Injeção de Falhas de Comunicação em Aplicações Java Multiprotocolo

Cristina Ciprandi Menegotto[1], **Taisy Silva Weber**[1]

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

{ccmenegotto,taisy}@inf.ufrgs.br

Abstract. *Some networked applications are based on more than one communication protocol, such as UDP, TCP and RMI, and must be carefully tested under communication faults. If the emulation of a fault that affects message exchanging does not take into account all simultaneously used protocols, the behavior emulated in an experiment can be different from that observed under real fault occurrence. This paper presents* Comform*, a communication fault injector for multi-protocol Java applications. It works at JVM level, intercepting protocol messages, and, in some cases, it also operates at the operating system level, using firewall rules. The approach is useful for both white box and black box testing and preserves the target application's source code.*

Resumo. *Algumas aplicações de rede são baseadas em mais de um protocolo de comunicação, como UDP, TCP e RMI e devem ser testadas cuidadosamente em presença de falhas de comunicação. Caso a emulação de uma falha que afete a troca de mensagens não considere todos os protocolos simultaneamente utilizados, o comportamento emulado poderá diferir do observado na ocorrência de uma falha real. Este artigo apresenta* Comform*, um injetor de falhas de comunicação para aplicações Java multiprotocolo que opera no nível da JVM, interceptando mensagens de protocolos, e, em alguns casos, opera também no nível do sistema operacional, usando regras de* firewall*. A abordagem é útil para testes de caixa branca e preta e preserva o código fonte da aplicação alvo.*

1. Introdução

Aplicações de rede com altos requisitos de dependabilidade devem ser testadas cuidadosamente em condições de falhas de comunicação para aumentar a confiança no seu comportamento apropriado na ocorrência de falhas. Injeção de falhas de comunicação é a técnica mais adequada para o teste dos mecanismos de tolerância a falhas destas aplicações. Ela é útil tanto para auxiliar na remoção de falhas como na previsão de falhas.

Dentre as aplicações de rede, algumas são baseadas em mais de um protocolo, como UDP, TCP e RMI. Elas são denominadas multiprotocolo no contexto desse trabalho, que foca naquelas escritas em Java e baseadas em protocolos que estão acima do nível de rede na arquitetura TCP/IP. Aplicações multiprotocolo são relativamente comuns, pois diferentes protocolos de comunicação podem ser empregados para diferentes propósitos em uma mesma aplicação de rede. Na literatura, alguns exemplos de aplicações Java multiprotocolo tolerantes a falhas são Zorilla [Drost et al. 2006], Anubis [Murray 2005] e algumas aplicações do *middleware* para comunicação de grupo JGroups [Ban 2002].

Tais exemplos fazem uso simultâneo dos protocolos, ou seja, podem usar mais de um em uma única execução.

Um injetor de falhas de comunicação adequado, que trate todos os protocolos utilizados, é necessário para o seu teste. Caso a emulação de uma falha que afete a troca de mensagens não leve em consideração todos os protocolos simultaneamente utilizados, o comportamento emulado durante um experimento poderá ser diferente daquele observado na ocorrência de uma falha real, de modo que podem ser obtidos resultados inconsistentes sobre o comportamento da aplicação alvo em presença da falha.

Muitos injetores de falhas de comunicação não são capazes de testar aplicações Java multiprotocolo. Outros possuem potencial para o teste dessas aplicações, mas impõem grandes dificuldades aos engenheiros de testes. Por exemplo, contrariamente ao enfoque deste trabalho, algumas ferramentas são voltadas à injeção de falhas de comunicação para o teste de protocolos de comunicação, e não de aplicações. Tal orientação ao teste de protocolos costuma levar a grandes dificuldades no *teste de caixa branca* de aplicações. Testes de caixa branca, também chamados de estruturais, levam em consideração o código fonte da aplicação alvo [Pezzé and Young 2008]. Entre outros exemplos de dificuldades proporcionadas por ferramentas da literatura estão a incapacidade de testar diretamente aplicações Java e a limitação quanto aos tipos de falhas que permitem emular. A análise de tais ferramentas motiva o desenvolvimento de uma solução voltada especificamente ao teste de aplicações multiprotocolo desenvolvidas em Java.

Este artigo apresenta uma solução para injeção de falhas de comunicação em aplicações Java multiprotocolo. A solução opera no nível da JVM, interceptando mensagens de protocolos, e, em alguns casos, opera também no nível do sistema operacional, usando regras de *firewall* para emulação de alguns tipos de falhas que não podem ser emulados somente no nível da JVM. Ela é útil para testes de caixa branca e preta e possui características importantes como a preservação do código fonte da aplicação alvo. A viabilidade da solução proposta é mostrada por meio do desenvolvimento de Comform (COMmunication Fault injector ORiented to Multi-protocol Java applications), um protótipo para injeção de falhas de comunicação em aplicações Java multiprotocolo que atualmente pode ser aplicado para testar aplicações Java baseadas em qualquer combinação dos protocolos UDP, TCP e RMI (incluindo as baseadas em único protocolo).

A seção 2 trata do problema da injeção de falhas de comunicação em aplicações Java multiprotocolo, define requisitos para uma solução, analisa o potencial de ferramentas da literatura para tratar deste problema e compara Comform aos trabalhos relacionados. A seção 3 apresenta um modelo genérico de solução, enquanto a seção 4 apresenta o injetor de falhas Comform. A seção 5 apresenta a condução de experimentos com Comform. Por fim, a seção 6 conclui o artigo.

