadas pelo usuário do simulador, através da adição de funcionalidades específicas ao problema em questão, bem como da criação do ambiente e cenário de simulação desejados.

3. SimmFI

A seção anterior apresentou uma visão *conceitual* da arquitetura do Simmcast, abrangendo o conteúdo de cada camada, bem como a interação entre elas. As camadas do SimmFI, segundo [M. P. Barcellos et al. 2005], podem ser organizadas em duas partes distintas:

- **Parte Superior:** representa o sistema alvo a ser simulado (como um protocolo distribuído, por exemplo). Esta parte envolve as camadas de Configuração e Simulação.

- **Parte Inferior:** representa o sistema físico no qual o sistema alvo é executado (computadores e sua respectiva rede de comunicação, por exemplo). Esta parte envolve as camadas Kernel e Engine.

No contexto do SimmFI, são abordadas somente as camadas **Kernel** e **Engine**, uma vez que a injeção de falhas é aplicada na Parte Inferior, a fim de monitorar o comportamento da Parte Superior sob condições de falhas. São apresentadas nas seções seguintes as principais características existentes no projeto conceitual do SimmFI. A Seção 3.1 aborda a injeção de falhas nos componentes fundamentais, enquanto a Seção 3.2 descreve o mecanismo projetado para especificação de comportamentos de falha.

3.1. Injeção de Falhas

Conforme a Seção 2, os componentes fundamentais do Simmcast são Nodos, Caminhos e Grupos. Logo, estes são também os componentes sujeitos a falhas no simulador, devido à importância dos mesmos no decorrer de um experimento. No caso de um Nodo, a ocorrência de uma falha reflete nos serviços oferecidos pelo mesmo, como o envio de pacotes. Já em um Caminho, as falhas afetam os pacotes transportados. Finalmente, no caso do Grupo, a falha compromete todos os pacotes enviados ao mesmo.

Assim, cada um destes componentes admite um determinado conjunto de falhas, conforme sua especificação, em termos de funcionalidade e de premissas sobre o ambiente. Este conjunto, denominado como **modelo de falhas**, define categorias de falhas (tal como [Hadzilacos and Toueg 1998, Jalote 1998]). Desta forma, a especificação de um sistema tolerante a falhas pode então basear-se na nomenclatura apresentada por este modelo, visando definir precisamente quais condições são tratadas, o que assume-se não ocorrer, e o que não é tratado. Em [M. P. Barcellos et al. 2005], foi definido um modelo de falhas, derivado do modelo proposto por Veríssimo e Rodrigues em [P. Veríssimo and L. Rodrigues 2001]. Este modelo, ilustrado na tabela 1, foi escolhido por possuir um nível adequado de abstração e ser orientado a sistemas distribuídos, um dos focos do framework de simulação.

Além do comportamento, outro quesito importante na aplicação de uma falha a um componente é o *escopo* do respectivo componente. Em outras palavras, dado um Pacote p, enviado por um Nodo n a um Caminho c, é preciso definir exatamente o momento em que p encontra-se em n, em p ou em ambos, de forma a afetar ou não p perante a existência de uma falha em n ou c. Neste caso, o pacote p pode estar em um de sete estados possíveis, ilustrados na Figura 3 e descritos a seguir.

- U1: espaço de aplicação no Nodo origem.

Tabela 1. Modelo de falhas.

Tipo de Falha	Descrição
Colapso	Componente pára silenciosamente de funcionar.
Omissão	Componente omite resultados, de forma completa ou parcial.
Temporização	Funcionamento com tempo arbitrário.
Sintática	Comportamento incorreto, detectável.
Semântica	Comportamento correto com sentido incorreto.

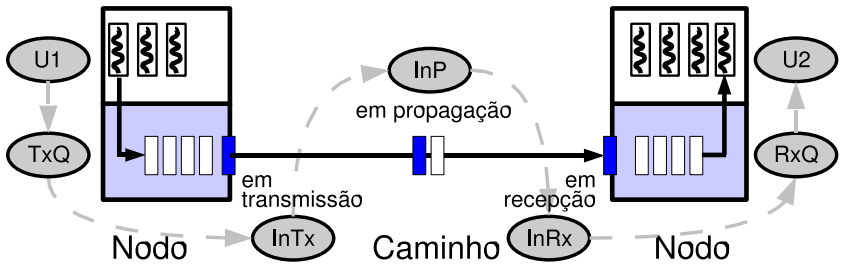


Figura 3. Diagrama de estados de um pacote.

