Um Injetor de Falhas para a Avaliação de Aplicações Distribuídas Baseadas no *Commune*

Dalton Cézane, Diego Renato, Miguel Queiroga, Sabrina Souto, Marco A. Spohn

Departamento de Sistemas e Computação

Universidade Federal de Campina Grande

Campina Grande – PB, Brasil

e-mail:{cezane|renato|miguel|sabrina|maspohn}@dsc.ufcg.edu.br

Abstract. *Distributed systems use communication protocols for exchanging messages. The validation of such systems takes into account fault tolerance strategies, which must be carefully tested. The observation of the behavior of the application under fault conditions can help refining the fault tolerance strategies themselves, anticipating the application behavior in real scenarios. This article presents a fault injector tool to evaluate distributed applications based on the Commune middleware which is implemented on top of the XMPP protocol. Communication faults are injected through the Commune interface with the addition of the fault injector in the middleware protocol stack, in such a way that it remains transparent to the user. This paper also describes the techniques employed in the fault injector development, and the fault injector functionality is demonstrated through an experiment.*

Resumo. *Sistemas distribuídos têm se utilizado de protocolos de comunicação para troca de mensagens. A validação destes sistemas envolve estratégias de tolerância a falhas, as quais devem ser criteriosamente testadas. A observação do comportamento da aplicação sob falhas permite refinar as estratégias de tolerância a falhas implementadas e antecipar a maneira como a aplicação reage as mesmas em uma situação real. Este artigo apresenta um injetor de falhas para avaliação de aplicações distribuídas implementadas sob o Commune, um middleware que dá suporte ao uso do protocolo XMPP. Falhas de comunicação são injetadas através da instrumentação do Commune, acrescentando o injetor de falhas na pilha de protocolos do middleware, de modo transparente ao usuário. O artigo ainda descreve as técnicas empregadas no desenvolvimento e demonstra as funcionalidades do injetor através da condução de um experimento.*

1. Introdução

O desenvolvimento de sistemas distribuídos, além de outras técnicas como invocação remota a métodos, por exemplo, têm se utilizado de protocolos de comunicação para troca de mensagens. Na validação destes sistemas, o engenheiro de testes se defronta com o desafio de, além de avaliar as funcionalidades de aplicações em resposta a entradas

especificadas e situações esperadas, incluir testes em situações de erro. Estes últimos permitem verificar a robustez, disponibilidade e confiabilidade do sistema. Entretanto, o teste de aplicações distribuídas sob situações de erro não é uma tarefa trivial, pois falhas nesse meio podem provocar inconsistências ou comportamento imprevisível da aplicação. Neste caso, o desafio é não apenas construir, mas também testar mecanismos de tolerância a falhas que considerem a possibilidade de ocorrência de problemas de comunicação.

Uma das primeiras etapas de um mecanismo de tolerância à falhas é a detecção dos erros causados por falhas. Essas falhas são aleatórias devido à fadiga dos componentes, imperfeição de manufatura e eventos externos, como aquecimento, e até mesmo radiação eletromagnética. Essa capacidade de detecção de erros causados por falhas deve ser testada antes que o sistema entre em produção para que se conheça a cobertura (coverage) do mecanismo e seu grau de confiança possa ser inferido. Injeção de falhas é um método muito usado por ser eficiente e simples. Os resultados alcançados complementam demais métodos, como simulações ou verificações formais. A injeção de falhas, que interessa a este trabalho, é realizada por software. Esta, por sua vez, usa código inserido na aplicação de modo a emular a falha.

Este trabalho se baseia nas idéias e no código do FIRMI [Vacaro, Weber 2006], e tem como proposta apresentar um injetor de falhas de comunicação voltado à validação de aplicações distribuídas desenvolvidas utilizando o protocolo de comunicação XMPP [XMPP 2008], mais especificamente o Commune [Commune 2008], que é um middleware que dá suporte ao uso de tal protocolo. O foco deste injetor é complementar o teste e a validação dos mecanismos de comunicação usados pela aplicação, baseados no Commune.

Este trabalho está estruturado da seguinte forma: a Seção 2 apresenta uma descrição geral sobre o Commune e o protocolo XMPP; a Seção 3 apresenta o conceito de injeção de falhas; na Seção 4 descrevemos detalhadamente a abordagem adotada para o desenvolvimento do injetor, como foi o desenvolvimento e como pode ser utilizado; na Seção 5 é demonstrado o uso da ferramenta através de um experimento com uma aplicação de troca de mensagens; a Seção 6 discorre sobre os trabalhos relacionados, e finalmente na Seção 7 apresenta-se a conclusão.

2. Commune

O OurGrid [Cirne, Brasileiro et al. 2006] é uma grade computacional entre-pares voltada à execução de aplicações *bag-of-tasks* (BoT). As primeiras versões do OurGrid utilizavam Java RMI para comunicação com objetos remotos, porém essa alternativa exigia o gerenciamento de muitas *threads*. Isso ocorre porque quando uma máquina cliente necessita de um serviço, mantém-se uma *thread* bloqueada até o recebimento da resposta do servidor. Caso não se queira ter perda de desempenho com o bloqueio, é preciso construir uma aplicação *multithread*, ampliando a complexidade de programação.

