Framework de Injeção de Falhas Simulada para Avaliação de Sistemas Distribuídos

Marinho P. Barcellos, Cristiane R. Woszezenki, Ruthiano S. Munaretti

[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

{marinho,ruthiano}@unisinos.br, crisrw@inf.pucrs.br

Abstract. *Simulation is an important methodology for the evaluation of computing systems. Fault injection, on its turn, is a technique that allows* reliable systems *to be evaluated, and helps to understand how these systems behave under faults. This paper draws from those two areas to present Simmcast-FT, a framework for the simulation of fault tolerant distributed systems. It presents the conceptual design of the system, defining how each type of fault of the adopted model is mapped onto fundamental components, namely Node, Path and Group, of Simmcast. The paper presents also an elegant and flexible approach for the specification of fault scenarios, based on the specification of activation and deactivation rules.*

Resumo. *Simulação é uma metodologia importante para avaliação de sistemas computacionais. Injeção de falhas, por sua vez, é uma técnica que permite a avaliação de* sistemas confiáveis*, e que possibilita entender como esses sistemas se comportam na presença de falhas. Alicerçado nestas áreas, o presente trabalho apresenta um* framework *para simulação de sistemas distribuídos tolerantes a falhas, denominado Simmcast-FT. O artigo oferece uma visão do projeto conceitual do sistema, definindo como cada uma das falhas do modelo adotado é mapeada em componentes fundamentais do Simmcast: Nodo, Caminho e Grupo. Apresenta, também, uma forma elegante e flexível para a especificação de cenários de falhas, através da especificação de regras para sua ativação e desativação.*

1. Introdução

Avaliação de dependabilidade visa quantificar a dependabilidade de um sistema ([Veríssimo and Rodrigues, 2001]), analisando o comportamento do sistema na presença de falhas em seus componentes. Existem diferentes metodologias para a avaliação de dependabilidade, como *modelagem analítica* e *injeção de falhas*. A primeira é um mecanismo de prova das propriedades que tem sido usado com sucesso ao longo dos anos, porém sua aplicabilidade e precisão são bastante restritas em sistemas mais complexos ([Arlat et al., 2003]). *Injeção de falhas* (IF), por sua vez, é uma técnica que provoca falhas de forma controlada sobre o sistema em questão, ajudando a entender o comportamento do mesmo na presença de falhas. IF permite a identificação de erros de implementação em mecanismos de tolerância a falhas e o fornecimento de informações sobre a eficiência dos mesmos.

Existem duas categorias principais de experimentos com injeção de falhas: IF física e IF simulada, sendo o segundo tipo o foco deste trabalho. Enquanto métodos de IF física requerem uma implementação do sistema alvo, a IF simulada pode ser usada sobre um modelo abstrato antes de sua implementação. Essa técnica oferece um bom grau de

controlabilidade (habilidade de controlar onde e quando as falhas são injetadas) e *observabilidade* (habilidade de monitorar eventos internos dos componentes) no processo de injeção de falhas ([Ejlali et al., 2003]).

Esse trabalho apresenta o *framework* de simulação Simmcast-FT, que tem como objetivo permitir a execução de experimentos de simulação com sistemas distribuídos tolerantes a falhas. O Simmcast-FT, uma extensão do Simmcast ([Barcellos et al., 2001, Muhammad and Barcellos, 2002]), emprega uma variante da injeção de falhas simulada para permitir a avaliação da dependabilidade e desempenho de um sistema distribuído através da modelagem e simulação.

O artigo oferece uma visão global do Simmcast-FT, justificando as principais decisões de projeto. Ênfase é dada à definição da semântica adotada na simulação de falhas e nos recursos disponíveis para especificar um *comportamento falho,* ou seja, **quais** tipos de falhas podem ocorrer em um dado componente e **quando** elas ocorrem. Um *cenário de falhas* consiste no conjunto de comportamentos de falha especificados para uma simulação.

O restante do documento é estruturado da seguinte maneira: A Seção 2 apresenta trabalhos relacionados, enquanto a Seção 3 revisa brevemente o Simmcast e seus componentes básicos. A Seção 4 introduz o Simmcast-FT e discute como as falhas do modelo adotado se refletem nos componentes fundamentais do Simmcast. A Seção 5 ilustra a especificação de condições de falhas, permitindo um maior grau de controlabilidade nos experimentos. Uma avaliação qualitativa da abordagem proposta é apresentada na Seção 6, e comentários finais e trabalhos futuros encerram o artigo na Seção 7.

2. Trabalhos Relacionados

O Simmcast-FT representa uma convergência de duas áreas: *simulação discreta de sistemas* e *computação tolerante a falhas*. A execução de um experimento no Simmcast envolve a criação de um modelo de simulação com duas partes: uma representa o *sistema alvo* (por exemplo, um protocolo distribuído), outra o *sistema físico subjacente* (computadores e a rede que os interliga). No Simmcast-FT, as falhas são injetadas (simuladas) na parte correspondente ao sistema subjacente, de maneira a permitir o monitoramento, medição e compreensão do sistema alvo sobre condições de falhas.

No campo da simulação, a abordagem que mais se aproxima à presente proposta é o Neko ([Urbán et al., 2001]). Neko é um ambiente de simulação que auxilia no projeto, prototipagem e avaliação de desempenho de algoritmos distribuídos. Assim como o Simmcast-FT, o simulador foi desenvolvido em Java e usado na avaliação de protocolos de comunicação em grupo. Diferentemente do Neko (e do Simmcast), o Simmcast-FT oferece funcionalidade específica para a condução de experimentos com injeção de falhas sobre sistemas distribuídos tolerantes a falhas.

No campo da injeção de falhas, o Simmcast-FT relaciona-se a trabalhos em IF simulada assim como IF em sistemas distribuídos. O ORCHESTRA ([Dawson et al., 1996]) é uma ferramenta desenvolvida para testar a dependabilidade e propriedades temporais de protocolos distribuídos. Ao contrário da IF simulada, essa abordagem depende de uma implementação do protocolo distribuído. O ORCHESTRA fornece um *framework* baseado em *script* para controlar a injeção de falhas e investigar o protocolo implementado.