2. Injeção de Falhas em Aplicações Java Multiprotocolo

Para avaliar o comportamento de uma aplicação Java multiprotocolo em presença de falhas de comunicação usando injeção de falhas, um injetor de falhas de comunicação adequado é necessário. É inconsistente testar uma aplicação em que há uso simultâneo de protocolos com um injetor de falhas voltado ao teste de aplicações Java baseadas só em UDP, como FIONA [Jacques-Silva et al. 2006], e depois testá-la com um injetor voltado ao teste de aplicações Java baseadas em TCP, como FIERCE [Gerchman and Weber 2006].

Para mostrar a inconsistência, Zorilla[Drost et al. 2006], um *middleware par-a-par*, implementado em Java, que visa à execução de aplicações de supercomputação em grades, pode ser considerado como alvo. Seu projeto é baseado em uma rede de nodos, todos capazes de tratar submissão, escalonamento e execução de *jobs* e armazenamento de arquivos. Um nodo faz *broadcast* de pacotes UDP periodicamente na rede local para procurar por outros nodos. O endereço de um ou mais nodos existentes da rede é necessário para se unir a ela. Usando TCP, nodos se conectam diretamente um a outro para envio de grandes quantidades de dados e também é permitido que programas se conectem a Zorilla para, por exemplo, submeter um *job*.

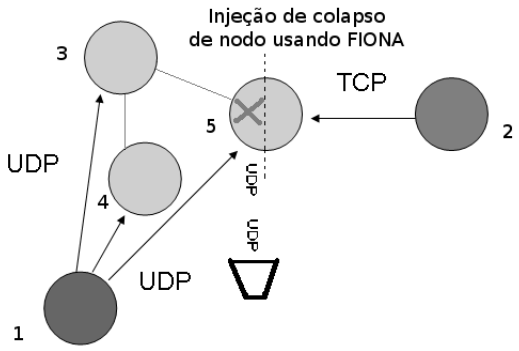

Figura 1. Inadequação de injeção de colapso em nodo Zorilla usando FIONA.

A Figura 1 exemplifica uma tentativa de injeção de colapso de nodo em Zorilla usando FIONA, que elimina a comunicação UDP entre o nodo onde o injetor é executado e os demais. Colapso de nodo refere-se à situação em que um computador pára sua execução e entra em colapso, sem reinicialização. Os nodos 3, 4 e 5 compõem uma rede Zorilla. O nodo 1 está fazendo *broadcast* de pacotes UDP com o objetivo de se unir à rede. O nodo 2 representa um programa externo conectando-se a Zorilla para submissão de um *job*. A injeção de um colapso no nodo 5 usando FIONA resultará em um teste inconsistente no qual o colapso não é percebido pelo nodo 2, pois só a comunicação UDP é suprimida. Problemas similares ocorrem usando um injetor voltado ao teste de aplicações baseadas só em TCP, como FIERCE.

2.1. Requisitos Identificados para a Solução

Além de ser capaz de injetar falhas em aplicações Java multiprotocolo, alguns outros requisitos importantes foram identificados para a solução. Tais requisitos, que ajudam a minimizar as dificuldades enfrentadas por um engenheiro de testes, são listados abaixo:

- Deve ser capaz de emular os tipos de falhas de comunicação que podem ocorrer comumente em ambientes de rede, como, por exemplo, colapsos de nodos, colapsos de *links*, falhas de temporização, omissão e particionamento de rede. Caso contrário, a cobertura dos testes pode ser pobre em muitos casos, como no teste de aplicações que possuem altos requisitos de dependabilidade.
- Deve preservar o código fonte da aplicação alvo, pois ele nem sempre está disponível e, mesmo se disponível, modificá-lo leva à grande intrusividade espacial.
- Deve ser capaz de injetar falhas tanto independentemente do conhecimento do código fonte da aplicação alvo (para testes de caixa preta) como também levando em consideração o código fonte da aplicação alvo (para testes de caixa branca).

- Deve prover um mecanismo adequado para descrição de cargas de falhas, evitando dificuldades que levem o engenheiro de testes à desistência da condução de certos experimentos.

A seguir, outras ferramentas de injeção de falhas são analisadas e é indicado como elas atendem ou não aos requisitos identificados para a solução.

2.2. Potencial de Injetores para o Teste de Aplicações Java Multiprotocolo

FIRMI [Vacaro 2007] é um injetor de falhas cujas aplicações alvo são as escritas em Java e baseadas em RMI, ou seja, ele também não é voltado ao teste de aplicações multiprotocolo. Embora FIRMI e as outras ferramentas citadas anteriormente não atinjam ao objetivo deste trabalho, elas preenchem alguns dos outros requisitos considerados importantes para uma solução e inspiraram a modelagem e desenvolvimento de Comform. Em especial, FIRMI é útil tanto para testes de caixa preta como de caixa branca.

As diferentes implementações do *framework* FIT [Looker et al. 2004] interceptam mensagens SOAP em sistemas baseados em *Web Services*, não tratando outros tipos de protocolo. Tais abordagens trabalham somente em um alto nível de abstração, não sendo adequadas aos propósitos deste trabalho.

NFTAPE [Stott et al. 2000] é um *framework* para avaliação de dependabilidade em sistemas distribuídos usando injeção de falhas. Diferente de outras abordagens, ele faz distinção entre *injetores leves* (*lightweight fault injectors* ou LFI) – componentes responsáveis pela injeção da falha – e *gatilhos* (*triggers*) – responsáveis por disparar a injeção de falhas. Para testar aplicações Java multiprotocolo com NFTAPE, seria necessário projetar e implementar corretamente estes componentes, já que não há relatos sobre sua existência na literatura. Em outras palavras, a abordagem de NFTAPE não provê uma solução para injeção de falhas de comunicação em aplicações multiprotocolo e um grande esforço seria necessário para criar componentes adequados.