Para contornar as principais deficiências do RMI, nas versões subseqüentes do *OurGrid*, desenvolveu-se uma solução de comunicação assíncrona, o JIC, baseada no protocolo *Extensible Messaging and Presence Protocol* [XMPP 2008] (XMPP). O XMPP é um protocolo aberto para a troca de mensagens e de notificação de presença, menos vulnerável as restrições de comunicação impostas por *firewalls* e NAT [Cirne, Brasileiro, Fireman 2006].

O Commune [Commune 2008] representa uma evolução do JIC, deixando ainda mais transparente os detalhes da comunicação, diminuindo o esforço necessário para criar uma aplicação baseada neste *middleware*.

Ao utilizar o Commune, aplica-se a classe `ApplicationServer`, que por sua vez necessita da criação de um Container. O Container é o responsável por gerenciar todas as operações do Commune como, por exemplo, a geração de mensagens, a identificação dos objetos e a criação de *stubs*.

O container (vide Figura 1) contém os seguintes componentes: os *processors*, que servem para processar as mensagens, fazendo operações como invocando métodos em objetos ou transferindo arquivos. Os *objects* são os objetos que são responsáveis por executar algum serviço. São os objetos que realizam as operações vindas de outras máquinas por meio de mensagens. Os *stubs* são objetos que possuem a mesma interface dos objetos remotos. Por exemplo: supondo que haja um computador A que necessita realizar operações em outra máquina B. Então o container localizado em A cria *stubs* dos objetos remotos de B. Os *stubs* então transformam as chamadas de método em A em mensagens que serão enviadas a B. Os *helpers* são úteis para quando *objects* são adicionados ao container.

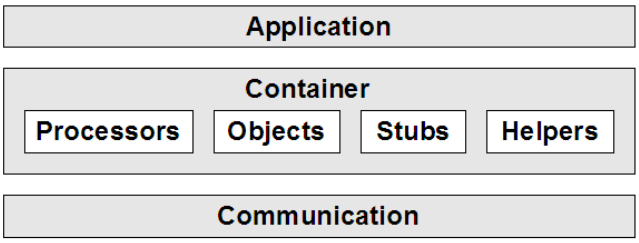


Figura 1: Diagrama simplificado do Commune.

O container também possui referência a um objeto da classe `CommuneNetwork`, que disponibiliza uma pilha de protocolos envolvidos no envio e recebimento de mensagens. Os protocolos são de: apresentação, *loopback*, assinatura, certificação e o XMPP. A adoção do XMPP elimina a necessidade de uma quantidade de portas (i.e., *sockets*) equivalente a quantidade de máquinas remotas, bastando a comunicação direta de todas as entidades comunicantes com o servidor XMPP.

3. Injeção de Falhas

A injeção de falhas é uma técnica eficiente de validação experimental de software. No ambiente do sistema sob teste é emulada a ocorrência de falhas sem comprometer fisicamente o sistema. A injeção de falhas tem como objetivos auxiliar na fase de teste de depuração, estudar o comportamento do sistema em um cenário de falhas, avaliar a cobertura dos mecanismos de detecção e recuperação de falhas e aferir sua eficiência e desempenho [Hsueh, Tsai, Iyer 1997].

Uma técnica para emulação de falha de comunicação consiste em interceptar as mensagens trocadas pela aplicação. A mensagem enviada ou recebida é analisada e é decidido se uma falha deve ser injetada. Exemplos de falhas incluem o descarte e o atraso de pacotes.

Aplicações distribuídas e de rede se caracterizam pela existência de máquinas individuais; no entanto, interconectadas por canais de comunicação. Para oferecer alta disponibilidade, essas aplicações precisam reagir a falhas relacionadas a troca de mensagens entre as máquinas do sistema. Para observar essa reação, aplica-se injeção de falhas de comunicação, simulando problemas no envio e recebimento de mensagens pela rede. A ocorrência de falhas em sistemas distribuídos, o qual é o foco deste trabalho, já foi modelada por vários autores, entre eles Birman [Birman 1996]. Este modelo descreve as seguintes falhas: colapso, omissão de envio e recepção de mensagens e temporização.

4. FICommune

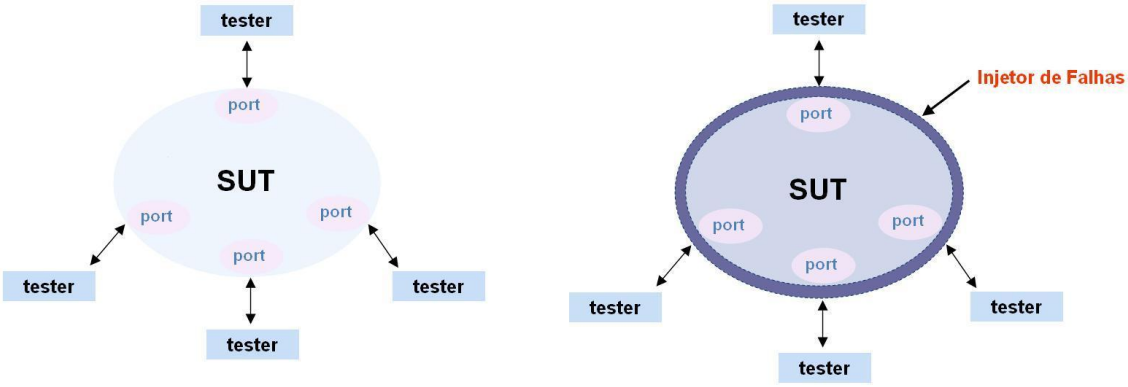
Com a crescente demanda de aplicações baseadas no Commune, observou-se a necessidade de se criar um ambiente de testes que simule as possíveis falhas do mundo real. Motivados por essa necessidade, desenvolveu-se a ferramenta de injeção de falhas *FICommune* como uma alternativa para teste dessas aplicações.