Apesar de existirem várias ferramentas desenvolvidas para avaliação da dependabilidade de sistemas por meio da IF simulada, essa técnica é pouco explorada no âmbito específico dos sistemas distribuídos. A maioria das ferramentas que utiliza essa técnica é voltada para simulações em nível de hardware ([Choi and Iyer, 1992, Jenn et al., 1994, Sieh et al., 1997]).

A abordagem de injeção de falhas mais próxima daquela proposta neste trabalho é o DEPEND ([Goswami, 1997]). Uma ferramenta de simulação baseada em processo, o DEPEND fornece um ambiente integrado de projeto e injeção de falhas para análise do nível de dependabilidade do sistema. A biblioteca DEPEND é composta por objetos que simulam os componentes fundamentais da maioria das arquiteturas tolerante a falhas: CPUs, processadores (com auto-checagem), processadores com redundância *N*-modular, enlaces de comunicação, votadores, e memória. O DEPEND usa um modelo de falhas funcional para simular a manifestação das falhas. Dado que os componentes representam elementos específicos de uma arquitetura real, cada componente possui um modelo de falhas específico. Segundo [Goswami, 1997], o DEPEND possibilita a condução de experimentos com injeção de falhas também em sistemas distribuídos.

O modelo de falhas adotado pelo Simmcast-FT (descrito na Seção 4) reflete a sua ênfase em sistemas distribuídos. O projeto do sistema busca escolhas racionais e elegantes: falhas são aplicadas de forma **simétrica** (similar) em componentes sujeitos a falhas, denominados *componentes falháveis*. Ao simular-se um sistema distribuído, as falhas são injetadas dinamicamente nos componentes fundamentais do sistema em momentos específicos e/ou segundo dadas condições (conforme explicado na Seção 5).

O Simmcast-FT permite que um sistema seja testado usando implementações parciais (protótipos) de um sistema, oferecendo *feedback* valioso para o projetista durante o processo de desenvolvimento. Ao contrário das abordagens que empregam uma "biblioteca de falhas", falhas são fornecidas através de **extensão**: as classes que definem os componentes são estendidas (herdadas e modificadas) de maneira a incluir um comportamento de falha. Essa abordagem fornece **transparência**, uma vez que não são necessárias mudanças no código-fonte do sistema distribuído alvo (no que diz respeito a experimentos de simulação com injeção de falhas). Por fornecer regras flexíveis para ativar falhas em componentes subjacentes, o Simmcast-FT possibilita ao usuário manipular um protocolo em estados específicos e disparar falhas quando desejado. Os aspectos acima serão explorados nas seções subseqüentes.

3. Simmcast

O Simmcast ([Barcellos et al., 2001]) é um *framework* de simulação orientado a objetos, para pesquisa de protocolos de rede e sistemas distribuídos. Ele emprega um *modelo de eventos discretos baseado em processos*, no qual componentes são combinados e estendidos a fim de criar novos ambientes de simulação. Esta seção descreve os conceitos do Simmcast fundamentais ao artigo, e a visão é orientada nesse sentido.

3.1. Componentes

No Simmcast, uma simulação é descrita pela combinação de componentes (também denominados "blocos de construção"), fornecendo código adicional quando necessário. Os componentes utilizados no Simmcast são apresentados a seguir. Para cada componente, existe uma classe correspondente no simulador.

Nodos são as entidades fundamentais de interação e podem representar diferentes papéis, dependendo do sistema alvo sendo simulado e o nível de detalhamento empregado. Um Nodo contém uma ou mais **Threads** de execução, que podem representar no framework threads de um processo (no espaço de endereçamento deste) ou processos de um mesmo computador (mesma memória física).

Nodos são conectados através de **Caminhos**. Seu significado no modelo simulado, assim como os Nodos, depende do sistema alvo e do nível de abstração escolhido. Dessa

forma, um Caminho pode representar, por exemplo, uma conexão via protocolo TCP entre os múltiplos processos de um sistema distribuído, ou uma das interligações lógicas de uma relação muitos-para-muitos que se materializa via protocolo UDP. Pode representar, também, uma fila de pacotes ou um enlace físico entre dois nodos de rede (computadores ou roteadores).

Para facilitar a simulação de protocolos multicast, o Simmcast oferece o componente básico **Grupo**. Um Grupo é uma entidade global da simulação que tem um identificador análogo ao Nodo, porém refere-se a um conjunto de Nodos. A semântica de Grupo no Simmcast é bem simples: quando um Nodo envia uma mensagem ao Grupo, a mesma é encaminhada para cada membro do Grupo que seja vizinho direto do Nodo que envia a mensagem[1].

Uma **Rede** é uma combinação arbitrária de Nodos e todos os Caminhos que os conectam. Uma mensagem (ou **Pacote**) é uma unidade de comunicação entre dois ou mais Nodos. As mensagens contêm um conjunto de atributos, mas pode ser estendido pelo usuário para armazenar dados relevantes do protocolo.

3.2. *Application Program Interface* (API)

A API do Simmcast é composta pela interface das classes disponíveis nos pacotes Simmcast. Apesar de extensa em termos de recursos, a API oferece apenas dez primitivas, as quais podem ser alocadas a três categorias:

- **comunicação**: send(), receive(), tryReceive(), que servem para envio e recepção de mensagens, e join() e leave(), para entrada e saída de grupos, respectivamente;
- **tratamento de eventos assíncronos**: setTimer(), cancelTimer() e onTimer(), que correspondem à criação, remoção e tratamento de um evento, respectivamente;
- **manipulação de *threads***: sleep() e wakeUp(), que fazem com que uma *thread* durma ou que uma *thread* acorde outra.

Para criar uma simulação, além da combinação dos componentes acima mencionados, é necessário estender as classes do *framework* de acordo com as peculiaridades do sistema alvo. Parte da funcionalidade já está presente no *framework* e, dessa forma, pode ser reutilizada, evitando replicação de código.