FAIL-FCI [Hoarau et al. 2007] é um injetor de falhas para aplicações distribuídas. Ele pode ser usado para injeção de falhas em aplicações tanto de modo quantitativo, como qualitativo, e não requer a modificação do código fonte da aplicação alvo. FAIL-FCI tem potencial para o teste de aplicações Java multiprotocolo, mas com a forte desvantagem de que somente é capaz de emular colapsos de processos e suspensão de processos. Em geral, é muito importante testar aplicações de rede com mais tipos de falhas de comunicação, como colapsos de nodos, colapsos de *links* e falhas de omissão.

Loki [Chandra et al. 2004] é um injetor de falhas para sistemas distribuídos que leva em consideração o estado global do sistema para injeção de falhas. Ele tem potencial para o teste de aplicações multiprotocolo, mas leva a diversas dificuldades. Loki requer a modificação do código fonte da aplicação alvo. Ainda, foi implementado em C/C++ e não pode testar diretamente aplicações Java.

FIRMAMENT [Drebes 2005] é um injetor de falhas cujo propósito principal é testar protocolos baseados em IP. Ele é implementado como um módulo do núcleo Linux, de modo que somente sistemas que podem ser executados nesse ambiente podem ser testados com a ferramenta. FIRMAMENT intercepta o envio e recebimento de pacotes IP em um nodo. Cargas de falhas (*faultloads*), especificadas usando uma linguagem de *bytecode*, são capazes de alterar conteúdo de pacotes e retornar a ação final a ser realizada

sobre eles. FIRMAMENT pode ser usado no teste de aplicações Java multiprotocolo, mas somente teste de caixa preta é viável, pois é muito difícil construir *faultloads* nos quais falhas são ativadas em pontos específicos da execução de aplicações (já que o injetor é "cego" à semântica delas). Mesmo para teste de caixa preta de aplicações, pode ser muito difícil escrever *faultloads*, principalmente para aplicações baseadas em protocolos de mais alto nível, como RMI [Vacaro 2007]. Como todas as mensagens enviadas ou recebidas por um nodo são interceptadas, a seleção daquelas específicas de um determinado processo requer um grande esforço para codificação de faultloads. Ferramentas como FIONA e FIRMI, apesar de não serem multiprotocolo, visam ao teste de *aplicações* Java e superam muitas das dificuldades impostas por ferramentas que visam ao teste de *protocolos*.

2.3. Comparação de Comform a Trabalhos Relacionados

A Tabela 1 compara, sumariamente, Comform a Loki [Chandra et al. 2004], FAIL-FCI [Hoarau et al. 2007] e FIRMAMENT [Drebes 2005]. Contrariamente à abordagem de Comform, estas últimas três ferramentas não cumprem todos os requisitos identificados para a solução. Apesar disso, diferente das outras ferramentas apresentadas anteriormente, elas podem potencialmente testar aplicações multiprotocolo. Para as células sobre as quais não foi possível chegar a uma conclusão, foi atribuído o valor "?".

Tabela 1. Comparando Comform a trabalhos relacionados.

	Comform	FAIL-FCI	Loki	FIRMAMENT
Capaz de injetar falhas em aplicações Java	sim	sim	não	sim
Vários tipos de falhas de comunicação	sim	não	?	sim
Preserva código fonte inalterado	sim	sim	não	sim
Teste de caixa branca de aplicações	sim	sim	sim	não
Teste de caixa preta de aplicações	sim	sim	não	sim
Descrição facilitada de faultloads	sim	?	?	não

3. Modelo Genérico

Esta Seção apresenta um modelo genérico de solução para injeção de falhas de comunicação em aplicações multiprotocolo.

3.1. Cuidados para Emulação de Colapso em Aplicações TCP

Protocolos de transporte, como UDP e TCP, são implementados no núcleo do sistema operacional. A emulação de falhas de comunicação em aplicações Java baseadas *somente* em UDP pode ser feita completamente no nível da JVM [Jacques-Silva et al. 2006]. No caso de falhas de colapso de nodo, por exemplo, essa emulação pode ser feita por meio da inibição do envio e recebimento de mensagens UDP no nível da JVM. Isso é suficiente devido à simplicidade do protocolo UDP. Por outro lado, considerando o TCP e aqueles protocolos de mais alto nível que o têm como base, falhas de comunicação que envolvem o descarte de mensagens dificilmente podem ser emuladas somente no nível da JVM.

A Figura 2 explica, de modo simplificado, a inadequação de emular falhas de colapso de nodo (ou outros tipos de falhas caracterizados pelo descarte de mensagens) em aplicações Java baseadas em TCP, e/ou em protocolos baseados em TCP, apenas no nível da JVM. O nodo A está executando uma aplicação Java cliente, baseada em TCP, enquanto o nodo B está executando sua correspondente aplicação servidor. Uma conexão

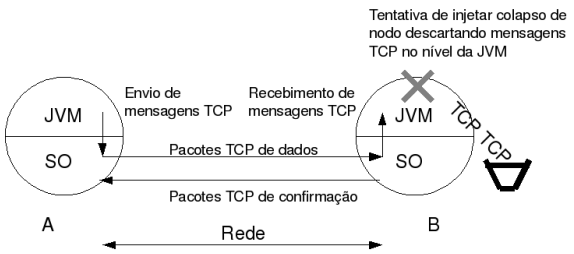

Figura 2. Emulação inconsistente de falhas de colapso no nível da JVM.

já foi estabelecida e tenta-se injetar uma falha de colapso em B por meio do descarte de mensagens TCP no nível da JVM de B. Pode-se observar o cliente enviando mensagens TCP ao servidor. Embora essas mensagens sejam descartadas no nível da JVM do servidor, o seu sistema operacional (SO) ainda irá enviar ao cliente confirmações relacionadas ao recebimento das mensagens, de modo que a falha não é emulada corretamente.