A principal finalidade do *FICommune* é proporcionar aos testadores de aplicações que utilizam Commune uma forma simples de injetar falhas de temporização e de omissão de envio e recebimento de mensagens. Para utilizar tal recurso, é necessário que o projeto importe o pacote de injeção de falhas e a partir dele programe as falhas (identificando as aplicações que terão as falhas injetadas).

Para abordar o funcionamento do *FICommune*, nesta Seção apresenta-se a arquitetura de um sistema a ser testado, *System under testing* (SUT), com injeção de falhas, comparando-a com a arquitetura de um sistema comum. Em seguida, faz-se uma comparação entre o FICommune e o FIRMI [Vacaro, Weber 2006]. Na seqüência, descreve-se o desenvolvimento da ferramenta de injeção de falhas. Por fim, uma breve introdução de como a ferramenta pode ser aplicada em um projeto qualquer.

4.1. Arquitetura do FICommune

Para testar uma aplicação SUT, é necessário que ela possua um conjunto de portas, para instrumentação dos Testadores (*Testers*). Os testadores irão fornecer os dados do teste para o SUT e coletar todas as saídas. Essas atividades deveriam, a princípio, acontecer apenas por meio das portas, e os testadores deveriam se comunicar entre si apenas através do SUT. O SUT com injeção de falhas possui características semelhantes ao SUT comum, diferindo apenas de uma camada que envolve o SUT, o injetor de falhas, e tem a capacidade de alterar as informações trocadas pelo SUT com outras aplicações.



(a) SUT comum (b) SUT com Injeção de Falhas

Figura 2: Ilustração de um SUT comum (a) e um SUT com injeção de falhas (b).

A camada de injeção de falhas pode ser transparente aos componentes externos caso não haja falhas relacionadas ao destinatário ou a fonte da mensagem que está em trânsito. Na Figura 2 (a) podemos observar um sistema comum onde, em condições ótimas, os componentes externos recebem os resultados esperados do SUT. Em contrapartida, na Figura 2 (b) existe uma camada de injeção falhas que pode simular problemas de ordem não determinística.

4.2. FIRMI x FICommune

A princípio, utilizou-se como base de conceitos para a estrutura da injeção de falhas o "Injetor de Falhas para a Avaliação de Aplicações Distribuídas baseadas em RMI" [Vacaro, Weber 2006] (FIRMI), a despeito da existência de várias características que o diferenciam do *FICommune*. A seguir, apresenta-se algumas características comuns às duas ferramentas de injeção de falhas, como também as principais vantagens e desvantagens do *FICommune* em relação ao FIRMI.

A principal característica em comum entre as duas ferramentas está relacionada às falhas suportadas pelo sistema. Ambos os sistemas utilizam a classe base *Fault*, a qual todas as classes de falhas a serem injetadas devem herdar, para que o sistema as caracterize como falhas suportadas pela ferramenta. Além disso, ambas as ferramentas implementam classes de falhas genéricas, tendo em comum as falhas de *CrashFault* e *TimingFault*, que caracterizam falhas por omissão e atraso, respectivamente, no envio e recebimento de mensagens. Parte desse código em comum foi aproveitado pelo *FICommune*.

Para ser utilizado, o FIRMI precisa ser executado junto à Máquina Virtual Java (em inglês, Java Virtual Machine, JVM) instalada no sistema operacional Linux. Para isso, o FIRMI possui um conjunto de classes que permitem essa intervenção junto a JVM. Em contrapartida, para utilizar o *FICommune* é necessário que a aplicação em questão importe a API do *FICommune*, a API do Commune com adição do protocolo de falhas e do aspecto que adiciona esse novo protocolo à pilha de protocolos, e a API do AspectJ Runtime [AspectJ 2008], que é necessária para rodar o aspecto.

Essas diferenças de utilização do *FICommune* em relação ao FIRMI trazem algumas vantagens e desvantagens. Como vantagens, podem-se citar sua estrutura descomplicada e sua independência de sistema operacional. Com relação às desvantagens, pode-se citar o fato de o Aspecto ser compilado com o Commune e de que a aplicação testada necessita importar todas as APIs citadas anteriormente.

4.3. Desenvolvimento do *FICommune*

O desenvolvimento da ferramenta *FICommune* ocorreu em duas etapas distintas: (i) criação de uma versão do Commune com suporte a injeção de falhas; (ii) implementação do injetor de falhas responsável pela carga e execução das falhas. A seguir, serão descritas essas duas etapas.