De forma geral, um novo sistema ou protocolo pode ser criado através da definição de Nodos, Threads, uma Rede e um arquivo de configuração da simulação. Os Nodos usados no experimento estendem a classe Node do Simmcast. As Threads definem o comportamento do Nodo, estendendo a classe NodeThread. As Threads do Simmcast contêm um método execute(), onde a lógica do protocolo é adicionada. Um arquivo texto (**.sim**), descreve um cenário de simulação: os Nodos necessários são declarados, suas conexões (ou seja, a topologia da rede), bem como demais parametrizações do experimento.

Uma vez que o Simmcast estende a biblioteca do JavaSim ([Little, M. C., 2004]), ele fornece dois tipos de facilidades estatísticas tipicamente necessárias em estudos de simulação. Primeiro, valores randômicos de acordo com uma distribuição podem ser obtidos por meio de *streams aleatórias*. Existem diversos tipos de distribuições randômicas disponíveis: Random, Uniform, Exponential, Erlang, HyperExponential, Normal. Objetos dessas classes podem ser usados para controlar vários parâmetros do cenário da simulação.

A segunda facilidade diz respeito a um conjunto de classes responsáveis pela coleta de dados e análises estatísticas. As classes disponíveis são: Mean, Variance, Histogram

[1]Dessa forma, quando são utilizados os Grupos, tipicamente (mas não necessariamente) a topologia lógica será totalmente conexa.

e TimeHistogram. Uma vez criado, um objeto (de tais classes) pode "receber" valores quando são medidos durante o experimento; em qualquer momento, o objeto pode ser acessado para obter estatísticas sobre valores coletados.

3.3. Transmissão de Mensagens

Central a este trabalho é a noção de como mensagens são transmitidas no Simmcast. A Figura 1 ilustra os diferentes estados através dos quais uma mensagem transita durante sua vida. Na figura, Nodos na esquerda e direita são *adjacentes* ao Caminho.

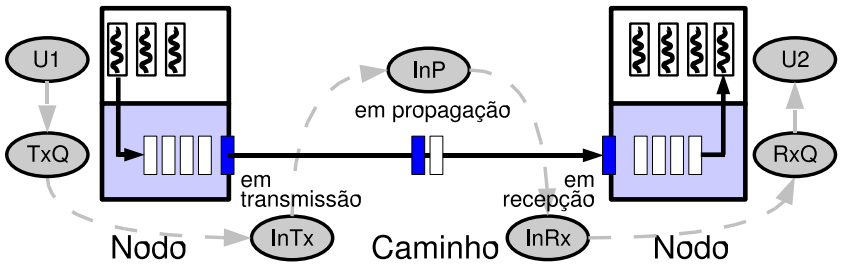

Figura 1: Diagrama de estados de uma mensagem.

Primeiramente, a aplicação no Nodo da esquerda prepara uma mensagem, que corresponde ao estado U1, e solicita seu envio (com send() ou uma variante). A mensagem então é instantaneamente copiada e acrescentada à fila de envio (estado TxQ), caso haja espaço na mesma. A mensagem mais tarde alcança o início da fila e passa a estar em transmissão (estado InTx) - gradualmente entrando no Caminho. Quando a mensagem deixa o nodo e está completamente no Caminho, ela está em propagação (estado InP). Quando ela começa a chegar no Nodo da direita, diz-se que está em recepção (estado InRx). Assumindo existir espaço na fila nesse momento, o conteúdo da mensagem é salvo na mesma (estado RxQ). Mais tarde, a mensagem alcança o início da fila e aguarda o momento de ser entregue, sob requisição, para a camada superior. Quando ocorre, a entrega da mensagem é instantânea, e a mesma passa ao estado U2.

4. Injeção de Falhas

Embora o Simmcast original permita a execução de experimentos com sistemas distribuídos, a noção de falha é restrita: Caminhos podem descartar mensagens de forma probabilística, ou em Nodos, por contenção de recursos através de filas de tamanho finito. Esta seção demonstra quais os componentes básicos do Simmcast foram imbuídos de um comportamento de falha, que tipo de falhas são oferecidas (isto é, modelo de falhas do Simmcast-FT) e o comportamento dos componentes quando da ativação e atuação de falhas.

Relembrando, os componentes básicos do Simmcast são: Nodo, Thread, Caminho, Mensagem (Pacote), Grupo e Rede. No entanto, falhas são previstas apenas em Nodos, Caminhos e Grupos, conforme justificado a seguir.

Threads não possuem comportamento de falha próprio porque, no mundo real, compartilham um mesmo espaço de endereçamento de um processo e, portanto, assume-se que não falham de forma isolada. Mensagens não possuem comportamento de falha próprio porque as mesmas são de caráter temporário. Como elas não fazem parte da configuração (descrição) de um experimento de simulação, não seria possível ou adequado atribuir um comportamento de falhas a mensagens individuais. Ao invés disso, mensagens são afetadas por falhas nos componentes pelos quais as mesmas circulam, Nodos, Caminhos e Grupos.

O componente Rede de uma simulação não possui comportamento próprio de falha porque o mesmo é formado pelo conjunto completo de Nodos e Caminhos; nesse caso, não faria sentido, portanto, provocar uma falha no sistema inteiro.

Dessa forma, os componentes ditos falháveis são Nodos, Caminhos e Grupos. A ocorrência de falhas em Nodos causa erros (estado inconsistente) e se reflete nos resultados emitidos pelo Nodo bem como no seu envio de mensagens. Caminhos não possuem estado interno, e falhas nos mesmos se refletem nas mensagens que eles transportam. Falhas em Grupos afetam quaisquer mensagens que são enviadas ao mesmo. O conjunto de falhas admitido pelo Simmcast-FT, apresentado na Tabela 1, corresponde ao modelo de falhas proposto por Veríssimo em [Veríssimo and Rodrigues, 2001]. Esse modelo foi escolhido por possuir o nível de abstração apropriado ao esquema de simulação proposto e por ser orientado a sistemas distribuídos.