3.2. Seleção e Manipulação de Mensagens

Para que falhas que envolvem o descarte de pacotes sejam emuladas com representatividade em um nodo, pacotes relativos a todos os protocolos utilizados pela aplicação alvo devem ser passíveis de seleção e manipulação antes que sejam entregues à aplicação alvo (no caso de recebimento de pacotes) e antes que sejam enviados a outros nodos (no caso de envio de pacotes). Assim, essa seleção e manipulação deve ser feita no contexto do sistema operacional. Por outro lado, ao contrário de falhas caracterizadas pelo descarte de pacotes, falhas de temporização podem ser injetadas pela seleção e manipulação de mensagens em um nível mais alto de abstração.

Visando atender aos requisitos identificados para a solução, o modelo também requer a interceptação de mensagens em um nível de abstração mais alto do que o nível do sistema operacional de modo que informações específicas relacionadas à aplicação alvo possam ser facilmente recuperadas para a ativação e desativação de falhas. Por exemplo, considerando uma aplicação baseada em RMI, pode ser interessante a um engenheiro de testes ter a possibilidade de ativar uma falha antes da invocação de um determinado método remoto ou depois de um certo número de invocações de métodos remotos. Este tipo de informação é de difícil obtenção no nível do núcleo do sistema operacional, mas pode ser facilmente obtido em um nível mais alto de abstração.

4. Arquitetura de Comform

Esta Seção apresenta a arquitetura básica de Comform. Como descrito na Seção 3, o modelo genérico requer um modo de selecionar e manipular mensagens no nível do sistema operacional de forma a emular corretamente falhas que envolvem o descarte de pacotes (como falhas de colapso de nodos e *links*). A arquitetura usa um componente Firewall para esse fim. Esta abordagem foi aplicada com sucesso no injetor de falhas FIRMI e seu reuso mostrou-se conveniente. O uso de regras de *firewall* para o descarte de pacotes evita problemas como o descrito na Figura 2, pois os pacotes são descartados pelo próprio sistema operacional, antes que eles sejam entregues à aplicação e antes que sejam enviados a outros nodos. Deste modo, os nodos que devem perceber o estado falho do nodo onde a injeção está sendo realizada o farão de modo consistente.

O modelo também requer um modo de interceptar mensagens em um nível de abstração mais alto que o do sistema operacional, tal que informações específicas relacionadas à aplicação alvo possam ser facilmente recuperadas para *ativação* e *desativação* de falhas em testes de caixa branca. Seguindo a abordagem usada em outras ferramentas ([Jacques-Silva et al. 2006], [Vacaro 2007]), classes que implementam os protocolos de interesse – UDP, TCP e RMI – são instrumentadas no nível da JVM. Com a sua instrumentação, é possível obter informações de interesse para ativação e também para emulação de falhas. Para promover interação entre as partes da arquitetura que operam nos níveis da JVM e do sistema operacional, informações de portas locais sendo utilizadas pela aplicação alvo são obtidas por meio da instrumentação e usadas para a construção de regras de *firewall*. Ainda, falhas de temporização podem ser emuladas no nível da JVM.

Os tipos de falhas de comunicação que podem, atualmente, ser emulados por Comform incluem colapso de nodos, colapso de *links* e falhas de temporização. O protótipo pode ser estendido com a inclusão de novos tipos de falhas, como falhas de omissão, mas já possui uma variedade de tipos de falhas mais rica do que a oferecida por ferramentas como FAIL-FCI [Hoarau et al. 2007].

A Figura 3 apresenta uma visão simplificada da arquitetura de Comform. Uma linha tracejada separa o nível da JVM do nível do sistema operacional (SO). O *Firewall* e as implementações dos protocolos TCP e UDP na pilha de protocolos TCP/IP operam no nível do SO. No nível da JVM, a Aplicação Alvo é executada e as classes de comunicação de interesse da API Java são instrumentadas em tempo de carga. A instrumentação dessas classes provê a interação com o Controlador do injetor de falhas. Este componente é responsável pelo controle dos experimentos e interage com os outros componentes do injetor. O Monitor coleta informações importantes sobre o experimento. A Carga de Falhas (*Faultload*) inclui as noções de Carga de Falhas propriamente dita e de Carregador de Carga de Falhas. Em Comform, *Faultloads* são descritos como classes Java e um Carregador de Carga de Falhas é responsável por sua carga. Esses conceitos são reusados de FIRMI (e também foram reusados em um trabalho relacionado [Cézane et al. 2009]). A caixa Falha representa as falhas que a ferramenta é capaz de emular. Por fim, o Filtro de Mensagens é responsável pela efetiva injeção das falhas especificadas por um módulo de Carga de Falhas. A interface *Firewall* representa a interação com um *Firewall* de modo a emular corretamente falhas caracterizadas pelo descarte de mensagens.

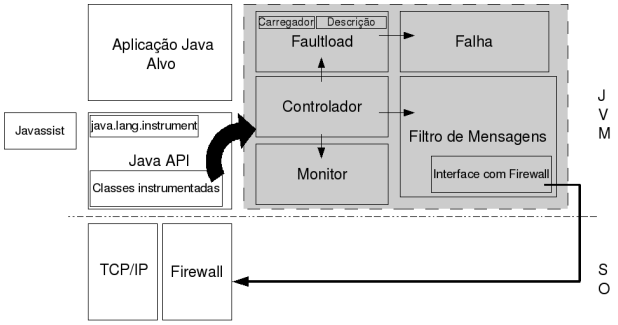

Figura 3. Arquitetura simplificada da ferramenta.