- TxQ: fila de envio, no Nodo origem.
- InTx: pacote em transmissão, entre o Nodo origem e o Caminho.
- InP: pacote em propagação, totalmente no caminho.
- InRx: pacote em recebimento, entre o Caminho e o Nodo destino.
- RxQ: fila de recebimento, no Nodo destino.
- U2: espaço de aplicação no Nodo destino.

Desta forma, tem-se um conjunto S = {U1, TxQ, InTx, InP, InRx, RxQ, U2}, correspondente aos estados possíveis de um pacote, um conjunto C = {Nodo, Caminho, Grupo}, correspondente aos componentes sujeitos a falhas, e um conjunto B = {0, 1}, correspondente a valores booleanos. A partir destes conjuntos, obtém-se uma **Matriz de Contato** M, dada por M(c,s) = b e ilustrada na tabela 2, onde c pertence ao conjunto C, s pertence ao conjunto S e b pertence ao conjunto B. Nesta matriz, é representado o momento exato em que um determinado pacote está *em contato* com um determinado componente. Vale ressaltar que, nesta matriz, as trocas de estado ocorrem da esquerda para a direita.

Tabela 2. Matriz de Contato M, c x s.

	U1	TxQ	InTx	InP	InRx	RxQ	U2
Nodo	0	1	1	0	1	1	0
Caminho	0	0	1	1	1	0	0
Grupo	0	1	1	1	1	1	0

Visto o efeito imediato de uma falha em um componente, há casos em que uma determinada falha pode afetar *continuamente* o respectivo componente, comprometendo assim os demais pacotes que possam vir a existir no mesmo. Para este caso, é utilizado o mecanismo de **filtro**, responsável por *atuar* a falha no componente. A instalação deste

filtro varia de acordo com o componente. A Figura 4 ilustra a aplicação de uma falha em um caminho, através da instalação de um filtro entre os estados TxQ e InTx, impedindo assim que pacotes trafeguem no dado caminho.

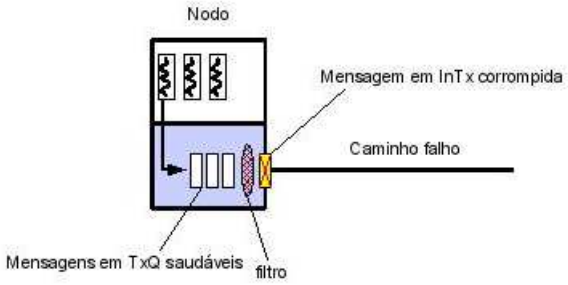


Figura 4. Atuação de uma falha em um Caminho.

3.2. Especificação de Comportamentos de Falha

A partir do conjunto de falhas possíveis em um experimento, é necessário um mecanismo para especificar *quando*, *como* e/ou *de que forma* estas falhas irão ocorrer. Para isso, é definido um mecanismo de **ativação/desativação de falhas**, revisado nos parágrafos seguintes e cuja implementação é detalhada na Seção 4.

Este mecanismo, como o próprio nome indica, possibilita duas modalidades de regras: a *regra de ativação* e a *regra de desativação*, utilizadas para **ativar** e **desativar** uma falha, respectivamente. Estas regras podem ser de vários *tipos*, sendo a escolha desse tipo inerente ao cenário de simulação desejado. Os tipos, por sua vez, podem ser divididos em duas grandes classes: *intervalos* e *expressões booleanas*.

Correspondendo a valores numéricos, os **intervalos** representam o tempo relativo em que, a partir de uma recuperação, uma determinada falha em um componente deve ser ativada. Além disso, este valor pode representar também o tempo relativo em que, a partir de uma falha, um determinado componente deve ser recuperado.

Já as **expressões booleanas** consistem de expressões que, no momento em que forem verdadeiras, ativam ou desativam uma determinada falha. Expressões booleanas são compostas de **termos**. Estes termos, por sua vez, são separados por *operadores binários*, que podem ser **lógicos** (“&&” e “||”) ou **relacionais** (“==”, “>”, “<”, “>=”, “<=” e “! =”). No contexto do mecanismo definido, um termo pode ser um dos seguintes ítens: *referência ao relógio de simulação*, *conteúdo de pacotes*, *variáveis de distribuição aleatória* ou *estado interno de nodos*.