Tabela 1: Tipos de Falha.

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

A interface da classe que corresponde ao componente passa a oferecer um conjunto de métodos relacionado às falhas (crash(), omission(), etc.), um para cada tipo de falha, usados para ativar e desativar falhas em objetos dessa classe (criando um comportamento de falhas para o componente). Tais métodos tomam como argumentos uma *regra de ativação* e uma *regra de desativação* (a discussão sobre métodos e regras é postergada até a Seção 5). A seguir, a Seção 4.1 descreve os princípios gerais que regem o comportamento de cada componente perante cada tipo de falha, enquanto a Seção 4.2 detalha as ações específicas tomadas no instante em que uma falha é ativada.

4.1. Comportamento de componentes falhos

Falha de colapso de Nodo é bastante popular na modelagem de sistemas distribuídos. Quando um Nodo falha por colapso, seu processamento é interrompido (parado) e suas informações de estado perdidas (amnésia). Um Nodo pode se recuperar de um colapso, sendo reinicializado com estado correspondente ao início do experimento. Quando um Caminho sofre uma falha de colapso, o transporte de mensagens pelo mesmo é encerrado imediatamente. A falha de colapso de um Grupo faz com que todas as mensagens enviadas ao mesmo sejam silenciosamente descartadas.

A **falha de omissão** faz com que o Nodo omita uma ação que é esperada do mesmo. Dessa forma, se um Nodo implementa um servidor, ao falhar por omissão, poderá receber requisições de clientes, processá-las e enviar respostas, mas estas não serão entregues aos clientes (para os clientes, o serviço está sendo omitido). Um Caminho, ao sofrer uma falha de omissão, omite a ação esperada não transportando a mensagem que lhe é repassada pelo Nodo adjacente; essa omissão pode ser total ou parcial, de acordo com a probabilidade que é fornecida quando da especificação de um comportamento de falha (explicado na Seção 5). Similarmente, mensagens enviadas ao Grupo são submetidas a um descarte probabilístico no transmissor, fazendo com o que grupo como um todo seja afetado ou nenhum membro. Falhas de omissão individuais são obtidas conforme descrito acima, para um Nodo.

Uma **falha temporal** em um Nodo faz com que o valor do relógio local do Nodo seja alterado, o que poderá levar ao envio de mensagens fora de hora. Quando um Caminho é

afetado por uma falha de temporização, as mensagens transportadas pelo mesmo são atrasadas. Uma mensagem enviada a um Grupo com falha de temporização é sujeita a atrasos especificados. Note-se que, neste caso, o mesmo atraso (fixo ou aleatório) é aplicado a todas as cópias de uma dada mensagem em transmissão, e que isso difere de uma falha de temporização aplicada individualmente a cada um dos Caminhos através dos quais as mensagens do Grupo passarão, pois apenas mensagens destinadas ao Grupo serão afetadas.

A ocorrência de uma **falha sintática** em um componente faz com que o mesmo exiba um comportamento que é incorreto, porém detectável por Nodos saudáveis. Falhas sintáticas em Nodos ocasionam o envio de mensagens comprometidas por uma sintaxe incorreta. Em Caminhos, corrompem mensagens transportadas pelo mesmo. Uma falha sintática em um Grupo faz com que todas as mensagens enviadas através do mesmo sejam corrompidas.

A **falha semântica** de um Nodo, quando ativada, faz com que o mesmo passe a enviar mensagens que são sintaticamente corretas, porém contendo valores incorretos. Um Caminho afetado por uma falha semântica faz com que todas as mensagens transportadas sejam semanticamente afetadas. Por fim, falhas semânticas aplicadas a um Grupo fazem com que todas as mensagens que sejam enviadas ao Grupo sejam alteradas.

4.2. Ativação de falhas

No momento da ativação de uma falha sobre um componente, mensagens em **contato** com o mesmo são imediatamente afetadas. A seguir, é definido e explicado quais mensagens estão em contato com um componente, baseando-se nos estados de mensagens apresentados na Seção 3.3.

Mensagens que estão nas filas de envio ou recebimento (estados TxQ e RxQ, respectivamente) de um Nodo estão em contato com o mesmo e portanto são afetadas. Quando começam a ser transmitidas (estado InTx), mensagens encontram-se em contato com o Nodo transmissor **e** com o Caminho, sendo o equivalente aplicável à recepção (estado InRx). Em contraste, mensagens em propagação (estado InP) ficam em contato apenas com o Caminho.

O conceito de Grupo envolve múltiplos Nodos e Caminhos; arbitra-se, neste caso, que o sistema de Grupo abrange do estado TxQ ao RxQ. Em outras palavras, uma mensagem "entra" no sistema de grupo quando passa de U1 para TxQ, e o "deixa" quando passa de RxQ para U2. Mensagens que estão no espaço da aplicação merecem consideração especial: em princípio, assume-se que mensagens em U1 ou U2 não estejam em contato com o sistema físico subjacente e, portanto, não estejam em contato com nenhum componente "falhável". No entanto, esse não é o caso da falha de colapso, conforme discutido mais tarde.

A lógica acima é expressa através de uma Matriz de Contato, M, conforme a Tabela 2. Estados s são pertencentes ao conjunto S = {U1, TxQ, InTx, InP, InRx, RxQ, U2}, componentes c são pertencentes ao conjunto C = {Nodo, Caminho, Grupo} e valores booleanos b são pertencentes ao conjunto B = {0, 1}. Portanto, a matriz é definida por $M(c,s) = b$. Na matriz representada, mensagens transitam de estado, temporalmente, da esquerda para direita.

Tabela 2: Matriz de Contato *M*, *c*×*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

Para todos os casos em que $b = 1$, a mensagem no estado s é afetada quando ocorre uma falha no componente c. Em outras palavras, a matriz M demonstra se uma mensagem está em contato ou não com um determinado componente e, portanto, expressa também se a mensagem é afetada ou não por uma falha no momento de ativação. Entretanto, a matriz não demonstra *como* a mensagem é afetada.