O pacote `java.lang.instrument` [SUN MICROSYSTEMS 2008] é utilizado na arquitetura para a interceptação do carregamento de classes na JVM. Deste modo,

é obtido acesso ao *bytecode* de classes Java que implementam os protocolos de interesse. A adição de código especial nessas classes, ou seja, a instrumentação dessas classes, é necessária para que elas interajam com o Controlador do injetor de falhas quando seus objetos forem invocados pela aplicação alvo. Apesar de prover acesso às classes de interesse e possibilitar adição de *bytecodes*, o pacote não é capaz de realizar a instrumentação de código propriamente dita. Deste modo, uma biblioteca especializada em instrumentação de *bytecodes* deve ser selecionada para essa tarefa. Javassist [Chiba 1998] foi escolhida para esse propósito. Como ela não faz parte da API de Java, aparece representada na Figura 3 como uma caixa separada.

O funcionamento básico do injetor pode ser resumido da seguinte forma:

1. Uma instância do Controlador (classe `Controller`) é obtida e é feita instrumentação das classes de interesse dos protocolos alvo durante o seu carregamento. No contexto da criação de uma instância de `Controller`, é feito o carregamento do *Faultload* a ser utilizado no experimento (classe derivada da classe `Faultload` desenvolvida pelo usuário) e execução de seu método construtor. O carregamento do *Faultload* é feito por um Carregador de Módulos de Carga de Falhas (classe `BaseFaultloadLoader`).
2. Quando é realizada a associação de um `socket` a uma porta local no nodo o injetor está sendo executado, a porta local é registrada no Filtro de Mensagens (classe `MessageFilter`).
3. Quando uma mensagem dos protocolos de interesse é interceptada, ela é processada pela classe `Controller`, que aciona a coleta de dados pelo Monitor (classe `Monitor`), registra a mensagem em *log*, invoca o método `update` do `Faultload` para que sejam realizadas possíveis decisões sobre ativação de falhas (podendo considerar o conteúdo da mensagem) e, finalmente, injeta as possíveis falhas ativadas no passo anterior por meio da classe `MessageFilter`.

Falhas podem ser ativadas durante a carga do injetor de falhas, ficando ativas desde o início do experimento, ou mais adiante, durante a execução da aplicação alvo. Neste caso, a ativação pode levar em consideração o conteúdo e a quantidade de mensagens interceptadas. Os atributos de mensagens dos protocolos de interesse podem ser usados para propósitos de ativação de falhas. A Figura 4 apresenta a modelagem das mensagens dos protocolos de interesse. Mensagens UDP e TCP possuem como atributos o tipo de mensagem, os dados sendo enviados ou recebidos, o tamanho da mensagem, o endereço de rede do nodo remoto e a porta do nodo remoto. Uma requisição RMI é representada pela classe `RMIRequest` e possui como atributos principais o tipo de requisição, o endereço de rede do nodo remoto, a referência remota usada, o método que está sendo invocado e os valores dos parâmetros deste método. É possível construir um Módulo de Carga de Falhas que ative falhas com base, por exemplo, no nome de um método sendo invocado.

Os atributos correspondentes a tipos de mensagem UDP e TCP ou tipo de requisição RMI podem assumir vários valores. A Tabela 2 mostra esses valores para TCP, considerando o pacote `java.nio`, e para RMI (os valores para TCP no pacote `java.net` e para UDP em ambos os pacotes seguem lógica semelhante). Para cada método relacionado a envio ou recebimento de mensagens (ou estabelecimento de conexão), existe a possibilidade de realizar ativação de falhas antes de sua execução (ao início da execução do método) e após sua execução (logo antes do método encerrar

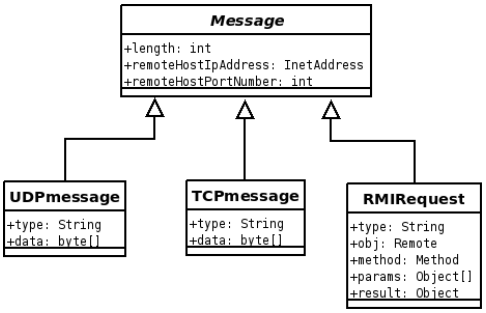

Figura 4. Modelagem de mensagens dos protocolos de interesse.

sua execução e retornar). Por exemplo, é possível realizar ativação de falhas antes da execução de um método remoto por um servidor ("RMI_beforeExecuting") ou logo antes da execução do método remoto retornar ("RMI_afterExecuting"). Também é possível ativar falhas com base em informações coletadas pelo Monitor, como o total de *bytes* enviados ou recebidos ou o total de requisições RMI efetuadas.

Tabela 2. Valores para o atributo tipo (`type`) de mensagens TCP (java.nio) e RMI.

Métodos TCP instrumentados (java.nio)	**Valores para atributo tipo**
connect	"TCPNio_connect_before", "TCPNio_connect_after"
long write	"TCPNio_writeLong_before", "TCPNio_writeLong_after"
int write	"TCPNio_writeInt_before", "TCPNio_writeInt_after"
long read	"TCPNio_readLong_before", "TCPNio_readLong_after"
int read	"TCPNio_readInt_before", "TCPNio_readInt_after"
accept	"TCPNio_accept_before", "TCPNio_accept_after"
Métodos RMI instrumentados	**Valores para atributo tipo**
invoke (servidor)	"RMI_beforeExecuting", "RMI_afterExecuting"
invoke (cliente)	"RMI_beforeInvoking", "RMI_afterInvoking"

5. Experimentos de Injeção de Falhas usando Comform

5.1. Experimento com Zorilla

Este experimento tem como propósito demonstrar que Comform pode, de fato, injetar adequadamente falhas de colapso em Zorilla [Drost et al. 2006], uma aplicação multi-protocolo baseada em UDP e TCP. A versão *1.0-beta1* de Zorilla, disponível para *download* [VRIJE UNIVERSITEIT 2007], foi utilizada. Interceptando o carregamento de classes dessa aplicação, constatou-se que ela é baseada nos protocolos TCP e UDP e faz uso tanto das implementações do pacote java.net como do pacote java.nio.