4.3.1. Commune com suporte a Injeção de Falhas

O primeiro passo para a criação da ferramenta *FICommune* foi a criação de uma versão do protocolo de comunicação Commune com suporte à injeção de falhas. Para tanto, fez-se necessário a interceptação de todas as mensagens trocadas entre as aplicações no sistema.

A princípio, considerou-se a possibilidade de desenvolver um *plugin* de um servidor XMPP que possibilitasse a interceptação das mensagens trocadas por aplicações baseadas no Commune, mas essa solução se tornou inviável pelo fato do injetor de falhas ficar atrelado a um servidor XMPP específico.

Para interceptar as mensagens trocadas por aplicações que utilizam o Commune, adicionou-se um novo protocolo à pilha de protocolos. No entanto, teve-se a preocupação de não alterar o código do Commune, de maneira a não ficar atrelado a uma versão específica deste.

A classe do Commune encarregada de criar a pilha de protocolos é a *NetworkInterface*. Ela é solicitada pelo *container* da aplicação no momento de sua criação. Em posse dessa informação, decidiu-se interceptar o método que cria a pilha de protocolos, de forma a adicionar o novo protocolo à pilha. A forma mais viável de fazer isso foi usar um aspecto. A função deste aspecto se divide em: capturar a instância do objeto *CommuneNetwork*, que possui a pilha de protocolos, retornada pelo método *NetworkInterfac.build()*; adicionar o protocolo *FaultInjectorProtocol* à pilha; e repassar esse objeto ao *container* de forma transparente. Dessa forma a adição do novo protocolo não altera o código do Commune e acontece de forma transparente às aplicações. Esse processo está ilustrado na Figura 3.

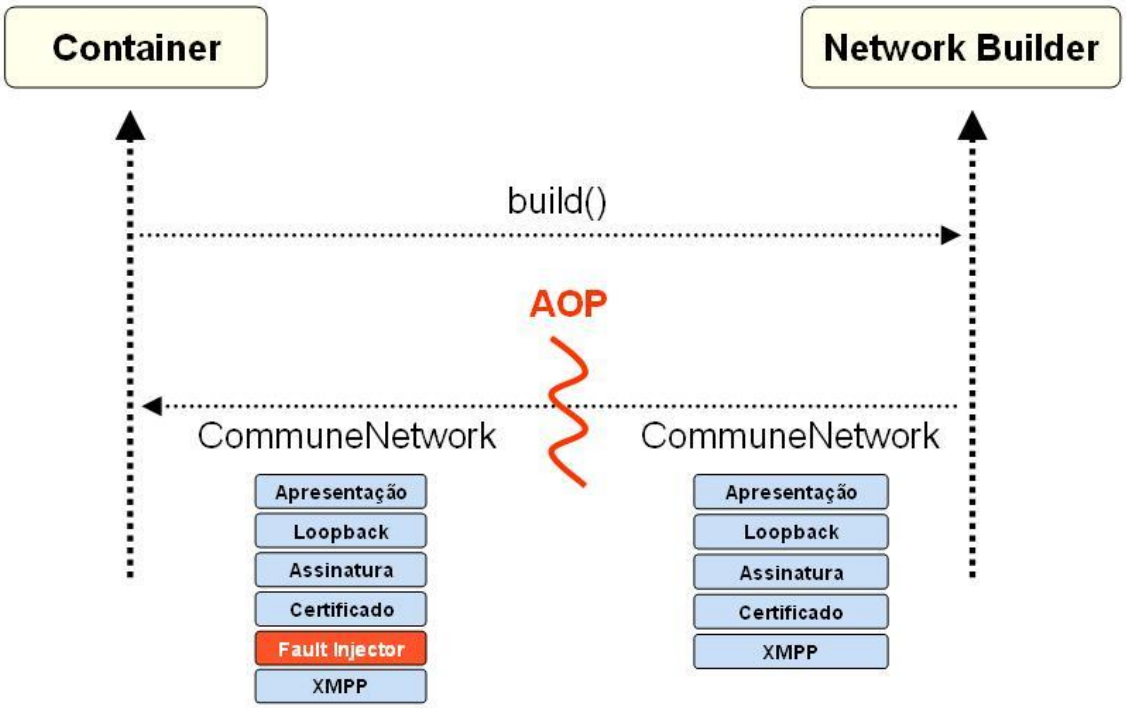


Figura 3: Inserção do protocolo de injeção de falhas pelo aspecto.

4.3.2. Injetor de Falhas

O Injetor de Falhas consiste em uma API que contém um carregador de falhas (*FaultLoader*) responsável por carregar e controlar as falhas desenvolvidas pelos testadores e classes de falhas genéricas que caracterizam falhas por omissão e atraso no envio e recebimento de mensagens (*Fault*, *CrashFault* e *TimingFault*). Um diagrama de classes simplificado dessas classes pode ser visto na Figura 4. A seguir, detalha-se o funcionamento dessas classes.