Complementando a tabela, define-se a seguir a maneira como uma mensagem é afetada por uma falha, quando este for o caso. Este efeito é dependente do tipo de falha, e não do estado da mensagem. Com isto, define-se um mapeamento direto entre o tipo de falha e a ação sobre a mensagem, conforme a seguir.

Tomando-se o conjunto de falhas F e definindo-se o conjunto possível de efeitos E, têm-se: F = {colapso, omissão, temporização, sintática, semântica} e E = {descarte, descarte probabilístico, atraso, corrupção, alteração}. Define-se a função A: $F \rightarrow E$ = {(colapso, descarte), (omissão, descarte probabilístico), (temporização, atraso), (sintática, corrupção), (semântica, alteração)}.

Considere um componente c, um estado s e uma falha f. Quando c e s satisfazem $M(c,s) = 1$, aplica-se $A(f)$ sobre a(s) mensagem(ns) no estado s. Ou seja, primeiro determina-se se a mensagem está em contato; caso sim, essa mensagem é afetada quando da ocorrência de uma falha. O efeito depende do tipo de falha: o mapeamento de ação para falha é unijetor e, salvo as duas exceções abaixo, independente do estado da mensagem:

- apesar de M(Nodo, U1) = 0 e M(Nodo, U2) = 0, em uma falha f = colapso **existe** a *A(f),* sendo portanto a mensagem afetada se no estado U1 ou U2;
- apesar de M(Nodo, InRx) = 1 e M(Nodo, RxQ) = 1, em uma falha f = omissão **não** existe a *A(f)* aplicada sobre a mensagem no estado InRx ou RxQ.

Quanto à primeira exceção, em falhas de colapso, o componente pára de funcionar completamente. No caso de componentes com estado interno modelado, que é o caso do Nodo (mas não do Caminho ou Grupo), as informações de estado (incluindo aplicação) são perdidas. Dessa forma, mensagens ora residentes na aplicação, portanto em estado U1 ou U2, são afetadas (descartadas).

A interpretação da semântica de falha de omissão leva à segunda exceção. No caso de falha de omissão em Nodo, a aplicação que executa no mesmo deixa de prestar o serviço esperado. Para possibilitar a modelagem desse tipo de falha no ambiente simulado, optou-se por afetar apenas o envio de mensagens, e não o seu recebimento.

4.3. Atuação de Falhas

Além do seu efeito imediato sobre mensagens, a falha de um componente continua a afetar novas mensagens que entram em contato com o mesmo. A Matriz de Contato, apresentada na Tabela 2, define também o estado em que uma mensagem passa a estar em contato com o componente, ou seja, na transição de 0 para 1 mais à esquerda (temporalmente anterior). Neste ponto, é aplicado um **filtro**, que tem como objetivo atuar a falha sobre mensagens. O mesmo é aplicado nas mensagens, de acordo com os componentes, conforme a seguir.

No caso de um Caminho falho, o filtro é sempre instalado em TxQ→InTx, ou seja, quando a mensagem entra em contato com o Caminho. No caso de Grupos, o filtro reside sempre em U1→TxQ, isto é, a mensagem é processada pelo filtro quando "entra" no Grupo. Já no caso de um Nodo falho, o filtro é instalado na transição U1→TxQ, mas com duas exceções. Primeiro, em caso de colapso de Nodo, há filtro apenas para recepção (na transição InP→InRx), pois a aplicação em um Nodo em colapso não estará criando novas mensagens. Segundo, nas falhas de omissão e temporização de Nodo, o filtro é instalado na transição TxQ→InTx.

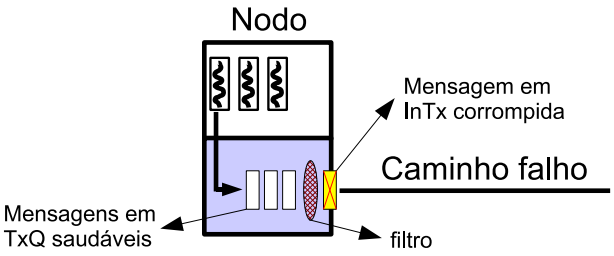

Figura 2: Atuação de uma falha sintática em um Caminho

O efeito aplicado pelo filtro é determinado pelo tipo de falha, conforme definido pela função *A(f)* (vide Seção 4.2). A Figura 2 ilustra a atuação de uma falha sintática em um Caminho, na qual o filtro é aplicado em mensagens que transitam do estado TxQ para InTx.

5. Especificação de Comportamentos de Falha

Na seção anterior foi apresentado o modelo de falhas empregado pelo Simmcast-FT com a definição dos componentes falháveis e os tipos de falhas. Esta seção descreve a forma com que usuários especificam comportamentos de falha nesses componentes. Como definido anteriormente, um comportamento de falha de um componente indica quais tipos de falhas ele pode sofrer durante o experimento de simulação, e quando exatamente essas falhas serão ativas/inativas.

Na vida real, um componente pode sofrer múltiplas falhas ao longo de sua vida. Além disso, um componente pode sofrer diferentes tipos de falha concomitantemente. O Simmcast-FT permite especificar componentes que atendam a esses requisitos: existe um método para cada tipo de falha (vide Seção 4), e cada método toma como argumento uma *regra de ativação* e outra de *desativação,* denotadas respectivamente como on e off, para especificar quando um tipo de falha deve ocorrer, e quando deve cessar.

Uma falha pode ser *efêmera*, *temporária* ou *permanente*. No primeiro caso, a falha atua sobre o componente (ações), e é imediatamente cancelada. No segundo caso, uma falha perdura por um tempo não-nulo porém inferior ao tempo total de simulação. No terceiro, como o próprio nome indica, não há recuperação.

Quando uma regra on se torna verdadeira, o componente muda imediatamente para o estado *falho* e quatro ações ocorrem instantaneamente: (i) o tempo da falha no componente é registrado como o último momento em que esse tipo de falha foi ativado nesse componente; (ii) a falha é instantaneamente aplicada ao componente; (iii) o estado do componente, que era saudável, se torna, agora, falho; (iv) a regra off é avaliada e, se for verdadeira, o componente se recupera imediatamente. Neste caso, o tempo de sua recuperação é registrado internamente no componente.