4. Ativação/Desativação de Falhas

O mecanismo de ativação/desativação de falhas consiste em um dos componentes mais importantes do framework de simulação estendido, uma vez que o mesmo permite a criação dos mais variados cenários dentro de uma simulação de falhas. Por este motivo, a seção atual descreve este mecanismo no contexto da ferramenta proposta, o que é melhor abordado através de um *pipeline* de execução. Este pipeline, ilustrado na Figura 5, é descrito nos parágrafos seguintes.

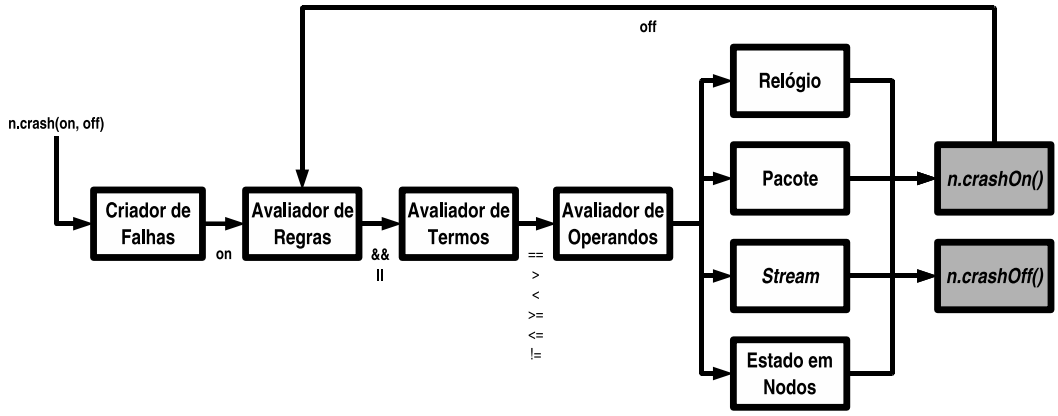


Figura 5. *Pipeline* de execução da Ativação/Desativação de falhas.

Ao invocar o método da falha (como n.crash(on,off)), o tipo da falha é remetido ao **Criador de Falhas**, juntamente com as regras on e off. A regra on é submetida para o *Avaliador de Regras*, enquanto a regra off é armazenada para posterior análise.

No **Avaliador de Regras**, a regra on é dividida a partir dos *operadores lógicos* ("&&" e "||"), caso existam. Para isso, a regra é convertida para a notação *pós-fixada*, devido a facilidade da mesma em manusear a prioridade deste tipo de operadores. Após essa divisão, cada uma das partes obtidas é enviada ao *Avaliador de Termos*. Se não existirem operadores lógicos, a regra inteira é enviada ao Avaliador de Termos.

Com relação ao **Avaliador de Termos**, ocorre uma nova divisão nos termos, mas desta vez a partir dos *operadores relacionais* ("==", ">", "<", ">=", "<=" e "! ="), se existirem. Desta forma, cada termo será dividido em duas partes: o que está a *esquerda* e o que está a *direita* do respectivo operador relacional. Assim como no Avaliador Regras, os termos divididos são enviados ao *Avaliador de Operandos*.

Finalmente, no **Avaliador de Operandos**, é verificado o *tipo de regra* existente, montando-se assim o cenário correspondente ao tipo especificado. Conseqüentemente, na execução deste cenário, a falha será imediatamente ativada, através do método crashOn(). Após a execução deste método, a regra off é submetida ao Avaliador de Regras, iniciando-se assim a sua avaliação a partir do mesmo processo descrito para a regra on.

5. Atuação de Falhas

Conforme descrito em [M. P. Barcellos et al. 2005], a injeção de falhas no SimmFI é realizada através do mecanismo de **extensão**. Desta forma, cada componente sujeito a falhas é implementado como uma *subclasse* da respectiva classe que representa o componente. Assim, nesta subclasse, coexistirão as funcionalidades específicas referentes a cada tipo de falha expressa no modelo, bem como as funcionalidades do próprio componente, obtidas através do mecanismo de herança.