4.3.2.1. Classe *FaultLoader*

A principal funcionalidade da classe *FaultLoader* é carregar e gerenciar as falhas desenvolvidas pelos testadores. Esta classe é utilizada pelo *FaultInjectorProtocol* como interface para as falhas que serão injetadas nas aplicações. Os principais métodos dessa classe (veja Figura 3) são:

- getInstance() – utilizado para obter a instância única da classe *FaultLoader*;
- load() – carrega as falhas que herdam a classe *Fault*, e guarda uma referência para elas.
- start() – ativa todas as falhas carregadas;
- stop() – desativa todas as falhas carregadas;
- process(Message) – solicita que cada uma das falhas carregadas processe a referente mensagem.

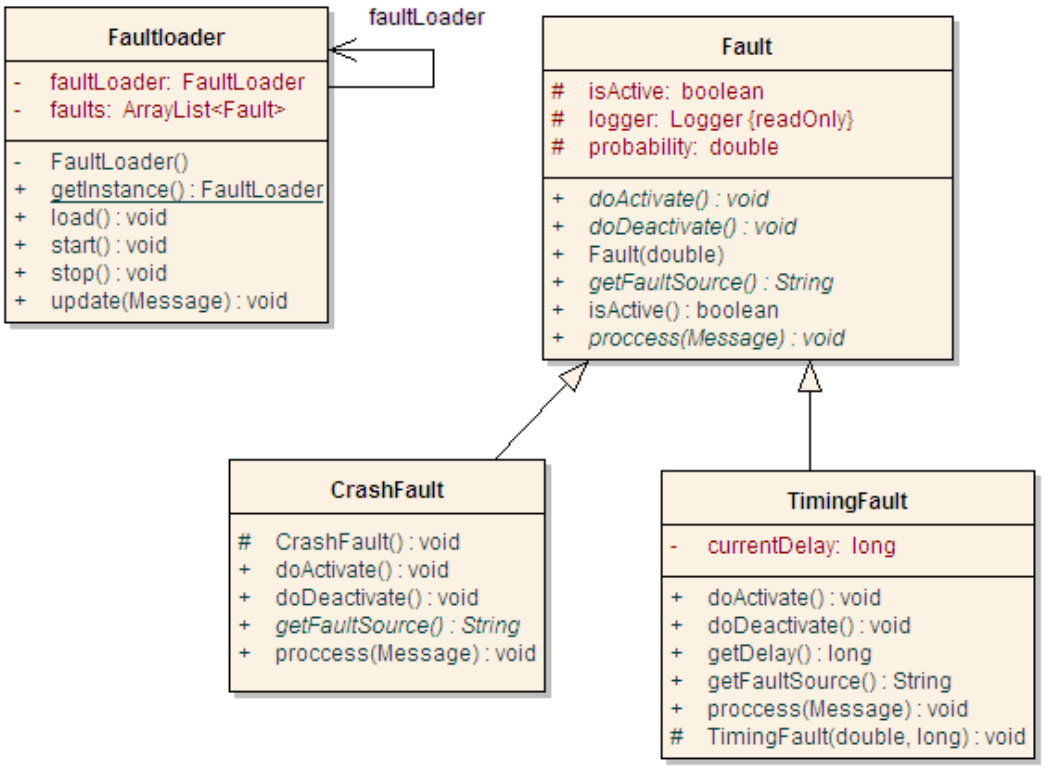


Figura 4: Diagrama simplificado da API de Injeção de Falha.

4.3.2.2. Classe *Fault*

A classe *Fault* define os comportamentos e características de uma falha para essa ferramenta. Por se tratar de uma classe abstrata, ela precisa ser herdada por outra classe que programe os métodos abstratos. Os principais atributos e métodos dessa classe (veja Figura 3) são:

- isActive – atributo que identifica se a falha está ativa ou não;
- logger – objeto responsável por relatar em arquivos de log as ações tomadas pela falha;
- probability – valor entre 0 e 1 que simboliza a probabilidade dessa falha ocorrer: por exemplo, caso o valor seja 0.2, na eventualidade do método *process* ser executado, a falha terá 20% de chances de ser injetada. Esse valor é recebido como parâmetro no construtor da classe;
- Fault(double) – construtor da classe que recebe como parâmetro a probabilidade dessa falha acontecer;
- doActivate() – ativa esta falha;
- doDeactivate() – desativa esta falha;
- isActive() – informa se a falha está ativa;

- process(Message) – verifica a probabilidade da falha acontecer e se a fonte ou o destino da mensagem recebida como parâmetro é a aplicação alvo da injeção de falhas. Em caso positivo, injeta a falha; caso contrário, não faz nada. Esse método abstrato deve ser implementado pela classe que identifica uma falha específica (por exemplo, *CrashFault* e *TimingFault*);
- getFaultSource() – informa a aplicação alvo da injeção de falhas. Deve ser implementado pela classe de falha criada pelo testador (por exemplo, *PongTimingFault*).

4.3.2.3. Classe *CrashFault*

A classe *CrashFault* é uma subclasse da classe *Fault*, que representa uma classe especial de falhas (as falhas por omissão de envio ou recebimento de mensagens). O único método que a difere consideravelmente da superclasse é o método *process(Message)*.

A ação de injeção de falha tomada pelo método *process(Message)* é descartar a mensagem lançando uma exceção do tipo *DiscardMessageException* (exceção da API do Commune que identifica o descarte de uma mensagem).