Quando uma falha não é efêmera, o comportamento falho continua atuando no componente e a regra off permanece em reavaliação enquanto o componente permanece falho. Quando off se torna verdadeira, a falha em questão é desativada (outras falhas podem permanecer ativas nesse componente). Após isso, on é avaliado para uma possível mudança no valor.

5.1. Regras de Ativação e Desativação

Existem dois tipos de regras on/off: *intervalos* e *expressões booleanas*. No primeiro caso, a regra representa o intervalo de tempo desde a última transição de estado entre falho e

saudável (no caso da primeira falha, desde o início da simulação). Este intervalo deve ser, naturalmente, um número não-negativo. Uma regra on:<valor> em um componente ativa, portanto, uma falha <valor> unidades de tempo após a última recuperação do componente para esse tipo de falha. Complementando, off:<valor> indica o intervalo de tempo até a recuperação do componente. Esse esquema é relevante porque implementa de forma simples o conceito de MTBF e MTTR, bem conhecidos na área.

O segundo tipo de regra on/off é uma expressão booleana, que é avaliada de forma contínua no tempo (a máquina de simulação, entretanto, faz a avaliação sob demanda e conforme o avanço discreto do tempo, evitando desperdício sem alterar a semântica.) No momento em que uma regra on de um componente saudável se torna verdadeira, a falha é ativada. Conversamente, uma regra off que se torna verdadeira em um componente falho faz com que o mesmo se recupere.

Na descrição de uma simulação criada através de um arquivo **.sim**, uma regra on/off é especificada e passada como parâmetro de entrada na especificação do comportamento de falha de um componente. O seguinte formato é usado:

<componente>.<falha>(<on>, <off> [, <args>])

onde <componente> representa o identificador do componente específico (que deve ser da classe Node, Path ou Group), <falha> indica o tipo de falha (crash, omission, timing, syntactic, semantic), <on> a regra de ativação, <off> a regra de desativação, e <args> são argumentos opcionais usados em determinados tipos de falhas. Argumentos adicionais são usados na especificação das seguintes combinações de falhas e componentes: probabilidade de descarte para omissão em Caminhos e Grupos; novo valor do relógio (deve ser não-negativo) e *rate of drift* (deve ser não-nulo) para falha de temporização em Nodos; e valor do atraso para falha de temporização em Caminhos e Grupos.

Regras que são expressões booleanas são compostas de *termos*; um termo é separado por um operador binário, que pode ser do tipo lógico ou relacional: "&&" (and) e "||" (or), ou "==", ">", "<", ">=", "<=" e "!=". Operadores aritméticos não são permitidos, pois seus benefícios não compensam a complexidade decorrente em termos de interface e implementação. Os termos constituintes de uma regra on/off podem fazer referência ao relógio da simulação, à troca de mensagens, a variáveis aleatórias e ao estado interno de Nodos. As próximas seções abordam cada uma dessas categorias.

5.2. Referência ao Relógio

Um dos atributos-chave da simulação é o relógio, ou o tempo de simulação (o qual é iniciado em 0 quando a simulação começa). Um termo pode fazer referência ao valor atual do relógio (denotado como T) em uma regra, conferindo caráter temporal à mesma. A seguinte sintaxe é adotada:

T <operador_relacional> <valor>

A semântica é intuitiva e é melhor explicada através de exemplos: um termo T==10 é verdadeiro quando, e somente quando, o relógio de simulação vale 10; T>=10 é verdadeiro a partir do momento 10 da simulação, inclusive, enquanto T<=20 é verdadeiro até o momento de simulação 20, inclusive. Os operadores ">" e "<" são adaptados ao tempo discreto: como o tempo não avança de forma contínua, T>10 é implementado no simulador como T+smallest_increment, sendo smallest_increment uma constante interna do simulador que representa o menor incremento de tempo possível. O equivalente se aplica a T<20.

Ilustrando, uma falha de colapso em um Nodo no tempo 5, com recuperação no tempo 15, pode ser provocada com n1.crash(T==5, T==15). Para especificar **múltiplas** falhas

em períodos fixos**,** seriam usados diretamente os valores 15 e 5, como em n1.crash(5, 15). Para simular médias como MTBF e MTTR, seriam usados objetos que representam distribuições aleatórias (*streams*), como n1.crash(MTBF, MTTR), assumindo que MTBF e MTTR são duas *streams* previamente declaradas.

5.3. Referência a Mensagens

É interessante que falhas possam ser ativadas e desativadas em função de mensagens trocadas no sistema distribuído. Isso permite, por exemplo, que uma mensagem *destrutiva* seja enviada a um Nodo, sem interferir na lógica do protocolo. Para tal, termos em regras podem fazer referências ao conteúdo de mensagens enviadas e recebidas por Nodos, ou transportadas através de Caminhos.

Esta classe de termos está baseada na especificação dos campos que são comuns a todas as mensagens, quais sejam: *origem*, *destino*, *identificador* e *conteúdo de dados*. Mensagens que são enviadas, recebidas ou transportadas por um Caminho são representadas pelas palavras chave sends, receives e transports, conforme explicado a seguir. Existem três tipos de termos que fazem referência a mensagens:

sends(<origem>, <destino>, <id>, <dados>)
receives(<origem>, <destino>, <id>, <dados>)
transports(<origem>, <destino>, <id>, <dados>)

onde <origem>, <destino>, <id> e <dados> representam respectivamente a origem, o destino, o identificador e o conteúdo de dados da mensagem. Para cada elemento, uma *expressão regular* é fornecida. A regra sends(<origem>, <destino>, <id>, <dados>) será verdadeira se, e somente se, houver um casamento dos parâmetros <id> e <dados> com uma mensagem sendo transmitida de <origem> para <destino>. O termo é avaliado no momento do envio, ou seja, quando a mensagem passa do estado U1 para TxQ. As regras transports e receives se comportam de maneira similar; entretanto, nestes casos, são avaliadas nas transições de estado InTx→InP e InP→RxQ , respectivamente.