Seguindo este raciocínio, as classes Node, Path e Group, existentes no Simmcast, foram estendidas para as subclasses FallibleNode, FalliblePath e FallibleGroup, representando Nodos, Caminhos e Grupos sujeitos a falhas, respectivamente. Nestas subclasses, tem-se um conjunto de métodos em comum, representando os tipos de falhas que

os componentes podem sofrer no decorrer de um experimento, de acordo com o modelo de falhas mencionado na Seção 3.1. Com relação ao conjunto de métodos, note-se que os parâmetros on e off correspondem às regras de *ativação* e *desativação* da respectiva falha representada no método, descritas na Seção 4. Estas subclasses, juntamente com o conjunto de métodos em comum, são ilustradas na Figura 6.

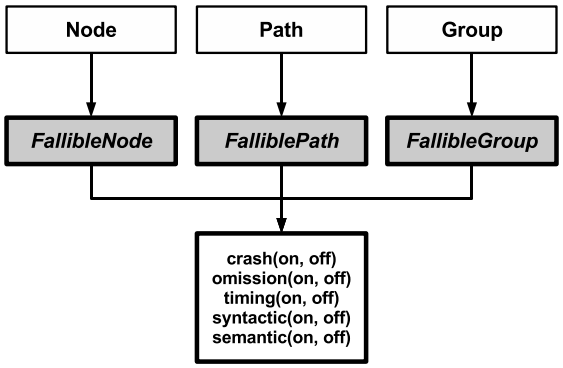


Figura 6. Subclasses dos componentes sujeitos a falhas.

Isto posto, tem-se que cada tipo de falha induz a um determinado **comportamento**, como já descrito na Seção 3.1. Logo, para cada um dos métodos em comum existentes nas subclasses acima, existirá um respectivo comportamento implementado, para a aplicação da respectiva falha no componente desejado. Por este motivo, serão descritas nas subseções seguintes as **implementações** realizadas dos comportamentos referentes às falhas de **Colapso** e **Omissão**, referentes ao componente Nodo.

5.1. Colapso em Nodo

O comportamento de uma falha de colapso possui duas características fundamentais: a *interrupção de execução* e a *reinicialização de estado*. Na interrupção, a execução do componente é imediatamente encerrada, enquanto que na reinicialização, as informações internas são perdidas e, desta forma, o componente é configurado com um estado correspondente ao início do experimento, obtendo-se assim uma amnésia total de estado.

No contexto da falha aplicada à ferramenta proposta, estas duas características fizeram-se presentes. Para a interrupção de execução, fez-se necessária modificações na classe Process, pertencente à camada Engine, de forma que um processo pudesse ter a sua execução interrompida. Para isso, foi adaptada na respectiva classe a inclusão do método interrupt(), pertencente à classe java.lang.Thread. Com relação a recuperação, a mesma é realizada a partir de uma *nova instância* ao objeto representado pelo processo, eliminando-se as informações existentes antes da falha (amnésia) e, conseqüentemente, obtendo-se uma nova execução.

5.2. Omissão em Nodo

Em uma falha de omissão, o componente Nodo deixa de prestar o serviço oferecido pelo mesmo aos demais nodos do sistema. Neste caso, o envio de pacotes é comprometido, mas sem prejuízos ao seu funcionamento interno, como o próprio recebimento de pacotes. Logo, o envio de pacotes é a principal característica a ser considerada neste tipo de falha.

Assim, com relação à ferramenta proposta, foi implementado um mecanismo de interrupção no envio. Desta forma, após a aplicação da falha de omissão, todo e qualquer pacote enviado nunca chegará ao Caminho, apesar de deixar com sucesso o contexto da aplicação. Assim como a falha de Colapso, os pacotes enviados durante a falha de omissão não são recuperados posteriormente, tendo-se assim um comportamento com amnésia. Na recuperação, a respectiva fila de envio é desbloqueada, retomando-se assim o envio normal de pacotes.

6. Testes: Metodologia e Resultados

Para testar o protótipo do mecanismo de ativação e desativação de falhas, bem como as falhas já implementadas, foi utilizado um modelo simplificado de experimento. O intuito deste modelo é demonstrar a forma pelo qual um determinado pacote é afetado, de acordo com a falha aplicada. Assim, o dado modelo tem como objetivo servir como prova de conceito para a construção de futuros experimentos mais complexos.