4.3.2.4. Classe *TimingFault*

A classe *TimingFault* é uma subclasse da classe *Fault*, que representa uma classe especial de falhas: as falhas por atraso de envio ou recebimento de mensagens. Os únicos comportamentos que diferem consideravelmente da superclasse são o construtor *TimingFault(long, double)* e o método *process(Message).*

O construtor *TimingFault(long, double)* além de receber como parâmetro a probabilidade da falha acontecer, recebe também um número do tipo *long* que representa o tempo de atraso, em milisegundos, que será utilizada na injeção da falha.

A ação de injeção de falha tomada pelo método *process(Message),* é atrasar a mensagem durante o intervalo de tempo passado como parâmetro. Esse atraso é executado pelo método *Thread.sleep()*.

4.4. Aplicação do FICommune

Para que o injetor de falhas desenvolvido seja utilizado na aplicação na qual se deseja inserir falhas para testes, necessita-se carregar alguns pacotes. Estes pacotes são: *ficommune.jar*, *commune.jar* e *aspectjrt.jar*. O pacote *ficommune.jar* equivale à biblioteca do injetor de falhas *FiCommune*. Nela estão todas as classes necessárias para a inserção das falhas na aplicação alvo. O segundo pacote, *commune.jar*, possui a biblioteca do *middleware Commune*. O último pacote, *aspectjrt.jar*, é necessário para a inserção do injetor, *FiCommune*, na pilha de protocolos do *Commune*, através da programação orientada a aspectos com *AspectJ* [AspectJ 2008].

Uma vez que a aplicação que será testada já possui estes pacotes "carregados", o testador deve criar as classes de falhas, herdando das classes *CrashFault* ou

TimingFault, que representam, respectivamente, falha por *crash* ou falha por *timing* (*delay*), acrescentando o que for necessário para o teste de interesse; o testador pode, ainda, herdar da classe *fault* e desenvolver um outro tipo de falha (além das de *crash* e *timing*), como, por exemplo, uma falha de particionamento de rede.

As classes de falhas devem ser colocadas no diretório *faults*, que é o padrão. Para utilizar outra pasta, altera-se o arquivo de configuração "*.properties"*, substituindo o parâmetro *faults* pelo nome do diretório desejado.

As Figuras 5 e 6 a seguir mostram os trechos das classes de falhas nos quais os testadores devem se basear na criação de suas falhas.

```
1  import ficommune.fault.CrashFault;
2
3  public class PongCrashFault extends CrashFault {
4
5      public PongCrashFault() throws RuntimeException {
6          super(0);
7      }
8
9      public String getFaultSource() {
10         return "pong@127.0.0.1/PongApp";
11     }
12 }
```

Figura 5: Exemplo de classe que indica onde a falha por *crash* deve ser injetada.

```
1  import ficommune.fault.TimingFault;
2
3  public class PongTimingFault extends TimingFault {
4
5      public PongTimingFault() {
6          super(2000, 0.2);
7      }
8
9      public String getFaultSource() {
10         return "pong@127.0.0.1/PongApp";
11     }
12 }
```

Figura 6: Exemplo de classe que indica onde a falha por temporização deve ser injetada.

5. Experimento

Objetivando testar o injetor de falhas *FiCommune*, variou-se a probabilidade das falhas acontecerem, tanto no caso do *CrashFault*, quanto no caso do *TimingFault*. Além disso, para a classe *TimingFault*, que representa falha por atraso na mensagem, também foi variado um outro parâmetro utilizado pela mesma: o tempo de atraso.

As falhas foram injetadas em uma aplicação de "*Ping-Pong*", sistema de troca de mensagens. Após os pacotes necessários para o correto funcionamento do injetor serem

adicionados à aplicação a ser testada, e as classes de falhas serem armazenadas na pasta *faults* (padrão), a probabilidade de ocorrência de falhas, em ambas as classes (*CrashFault* e *TimingFault*), foi definida com valor zero. Neste caso, a aplicação *Ping-Pong* funcionou normalmente, sem falhas (veja Figura 7). Em seguida, alterou-se apenas o valor da probabilidade de ocorrência de falha por *crash* (*CrashFault*). Dessa forma, pode-se perceber que a falha ocorreu com 100% de chance e que a mensagem não chegou ao destinatário, sendo sempre descartada (veja Figura 8). Em outra situação, alterou-se a probabilidade da ocorrência de falha por atraso de mensagem (*TimingFault*) para o valor um. Neste caso, o valor do atraso passado como parâmetro foi 2000 ms, fazendo com que o atraso das mensagens fosse igual a dois segundos. Neste teste, a falha também aconteceu, sempre que executado com esta probabilidade, e as partes comunicantes da aplicação ficaram tentando se comunicar, mas não conseguiam devido ao atraso dado, atraso este que fazia com que ambos os lados ficassem dormindo durante os dois segundos (de forma independente). As entidades comunicantes até conseguiam se perceber, ou seja, trocar mensagens de controle, mas não trocar alguma mensagem da aplicação (*Ping-Pong*) (veja Figura 9).