A expressão regular ".*" casa com qualquer ocorrência. Por exemplo, a primeira parte da especificação n1.omission(receives("n1", "n2", ".*", ".*"), sends("n3", ".*", ".*", ".*suspect.*")) provoca uma falha de omissão (vide Seção 4.1) em um componente n1 no estado saudável quando n2 receber uma mensagem de n1. A segunda parte indica que n1 se recupera (portanto n1 está falho) quando o nodo n3 envia uma mensagem contendo a *string* "suspect". Note-se que é possível, neste caso e em outros, especificar uma regra de recuperação que nunca será verdadeira, como por exemplo condicionar a recuperação de colapso de um nodo à transmissão de uma mensagem pelo mesmo.

5.4. Referência a Variáveis Aleatórias

Em um experimento com injeção de falhas em um sistema distribuído, é interessante poder especificar um comportamento, para algum componente, onde falhas ocorrem de maneira não-determinística (como p.ex. ao especificar propriedades de falha via MTTF e MTTR).

Termos em regras podem referenciar *streams* que fornecem, a cada leitura, um novo valor (numérico ou booleano) que é calculado segundo uma distribuição aleatória inicializada com parâmetros como média e desvio padrão. Tais *streams* são criadas no arquivo de configuração da simulação, e então referenciadas em regras on/off.

Embora *streams* aleatórias possam ser usadas em regras do tipo booleanas, pressupõe-se que seu uso seja particularmente útil em regras do tipo intervalo. Neste caso, conforme mencionado na Seção 5.1, o valor na regra determina um intervalo de tempo relativo à última transição de estado entre saudável e falho para o componente sobre o qual a regra

atua. A maneira mais simples de se especificar, por exemplo, que um Caminho c1 deverá sofrer uma falha sintática a cada 10 unidades de tempo de operação normal, com tempo de recuperação igual a 1 unidade, é: c1.syntactic(10, 1). Neste exemplo, 10 e 1 representam valores fixos. Tipicamente, são especificados intervalos de forma probabilística, como *tempos médio até falha* e *tempo médio até recuperação*.

Considere duas *streams* uniformes denominadas MTTF e MTTR, inicializadas com os parâmetros (limite inferior e superior) 80 e 120, e 3 e 5, respectivamente. A cada vez que é acessada, MTTF retorna um valor no intervalo [80, 120], enquanto MTTR retorna valores em [3, 5]. Portanto, uma especificação de falha c1.syntactic(MTTF, MTTR) fará com que, na média, uma falha sintática ocorra a cada 100 unidades de tempo de funcionamento, e perdure, na média, por 4 unidades de tempo (tempo de recuperação na média igual a 4).

Em regras booleanas, a leitura de valores das *streams* aleatórias retorna um "valor corrente". O mesmo é calculado de forma **periódica**: isto é necessário para preservar as propriedades estatísticas esperadas da distribuição, desconsiderando o número de vezes que a *stream* é acessada. O valor *padrão* para o intervalo entre regeneração de valores, denominado I, é 1 (unidade de simulação). Este parâmetro é global a um experimento e serve a todas as *streams* aleatórias. Assim, a cada intervalo I, um novo valor é recomputado para cada *stream* aleatória existente, de forma que acessos consecutivos a uma dada *stream* retornarão o mesmo valor até o próximo intervalo. A troca do valor corrente pode fazer com que uma regra seja avaliada de forma verdadeira, provocando a ativação ou desativação de uma falha.

Existe um *trade-off* para o valor de I: quanto maior, maior será o intervalo entre gerações de números aleatórios, diminuindo a precisão da *stream*; por outro lado, quanto menor o I, maior será a freqüência com que os valores de *streams* aleatórias serão recalculados e, portanto, maior será o custo computacional da simulação.

5.5. Referência a Estado em Nodos

Por fim, há situações em que é desejável que um termo de uma regra de ativação ou desativação faça referência ao estado interno de Nodos. Neste caso, um campo de dados público da classe Node pode ser acessado diretamente por um termo de uma regra, através da seguinte sintaxe: <nodo>.<nome_campo>. Estes campos de dados podem ser usados apenas em forma de *string* e com operadores relacionais, fazendo-se um casamento exato com uma dada *string* ou com o estado de outro nodo. Portanto, uma regra de ativação com referência ao estado de um Nodo pode ser algo como:

<nodo>.<nome_campo> <operador_relacional> "<string>"

sendo <string> um literal ou um campo de dados de outro Nodo. Por exemplo, a especificação n1.crash(n2.status == "AGREE" && n3.status == "AGREE", 2) faz com que n1 sofra uma falha de colapso quando n2 e n3 estiverem com seu campo status igual à *string* "AGREE", e se recupere após 2 unidades de tempo.

6. Avaliação Qualitativa

Uma avaliação qualitativa do Simmcast-FT é fornecida abaixo, referenciando o conjunto de atributos definidos por [Arlat et al., 2003]: *alcançabilidade, controlabilidade, repetibilidade de experimentos, reprodutibilidade de resultados, não-intrusividade, medida de tempos* e *eficácia.*

Em relação à *alcançabilidade*, por um lado o Simmcast-FT permite que um conjunto extenso de falhas seja injetado de forma a atingir todos os componentes da rede, assim

como os sistemas-fim que são sujeitos a falhas no sistema real; por outro, se restringe aos componentes presentes no modelo de simulação.

A *controlabilidade* é alta, pois o Simmcast-FT fornece um mecanismo elegante e flexível para especificar precisamente quais falhas ocorrem e quando, através de um comportamento de falhas, a ser especificado para os componentes desejados. Um componente é estendido através de herança de classes, e seus novos métodos (como crash() e omission()) são invocados de acordo com o comportamento de falhas desejado. Estas condições são ditadas por regras de ativação e desativação, que podem ser determinísticas ou probabilísticas.