A situação emulada corresponde à apresentada na Figura 5, que indica qual máquina representa cada nodo no experimento. Na Figura, os nodos `dkw`, `jaguar` e `maverick` constituem uma rede Zorilla. O nodo `dkw` sofre uma emulação de colapso, com o uso de Comform, quando o nodo `mercedes` tenta conectar-se a `dkw` para realizar a submissão de um *job*. A seguir, o nodo `grantorino` tenta se unir à rede. Para a emulação correta do colapso, este deve ser percebido por todos os demais nodos do sistema. Como será mostrado, ao contrário de injetores de falhas voltados a aplicações baseadas em um único protocolo, Comform atinge esse objetivo, já que descarta tanto mensagens UDP como TCP no nodo `dkw` a partir do momento de ativação da falha.

Inicialmente, foi formada uma rede Zorilla, constituída pelos nodos `dkw`, `jaguar` e `maverick`. Para a inicialização de Zorilla nos nodos `jaguar` e `maverick`,

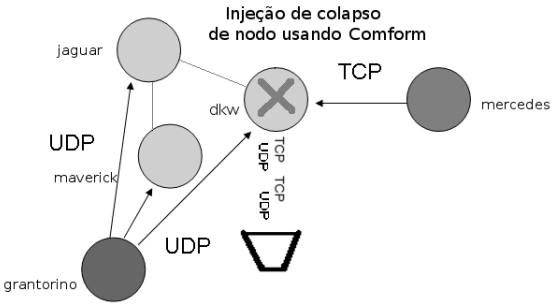

Figura 5. Planejamento de injeção de colapso em nodo Zorilla usando Comform.

foi utilizado, em cada máquina, o *script zorilla*, que é responsável pela inicialização e é encontrado no diretório *bin* de Zorilla *1.0-beta1*. Para a inicialização de Zorilla no nodo `dkw`, que é aquele onde Comform é executado com o propósito de injeção de uma falha de colapso, foi desenvolvido um novo *script* que alia as informações do *script zorilla* às informações necessárias para a instanciação de uma aplicação juntamente com Comform.

O *faultload* desenvolvido para o experimento, apresentado na Figura 6, ativa um colapso de nodo em `dkw` quando da tentativa de conexão feita por `mercedes` visando submeter um *job*. A falha de colapso é ativada no escopo do método `update`, que recebe como parâmetro as mensagens relacionadas aos protocolos sendo utilizados pela aplicação alvo. Quando uma mensagem TCP do tipo (atributo *type* de `TCPmessage`) "TCP_implAccept_after" proveniente de `mercedes` é recebida, a falha de colapso é ativada. O tipo "TCP_implAccept_after" foi utilizado porque, ao fim do corpo do método `implAccept`, o endereço do nodo que está tentando a conexão já é conhecido, de modo que é possível ativar a falha exatamente quando da tentativa de conexão por `mercedes`.

```
public class CrashFaultloadImplZorilla extends Faultload {
        CrashFault cf;
        int count = 0;
        public CrashFaultloadImplZorilla() throws Exception {
                cf = new CrashFault();
        }
        public void update(Message msg) throws Exception {
                if (msg.getClass().getName().equals("faultinjector.TCPmessage"))
                        if (((TCPmessage)msg).type.equals("TCP_implAccept_after") &&
                        (((TCPmessage)msg).remoteHostIpAddress).toString().equals("
                            /143.54.10.157")){
                                cf.activate();
        }
}
```

Figura 6. *Faultload* para emulação de colapso de nodo em `dkw`.

Depois de formada a rede Zorilla, uma tentativa de submissão de *job* no nodo `dkw` é realizada pelo nodo `mercedes` com o uso do *script zubmit*, responsável pela submissão de *jobs* a um nodo Zorilla (encontrado no diretório *bin* de Zorilla *1.0-beta1*). Para a tentativa de submissão, o nodo `mercedes` procura, primeiramente, estabelecer uma conexão com o nodo `dkw`. Porém, a falha de colapso é ativada em `dkw` quando desta tentativa e a exceção apresentada na Figura 7 é gerada em `mercedes`.

Logo após a ativação da falha, o nodo `grantorino` tenta se unir à rede. Para

```
ccmenegotto@mercedes:~/Desktop/apsteste/zorilla-1.0-beta1/bin$ ./zubmit -na
    143.54.10.205:5444 -c 1 .. ../../satin-2.1/examples/lib/satin-examples.jar
exception on running job: java.net.ConnectException: Connection setup failed
java.net.ConnectException: Connection setup failed
        at ibis.smartsockets.direct.DirectSocketFactory.createSingleSocket(
            DirectSocketFactory.java:1360)
        at ibis.smartsockets.direct.DirectSocketFactory.createSocket(DirectSocketFactory
            .java:1432)
        at ibis.smartsockets.direct.DirectSocketFactory.createSocket(DirectSocketFactory
            .java:1300)
        at ibis.zorilla.zoni.ZoniConnection.<init>(ZoniConnection.java:50)
        at ibis.zorilla.apps.Zubmit.main(Zubmit.java:278)
```

Figura 7. Exceção gerada no nodo `mercedes`.

tal, é executado o *script zorilla*. Inicialmente, o nodo apresenta um comportamento de inicialização convencional. A seguir, ele consegue se comunicar com os nodos `jaguar` e `maverick`, mas gera exceções ao tentar se comunicar com o nodo `dkw`, onde o colapso está sendo emulado. Os nodos `jaguar` e `maverick` também percebem o colapso de `dkw` e exceções do tipo `java.net.SocketTimeoutException` foram registradas em seus *logs* a cada tentativa de comunicação com `dkw`.