Por último, foram atribuídas probabilidades entre zero e um, seguindo o mesmo teste descrito no parágrafo anterior: atribuindo os valores a uma classe de falhas e depois à outra. Desse modo, observou-se que, na falha por *crash* a injeção de falhas seguiu o valor da probabilidade passado e, como não era igual a um (*i.e.*, 100% de chance de ocorrer), chegava um determinado momento em que a mensagem não era descartada (veja Figura 10). Além disso, as mensagens que conseguiram chegar em "*ping*" também foram descartadas de acordo com a probabilidade passada. Da mesma forma, para a falha de temporização, as partes comunicantes também conseguiram, em algum momento, realizar a comunicação (veja Figura 11).

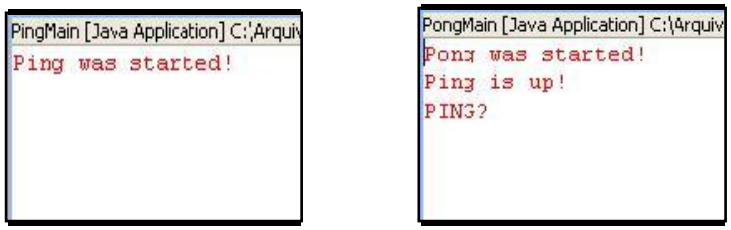


Figura 7: Aplicação executada sem falha inserida.

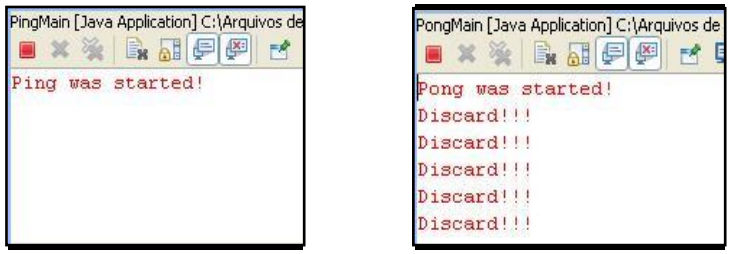


Figura 8: Aplicação executada com probabilidade um de ocorrência de falha por *crash*.

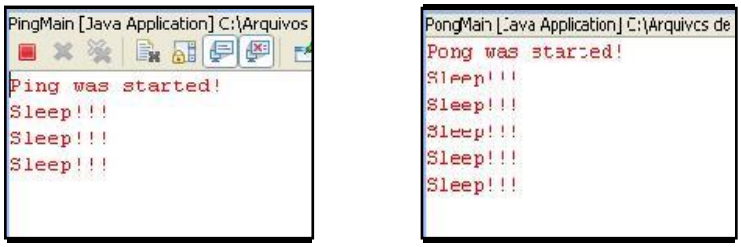


Figura 9: Aplicação executada com probabilidade um de ocorrência de falha por temporização.

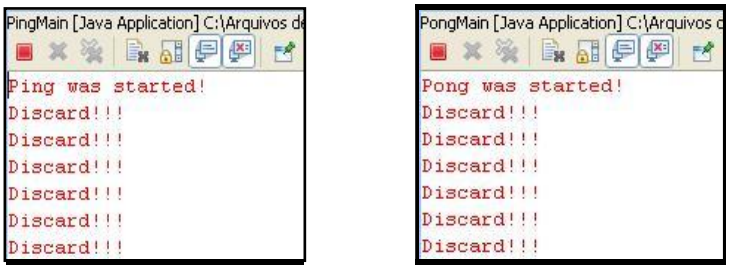


Figura 10: Aplicação executada com probabilidade 0,5 de ocorrência de falha por *crash*.

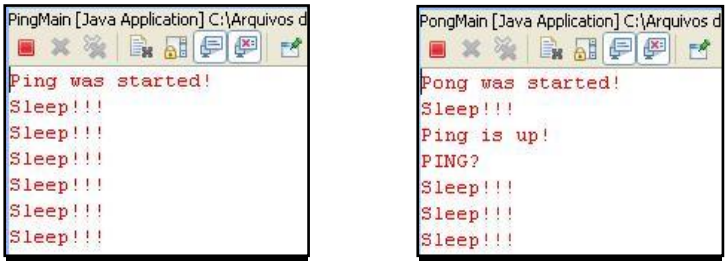


Figura 11: Aplicação executada com probabilidade 0,2 de ocorrência de falha por temporização.

6. Trabalhos Relacionados

Várias ferramentas injetam falhas de comunicação diretamente através de software, com o objetivo de diminuir a latência e permitir um maior controle sobre a manifestação da falha injetada, ao invés de fazer através da CPU e da memória. DOCTOR [Han, Shin, Rosenberg 1995] injeta falhas de memória, CPU ou de comunicação em ambientes distribuídos de tempo real. ORCHESTRA [Dawson, Jahanian et al. 1996] é utilizado na avaliação de protocolos de comunicação, onde a ferramenta é inserida na pilha de protocolos do sistema operacional, na camada imediatamente abaixo do protocolo sob testes, atuando sobre cada mensagem que flui pela pilha. Esta ferramenta inspirou ComFirm [Drebes, Leite et al. 2005], um módulo de kernel Linux que atua na camada IP. FIONA [Gerchman, Jacques-Silva et al. 2005] atua instrumentando classes de comunicação de sistema. A menor intrusividade temporal foi alcançada com a versão usando JVMTI [Jacques-Silva, Drebes et al. 2004], e uma ferramenta semelhante foi desenvolvida em paralelo na IBM [Farchi, Krasny, Nir 2004], também baseada na instrumentação da comunicação UDP. O FIRMI [Vacaro, Weber 2006] avalia o

comportamento sob falhas de aplicações que usam o RMI como infra-estrutura de comunicação, bem como a cobertura de falhas dos mecanismos de tolerância a falhas empregados.