O modelo em questão é implementado através de uma topologia 1:1, ou seja, um Nodo origem e um Nodo destino, sendo ambos interligados através de um Caminho. Durante o experimento, um pacote é criado e enviado pelo Nodo origem ao Nodo destino. Nesta criação e envio, são registrados todos os estados pelos quais o pacote trafega, conforme descrito na Seção 3.1. A fim de facilitar a visualização na saída do experimento, o próximo pacote somente é enviado pelo Nodo origem após o pacote atual atingir o estado U2 (último estado possível). A Figura 7 ilustra um exemplo de execução deste modelo, onde são enviados dois pacotes sem a ocorrência de falhas. Vale salientar que, ao lado esquerdo de cada uma das mensagens, é exibido o valor corrente do relógio de simulação.

```
-------------------------------------------
0.0: packet ID=0 is in U1 state.
0.0: packet ID=0 is in TxQ state.
0.0: packet ID=0 is in InTx state.
1.0: packet ID=0 is in InP state.
2.0: packet ID=0 is in InRx state.
2.0: packet ID=0 is in RxQ state.
2.0: packet ID=0 is in U2 state.
-------------------------------------------
3.0: packet ID=1 is in U1 state.
3.0: packet ID=1 is in TxQ state.
3.0: packet ID=1 is in InTx state.
4.0: packet ID=1 is in InP state.
5.0: packet ID=1 is in InRx state.
5.0: packet ID=1 is in RxQ state.
5.0: packet ID=1 is in U2 state.
```

Figura 7. Execução de um experimento sem falhas.

Desta forma, a partir do protótipo e modelo existentes, para cada um dos dois tipos de falhas implementados (colapso e omissão em Nodos), foram efetuados três experimentos, utilizando-se assim falhas *efêmeras*, *temporárias* e *permanentes*, descritas em [M. P. Barcellos et al. 2005]. A sintaxe no arquivo que descreve a simulação (.sim) das chamadas executadas para cada um destes experimentos é ilustrada na Figura 8.

```
SOURCE crash "T==1" "true"
SOURCE crash "T==1" "T==4"
SOURCE crash "T==1" "false"
```

(a) Colapso.

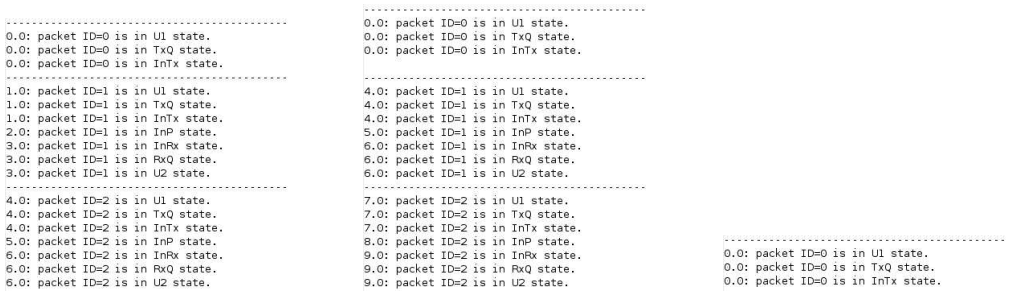
```
SOURCE omission "T==1" "true"
SOURCE omission "T==1" "T==4"
SOURCE omission "T==1" "false"
```

(b) Omissão.

Figura 8. Chamadas executadas para cada um dos experimentos.

Com relação à falha de colapso, observa-se que o pacote identificado como ID=0 é perdido, uma vez que, no momento da ativação da falha, o mesmo encontra-se presente

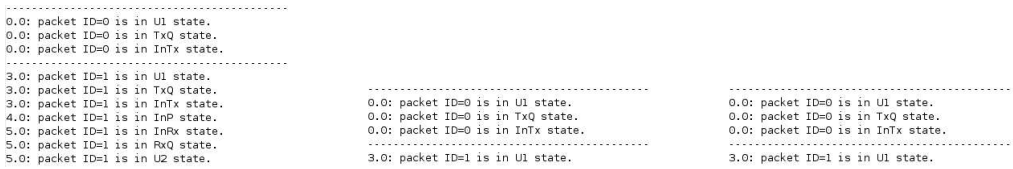
no Nodo. Como o colapso é com amnésia, o Nodo origem envia novamente dois pacotes logo após a sua recuperação, pois o estado anterior à ocorrência da falha não é conhecido pelo mesmo. A Figura 9 mostra a execução quando esta falha ocorre.



(a) Efêmera. (b) Temporária. (c) Permanente.