O experimento também foi realizado manualmente visando à comparação dos resultados com os obtidos na emulação com Comform. Para isso, a instrumentação do método `implAccept` foi alterada tal que, quando do recebimento de uma mensagem do tipo "TCP_implAccept_after" proveniente de `mercedes`, fosse executado um laço infinito. Esse laço infinito forneceu tempo necessário para remoção do cabo de força de `dkw` no contexto da execução do método `implAccept`, logo após a entrada do comando *zubmit* em um terminal de `mercedes`. A exceção obtida em `mercedes` foi igual à obtida na realização do experimento com Comform. Quanto a `jaguar` e `maverick`, o resultado obtido foi semelhante, mas, no lugar de exceções `java.net.SocketTimeoutException`, que haviam sido obtidas no experimento usando Comform, os experimentos manuais resultaram, em sua maioria, em algumas (de zero a 3) exceções `java.net.SocketTimeoutException` seguidas de exceções `java.net.NoRouteToHostException`. Já em `grantorino`, onde Zorilla foi instanciado somente após a ativação da falha, só foram registradas exceções `java.net.NoRouteToHostException` nos experimentos manuais. Porém, apesar do nome diferente dessa nova exceção registrada nos experimentos manuais, observou-se que todas as demais informações, referentes à origem da exceção, são iguais às obtidas no experimento com Comform, de modo que o colapso é emulado com precisão adequada. Uma investigação, baseada em outro experimento, no qual o colapso em `dkw` foi emulado utilizando-se regras do IPTables responsáveis pelo descarte de todos os pacotes recebidos e enviados sem que isso resultasse na geração desse tipo de exceção em `mercedes`, levou à conclusão de que a geração de exceções `NoRouteToHostException` só seria viável emulando-se falhas em nível de abstração mais baixo.

A análise dos *logs* do experimento com Comform mostrou que as seguintes classes utilizadas por Zorilla para comunicação UDP ou TCP foram instrumentadas no nodo `dkw`: `ServerSocket`, `DatagramSocket`, `Socket`, `SocketInputStream` e `SocketOutputStream` de `java.net` e `DatagramChannelImpl` de `sun.nio.ch`. Todos os nodos envolvidos no experimento perceberam o colapso

de dkw e a falha foi emulada consistentemente. O mesmo não ocorreria caso um injetor de falhas voltado ao teste de aplicações Java baseadas em um único protocolo tivesse sido utilizado. Não foram encontrados registros de injetores de falhas que tratem da instrumentação do pacote java.nio, de modo que Comform é pioneiro nesse sentido.

5.2. Experimento com Aplicação RMI

Este experimento mostra a habilidade de Comform para injetar falhas em testes de caixa branca (considerando o código fonte da aplicação alvo), além da correção da estratégia empregada para emulação de colapsos de nodo. Ele foi conduzido em dois computadores, mercedes e dkw. A aplicação alvo inclui um servidor que implementa um interface remota composta de dois métodos. O primeiro, multiply, multiplica duas matrizes recebidas como parâmetro. O segundo, sum, soma duas matrizes recebidas como parâmetro. A aplicação também inclui um cliente que obtém uma referência remota, s, ao servidor visando ser capaz de invocar métodos no servidor. Três matrizes – a, b e c – são declaradas no cliente. A Figura 8 mostra a ordem de invocação dos métodos remotos.

```
int [][] d = s.multiply(a, b);
d = s.multiply(a, c);
d = s.sum(a, b);
```

Figura 8. Um trecho de código do cliente.

O servidor foi executado em dkw, com Comform, e o cliente em mercedes. Para ilustrar a habilidade de Comform para injetar falhas em pontos específicos da execução de uma aplicação, um colapso de nodo foi injetado em dkw antes da segunda invocação de multiply, ou seja, foi emulado o colapso do servidor depois que o cliente invocou o método, mas antes que o servidor retornasse o resultado. Para comparação, o experimento também foi realizado manualmente pela substituição do código de multiply por um laço infinito, que proveu tempo suficiente para consequente remoção do cabo de força do nodo servidor enquanto ele executava o laço. A Figura 9 mostra o *faultload* desenvolvido para esse teste. A classe CrashFaultload estende a classe Faultload da API de Comform. A falha é ativada no escopo do método update, que recebe como parâmetro mensagens relacionadas aos protocolos em uso.

```
public class CrashFaultload extends Faultload {
  private CrashFault cf;
  int count = 0;
  public CrashFaultload() throws Exception {
    cf = new CrashFault();}
  public void update(Message msg) throws Exception {
    if (msg.getClass().getName().equals("faultinjector.RMIRequest")){
      if (((RMIRequest)msg).method.getName().equals("multiply") && ((RMIRequest) msg).
          type.equals("RMI_beforeExecuting")) {
        count++;
        if (count==2) cf.activate();}}}
}
```

Figura 9. *Faultload* para emulação de colapso.