7. Conclusão

Para que um produto de *software* seja de boa qualidade, é preciso integrar ao desenvolvimento de software uma política para testes. Os testes não garantem a total certeza de infalibilidade do produto, mas ajudam a encontrar comportamentos inesperados que podem ser corrigidos. Aplicações críticas demandam software altamente confiável e resistente a falhas, de modo que, ainda que não se consiga recuperar totalmente, minimize ao máximo as conseqüências danosas.

No caso de sistemas distribuídos, fatores extras surgem para afetar o funcionamento ao longo do tempo. Falhas como a perda de comunicação ou o atraso, entre outros mais, adicionam uma complexidade ao planejamento e execução de testes no sistema.

É pouco prático, e às vezes inviável, introduzir as próprias falhas na rotina de testes como, por exemplo, desconectar cabos de rede ou desligar máquinas. Portanto, é essencial o suporte de injetores de falhas que as simulem através de *software*.

Existem várias ferramentas para injeção de falhas, mas nenhuma delas é voltada diretamente para aplicações que usem o Commune como *middleware* de comunicação. O injetor de falhas (*FiCommune*) é capaz de inserir de forma simples falhas descritas pelo testador. Essa ferramenta, que já teve sua eficiência comprovada pelos experimentos, possui a vantagem de ser facilmente adaptada quando uma nova versão do Commune for lançada.

Atualmente as falhas são injetadas no nível das aplicações. Uma melhoria da ferramenta poderia envolver um aumento na abrangência das falhas, de maneira a podermos definir servidores e *containers* como foco de injeção de falhas. Outra melhoria futura seria ter relatórios salvos em arquivos de *logs*, de maneira que se possa ter um controle de todas as falhas que foram injetadas e com que freqüência elas aconteceram.

8. Referências

AspectJ Project (2008). http://www.eclipse.org/aspectj/.

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

Cirne, W., Brasileiro. F., Mowbray. M et al. (2006). “Labs of the World, Unite!!!”, Journal of Grid Computing, 4(3).

Cirne, W., Brasileiro. F., Fireman, D (2006). “A case for Event-Driven Distributed Objects”, Proceedings.of DOA’06, LNCS.

Commune project, http://commune.lsd.ufcg.edu.br/index.php/Main_Page.

Dawson, S., Jahanian, F., Mitton, T., and Tung, T.-L. (1996). “Testing of fault-tolerant and real-time distributed systems via protocol fault injection”, Proceedings of the 26th IEEE International Symposium on Fault-Tolerant Computing (FTCS ’96), pages 404–414, Sendai, Japan. IEEE Computer Society Press.

Drebes, R. J., Leite, F. O., Jacques-Silva, G., Mobus, F., and Weber, T. S. (2005). ComFIRM: a communication fault injector for protocol testing and validation. In IEEE Latin American Test Workshop, 6th (LATW’05), pages 115–120, Salvador, Brazil.

Farchi, E., Krasny, Y., and Nir, Y. (2004). “Automatic simulation of network problems in UDP-based Java programs”, Proceedings of the International Parallel and Distributed Processing Symposium, 18th (IPDPS’04), page 267, Santa Fe, New Mexico, USA. IEEE Computer Society.

Gerchman, J., Jacques-Silva, G., Drebes, R., and Weber, T. S. (2005). “Ambiente distribuído de injeção de falhas de comunicação para teste de aplicações java de rede”, Anais do 19 Simpósio Brasileiro de Engenharia de Software (SBES 2005), pages 232–246, Uberlândia, MG. SBC.

Han, S., Shin, K. G., and Rosenberg, H. (1995). “DOCTOR: An integrateD sOftware fault injeCTiOn enviRonment for distributed real-time systems”, Proceedings of the International Computer Performance and Dependability Symposium, pages 204–213, Erlangen, Germany. IEEE Computer Society Press.

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

Jacques-Silva, G., Drebes, R. J., Gerchman, J., and Weber, T. S. (2004). “FIONA: A fault injector for dependability evaluation of Java-based network applications”, Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (NCA’04), pages 303–308, Washington, DC, USA. IEEE Computer Society.

Vacaro J., Weber T. (2006). “FIRMI: Um Injetor de Falhas para a Avaliação de Aplicações Distribuídas baseadas em RMI”. WTF 2006 - VII Workshop de Testes e Tolerância a Falhas, Curitiba, PR. SBC.

Walter T., Schieferdecker I., Grabowski J. “Test Architectures for Distributed Systems - State of the Art and Beyond”, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.467.

XMPP Standards Foundation (2008). http://www.xmpp.org/.