Figura 9. Execução do experimento com falha de Colapso.

Já na falha de omissão, constata-se uma semelhança nos resultados das falhas temporária e permanente, devido a regra de desativação utilizada na falha temporária: antes do tempo 4, há um segundo pacote sendo transmitido pelo Nodo origem, que também será omitido. A execução desta falha é ilustrada na Figura 10.



(a) Efêmera. (b) Temporária. (c) Permanente.

Figura 10. Execução do experimento com falha de Omissão.

7. Conclusões

Neste artigo foi apresentada a arquitetura e implementação do SimmFI, um artefato de software para permitir a construção de simulações de sistemas distribuídos com injeção de falhas. Foram abordadas a arquitetura do framework de simulação, bem como o projeto conceitual e os mecanismos de ativação/desativação e injeção de falhas, vistos no contexto da ferramenta. Ao final, testes de execução foram realizados, servindo como prova de conceito para utilização da respectiva ferramenta em trabalhos futuros.

Apesar do projeto e implementação terem sido completados em sua maior parte, este artigo reporta um trabalho em andamento. Os procedimentos de teste realizados indicam que os mecanismos foram implementados corretamente, oferecendo uma segurança relativa quanto ao funcionamento correto do simulador. Na escrita deste documento, a implementação encontra-se em estágio final, e um conjunto mais amplo de testes está sendo preparado, incluindo um exemplo mais complexo em *Redes Peer-to-Peer* para avaliar o SimmFI tanto em amplitude (todos os tipos de falha) como em profundidade (conseqüências para o sistema).

Referências

(2006). The Network Simulator VINT ns-2. http://www.isi.edu/nsnam/ns. Acesso em: maio 2006.

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

Ambrosio, A. M. (2005). *CoFI: Uma Abordagem Combinando Teste de Conformidade e Injeção de Falhas para Validação de Software em Aplicações Espaciais*. PhD thesis, Instituto Nacional de Pesquisas Espaciais, São José dos Campos.

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.

H. H. Muhammad and M. P. Barcellos (2001). Simulation group communication protocols through an object-oriented framework. In SCS, editor, *35th Annual Simulation Symposium, ANSS 2001*, volume 1, San Diego, USA. SCS.

Hadzilacos, V. and Toueg, S. (1998). *Distributed Systems*, chapter 5, Fault-Tolerant Broadcasts and Related Problems, pages 97–146. Addison-Wesley, 2nd. edition.

Jalote, P. (1998). *Fault Tolerance in Distributed Systems*, chapter 2, Distributed Systems, pages 45–76. Prentice-Hall.

M. P. Barcellos, C. Woszezenki, and R. Munaretti (2005). Framework de Injeção de Falhas Simulada para Avaliação de Sistemas Distribuídos. In SBC, editor, *XXIII Simpósio Brasileiro de Redes de Computadores, SBRC 2005*, volume 1, Fortaleza, Brasil. SBC.

M. P. Barcellos, G. Facchini, L. F. Cintra, and H. H. Muhammad (2004). Projeto do Framework de Simulação Simmcast: uma Arquitetura em Camadas com Ênfase na Extensibilidade. In SBC, editor, *XXII Simpósio Brasileiro de Redes de Computadores, SBRC 2004*, volume 1, Gramado, Brasil. SBC.

Nagaraja, K., Li, X., Zhang, B., Bianchini, R., Martin, R., and Nguyen, T. (2003). Using Fault Injection and Modeling to Evaluate the Performability of Cluster-Based Services. In Proceedings of the Usenix Symposium on Internet Technologies and Systems.

P. Veríssimo and L. Rodrigues (2001). *Distributed Systems for System Architects*, chapter 6, Fault-Tolerant Systems Foundations, pages 171–192. Kluwer Academic Publishers.

R. de M. Trindade, M. P. Barcellos, and I. J. Porto (2002). Simulação de Sistemas Distribuídos em Cenários com Defeitos. In SBC, editor, *III Workshop de Testes e Tolerância a Falhas - WTF2002*, Búzios, RJ.

Urbán, P., Défago, X., and Schiper, A. (2001). Neko: A Single Environment to Simulate and Prototype Distributed Algorithms. In *Proc. of the 15th Int'l Conf. on Information Networking (ICOIN-15)*, Beppu City, Japan. Best Student Paper award.