A situação emulada deixa o cliente esperando que o servidor processe a requisição. Como a aplicação não é tolerante a falhas, quando um colapso é injetado, o cliente não é

capaz de saber se o servidor ainda está processando a requisição ou se ele sofreu colapso. Após a expiração do *timeout* do TCP, que pode levar um longo tempo dependendo da configuração do cliente, uma exceção é disparada na aplicação cliente indicando que a conexão sofreu *timeout*. Adicionalmente, exatamente a mesma exceção foi capturada tanto no experimento manual como no realizado com Comform, mostrando a adequação da abordagem usada para emulação de colapsos de nodo. A Figura 10 mostra um trecho da mensagem de exceção. A realização de um experimento como esse seria inviável com a utilização de um injetor de falhas voltado ao teste de protocolos como FIRMAMENT.

```
Exception in thread "main" java.rmi.UnmarshalException: Error unmarshaling return header
    ; nested exception is:
    java.net.SocketException: Connection timed out
    at sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:209)
    at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:142)
    at java.rmi.server.RemoteObjectInvocationHandler.invokeRemoteMethod(
        RemoteObjectInvocationHandler.java:178)
    at java.rmi.server.RemoteObjectInvocationHandler.invoke(
        RemoteObjectInvocationHandler.java:132)
    at $Proxy0.multiply(Unknown Source)
    at Client.main(Client.java:47)
Caused by: java.net.SocketException: Connection timed out
```

Figura 10. Exceção gerada no cliente.

6. Considerações Finais

Este artigo mostrou que muitos injetores de falhas de comunicação encontrados na literatura não são capazes de testar aplicações Java multiprotocolo e que outros possuem potencial para o teste dessas aplicações, mas impõem grandes dificuldades aos engenheiros de testes. Com base na necessidade identificada, apresentou a proposta de Comform. Quanto à capacidade de emular os tipos de falhas de comunicação que podem ocorrer comumente em ambientes de rede, Comform já é superior a ferramentas como FAIL-FCI [Hoarau et al. 2007], mas ainda pode ser melhorado pela inclusão de novos tipos de falhas em sua arquitetura. Quanto à preservação do código fonte da aplicação alvo, ela é atingida com o uso da combinação do pacote `java.lang.instrument` e de Javassist. A capacidade de injetar falhas tanto independentemente do conhecimento do código fonte da aplicação alvo como também o levando em consideração foi mostrada nos experimentos. Quanto ao provimento de um mecanismo adequado para descrição de Módulos de Carga de Falhas, a estratégia empregada não pôde ser tratada com maiores detalhes neste artigo, mas os experimentos mostraram sua facilidade de uso.

Trabalhos futuros incluem a continuação do desenvolvimento de Comform visando tratar mais tipos de falhas e protocolos. Seria interessante implementar falhas de *omissão* e investigar alternativas para emulação de particionamento de rede. Quanto a outros protocolos, seria útil tratar outros largamente utilizados que sejam baseados em UDP ou TCP. Comform já tem essa capacidade para o caso de *testes de caixa preta*. Para *testes de caixa branca*, ele pode ser estendido para prover facilidades para ativação de falhas como as que já oferece para teste de caixa branca de aplicações Java que usam RMI.

Referências

Ban, B. (2002). JGroups - a toolkit for reliable multicast communication. 2002. Disponível em: <http://www.jgroups.org>. Acesso em: jan. 2009.

Cézane, D. et al. (2009). Um injetor de falhas para a avaliação de aplicações distribuídas baseadas no commune. In *Anais do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, SBRC, 27.*, volume 1, pages 901–914, Recife. SBC.

Chandra, R. et al. (2004). A global-state-triggered fault injector for distributed system evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605.

Chiba, S. (1998). Javassist. 1998. Disponível em: <http://www.csg.is.titech.ac.jp/~chiba/javassist/>. Acesso em: jan. 2009.

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

Drost, N., van Nieuwpoort, R. V., and Bal, H. (2006). Simple locality-aware co-allocation in peer-to-peer supercomputing. In *Proc. of the 6th International Symposium on Cluster Computing and the Grid Workshops, CCGRIDW*, volume 2, Singapore. IEEE.

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 *Anais do Workshop de testes e tolerância a falhas, WTF, 7.*, volume 1, pages 41–54, Curitiba. SBC.

Hoarau, W., Tixeuil, S., and Vauchelles, F. (2007). FAIL-FCI: Versatile fault injection. *Future Generation Computer Systems*, 23(7):913–919.

Jacques-Silva, G. et al. (2006). A network-level distributed fault injector for experimental validation of dependable distributed systems. In *Proc. of the 30th Int. Computer Software and Applications Conference, COMPSAC*, pages 421–428, Chicago. IEEE.

Looker, N., Munro, M., and Xu, J. (2004). Ws-fit: a tool for dependability analysis of web services. In *Proc. of the 28th Annual International Computer Software and Applications Conference, COMPSAC*, volume 2, pages 120–123, Hong Kong. IEEE.

Murray, P. (2005). A distributed state monitoring service for adaptive application management. In *Proc. of the International Conference on Dependable Systems and Networks, DSN*, pages 200–205, Yokohama. IEEE.

Pezzé, M. and Young, M. (2008). *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons.

Stott, D. T. et al. (2000). Nftape: A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proc. of the 4th International Computer Performance and Dependability Symposium, IPDS*, pages 91–100, Chicago. IEEE.

SUN MICROSYSTEMS (2008). Java platform standard ed. 6. 2008. Disponível em: <http://java.sun.com/javase/6/docs/api/>. Acesso em: jan. 2009.

Vacaro, J. C. (2007). Avaliação de dependabilidade de aplicações distribuídas baseadas em rmi através de injeção de falhas. 2007. 89 f. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, UFRGS, Porto Alegre.

VRIJE UNIVERSITEIT (2007). Ibis: Grids as promised. 2007. Disponível em: <http://www.cs.vu.nl/ibis/downloads.html>. Acesso em: out. 2009.