A *repetibilidade de experimentos* e a *reproducibilidade de resultados* são ambas altas, uma vez que os experimentos de simulação discreta podem ser executados um número arbitrário de vezes e, dado o mesmo conjunto de entradas (e sementes aleatórias), os resultados sempre serão os mesmos.

A *não-intrusividade* é alta, pois o processo de injeção simulada de falhas não apresenta sobrecarga temporal nem causa desvio da lógica de execução normal do sistema alvo. O Simmcast-FT possui boas facilidades para *medida de tempos* permitindo que a latência de erros e tempo de recuperação em sistemas distribuídos sejam medidos. Por outro lado, isso é limitado pela precisão do modelo e os parâmetros de entrada usados.

Um ponto no qual a abordagem proposta não se sai tão bem é a *eficácia*, a qual se refere a capacidade de gerar um número limitado de experimentos não relevantes. Nesse caso, o Simmcast-FT compartilha das limitações inerentes das técnicas de simulação, nas quais os experimentos são conduzidos sobre um ambiente simulado. Entretanto, diferentemente de outras ferramentas de injeção de falhas simulada, o efeito negativo da baixa eficácia no Simmcast-FT é diminuído por duas razões: primeiro, o modelo de simulação *baseado em processos* (e não em *eventos*) adotado pelo Simmcast, com a especificação de Nodos compostos por múltiplas Threads, é bem próximo da realidade e reflete uma estrutura popularmente encontrada na implementação em sistemas e protocolos. Segundo, o *framework* permite que a simulação seja desenvolvida de maneira incremental, em nível crescente de detalhe, conforme reportado em [H. H. Muhammad et al., 2004].

7. Considerações Finais

Este artigo apresentou o projeto do Simmcast-FT, um *framework* de simulação discreta para avaliação de dependabilidade e desempenho de sistemas distribuídos tolerantes a falhas. Utiliza-se a técnica de injeção de falhas simulada, sendo as falhas injetadas tipicamente nos componentes do sistema subjacente. Componentes falháveis adequados foram identificados. De forma geral, tipos de falhas podem ser definidos especificamente para cada componente, ou o mesmo conjunto de falhas pode ser aplicado a **todos** os componentes falháveis. A decisão de projeto adotada, em favor da simetria e simplicidade, consiste em oferecer um único conjunto de falhas para os três componentes derivando, assim, semânticas de falhas consistentes para cada combinação de falha e componente. Esse conjunto de falhas é composto pelas falhas de colapso, omissão, temporização, sintática e semântica.

Cada componente da simulação pode ser configurado com um comportamento falho, o qual especifica exatamente quais tipos de falhas ocorrem em cada componente e quando elas ocorrem. As regras definem os momentos exatos em que as falhas são ativadas e desativadas. Termos de regras podem variar de uma expressão simples em função temporal até a representação de um cenário intrincado.

A abordagem de injeção de falhas do Simmcast-FT permite uma maneira elegante e flexível para especificar um comportamento de falha, no qual qualquer componente pode

ter seu próprio comportamento modificado. É possível estender e sobrescrever um certo componente no *framework* de maneira a modificar a semântica de um determinado tipo de falha no mesmo. O projeto descrito neste artigo deu origem a uma arquitetura, considerando aspectos como desempenho e facilidade de implementação, e atualmente um protótipo da mesma está sendo definido.

Agradecimentos

Agradecemos a Taisy Weber e Ingrid Jansch-Pôrto pela contribuição nos estágios iniciais deste trabalho, e ao CNPq, pelo suporte financeiro a este projeto (PDPG-TI 552178).

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.

Barcellos, M. P., Muhammad, H. H., and Detsch, A. (2001). Simmcast: a simulation tool for multicast protocol evaluation. In *XIX Simpósio Brasileiro de Redes de Computadores, SBRC 2001*, volume 1, pages 418 – 433, Florianópolis, Brasil.

Choi, G. S. and Iyer, R. K. (1992). FOCUS: An Experimental Environment for Fault Sensitivity Analysis. *IEEE Transactions on Computers*, 41:1515–1526.

Dawson, S., Jahanian, F., Mitton, T., and Tung, T.-L. (1996). Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Symposium on Fault-Tolerant Computing*, pages 404–414.

Ejlali, A., Miremadi, S., Zarandi, H., Asadi, G., and Sarmadi, S. (2003). A Hybrid Fault Injection Approach Based on Simulation and Emulation Co-operation. In *Proceedings of the IEEE/IFIP Internation Conference on Dependable Systems and Networks (DSN-2003)*, pages 479–488.

Goswami, K. (1997). Depend: A Simulation-Based Environment for System Level Dependability Analysis. *IEEE Transactions on Computers*, 46(1):60–74.

H. H. Muhammad, G. B. Bedin, G. Facchini, and M. P. Barcellos (2004). Quebrando a barreira entre simulação e experimentação prática em redes de computadores. In *XXII Simpósio Brasileiro de Redes de Computadores, SBRC 2004*, volume 1, pages 87–100, Gramado, Brasil. SBC.

Jenn, E., Arlat, J., Rimen, M., Ohlsson, J., and Karlsson, J. (1994). Fault Injection into VHDL Models: The MEFISTO Tool. In *Proceedings of the 24th International Symposium on Fault Tolerant Computing, (FTCS-24), IEEE, Austin, Texas, USA*, pages 66–75.

Little, M. C. (2004). *JavaSim User's Guide 1.0*. University of Newcastle upon Tyne. htt://javasim.ncl.ac.uk.

Muhammad, H. H. and Barcellos, M. P. (2002). Simulating Group Communication Protocols Through an Object-Oriented Framework. In *The 35th Annual Simulation Symposium (SS2002)*, pages 143–150. IEEE Computer Society.

Sieh, V., Tschache, O., and Balbach, F. (1997). VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions. In *Symposium on Fault-Tolerant Computing*, pages 32–36.

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.

Veríssimo, P. and Rodrigues, L. (2001). *Distributed Systems for System Architects*. Kluwer Academic Publishers.