Validando Sistemas Distribuídos Desenvolvidos em Java Utilizando Injeção de Falhas de Comunicação por Software*

Gabriela Jacques-Silva[1 †], Regina Lúcia de O. Moraes[2] ,
Taisy Silva Weber[1] , Eliane Martins[3]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul
Caixa Postal 15064 – 90501-970 Porto Alegre, RS

[2]CESET – Universidade Estadual de Campinas
Caixa Postal 456 – 13484-370 Limeira, SP

[3]Instituto de Computação - Universidade Estadual de Campinas
Caixa Postal 6176 – 13083-870 Campinas, SP

gjsilva@inf.ufrgs.br, regina@ceset.unicamp.br,
taisy@inf.ufrgs.br, eliane@ic.unicamp.br

Abstract. *Java distributed applications used in high-available systems require validated fault tolerance mechanisms, to avoid unexpected behavior during execution. Jaca is a fault injection tool to experimentally validate Java applications. In its first version, Jaca had three fault classes, based on interface faults. This work presents an expansion of Jaca's fault model to communication faults. This extension allows the use of Jaca to conduct validation experiments in distributed systems.*

Resumo. *Aplicações distribuídas em Java usadas em sistemas de alta disponibilidade exigem que mecanismos de tolerância a falhas sejam validados, para não apresentarem um comportamento inesperado no momento em que são requisitados no ambiente operacional. Jaca é uma ferramenta de injeção de falhas para validação experimental de aplicações Java. Em sua primeira versão, Jaca apresentava três classes de falhas, baseadas em falhas de interface. Este trabalho apresenta a expansão do modelo de falhas de Jaca para falhas de comunicação. Essa extensão permite o uso dessa ferramenta na condução de experimentos de validação em sistemas distribuídos.*

1. Introdução

Permitir que sistemas computacionais sejam incorporados ao cotidiano para atender serviços de missão crítica exige que os mesmos apresentem características de tolerância a falhas para corresponder a confiança depositada no comportamento correto desses serviços. Uma etapa fundamental no desenvolvimento de sistemas tolerantes a falhas é a fase de validação. Delegar a verificação do funcionamento do mecanismo de tolerância a falhas para uma situação de uso efetivo do *software* (uma falha real) pode gerar conseqüências desastrosas.

A validação pode ser tanto analítica quanto experimental, sendo estas duas formas complementares. Uma das técnicas usadas para validar experimentalmente um sistema é injeção de

*Parcialmente financiado pelos projetos ACERTE/CNPq (472084/2003-8) e DepGriFE/HP Brasil P&D
†Bolsista do Conselho Nacional de Desenvolvimento Científico e Tecnológico

falhas. Através desta técnica, são introduzidas falhas no sistema de maneira controlada e é monitorada a resposta do sistema nessas condições. Seu objetivo é testar a eficiência dos mecanismos de tolerância a falhas e avaliar a segurança de funcionamento dos sistemas, provendo uma realimentação no processo de desenvolvimento [Iyer 1995].

Sistemas distribuídos são comumente usados para oferecer alta disponibilidade para serviços de missão crítica, com isso faz-se indispensável a validação da sua dependabilidade. Uma maneira é injetar falhas no sistema de troca de mensagens, para forçar a ativação dos mecanismos de detecção de erro e recuperação do estado distribuído. Se forem usados outros modelos de falhas, como falhas de memória ou da unidade de processamento, a latência de manifestação da falha será muito grande. Para acelerar essas manifestações, as falhas são injetadas diretamente no sistema de troca de mensagens.

Este trabalho apresenta um injetor de falhas de comunicação escrito na linguagem de programação Java. Através desta ferramenta pode ser testada mais facilmente a dependabilidade em sistemas distribuídos implementados em Java, que é largamente usada no desenvolvimento de aplicações para este tipo de ambiente. A maior vantagem de injetar falhas em um nível acima da pilha de protocolos de comunicação do sistema operacional é a portabilidade, pois só dependerá de Java, portável para várias arquiteturas e sistemas operacionais.

Para desenvolver esta ferramenta estendeu-se o injetor de falhas Jaca [Martins *et al.* 2002]. Jaca é baseada no sistema de padrões para injeção de falhas [Leme *et al.* 2001] e sua implementação em reflexão computacional [Maes 1987]. Uma de suas principais características é a possibilidade de ser estendida para novos modelos de injeção de falhas. Neste trabalho, esta característica é explorada para injeção de falhas de comunicação no protocolo UDP. A próxima seção apresenta alguns mecanismos de injeção de falhas por *software*, mostrando algumas ferramentas de injeção de falhas e também o sistema de padrões existente para este tipo de ferramenta. A seção 3 apresenta Jaca e também como reflexão computacional foi usada em sua implementação. A seção 4 descreve a extensão da ferramenta, mostrando o modelo de falhas adicionado e como este foi implementado. A última seção apresenta algumas conclusões e trabalhos futuros.

2. Injeção de Falhas

Injeção de falhas é uma técnica de teste em que se procura produzir ou simular falhas e observar o sistema sob teste para verificar sua resposta nessas condições [Hsueh *et al.* 1997]. Conhecendo o modelo de falhas do sistema sob teste, podem ser criadas falhas que são injetadas no sistema de acordo com o modelo suportado. Durante e após a injeção das falhas, o sistema é monitorado para observar o efeito causado. Através deste tipo de experimento podem ser determinadas medidas de dependabilidade, como a cobertura da detecção de erros, a eficiência e o impacto no desempenho dos mecanismos de tolerância a falhas. Experimentos de injeção de falhas também podem revelar problemas de *software*, que não são encontrados com técnicas tradicionais de teste e métodos formais [Voas 1997].

A injeção de falhas pode ser aplicada a diferentes fases do ciclo de vida de um sistema, usando diferentes formas de aplicação, sendo as mais comuns injeção de falhas por simulação, por *hardware* e por *software*. A ferramenta Jaca e sua extensão se enquadram em injeção de falhas por *software*.

2.1. Injeção de Falhas por *Software*

Uma ferramenta de injeção de falhas por *software* geralmente é um trecho de código que usa todos os ganchos (*hooks*) possíveis do processador e do sistema para criar um comporta-

mento incorreto de maneira controlada [Carreira e Silva 1998]. Esta técnica pode simular falhas de *hardware*, de *software* e de interface. Interessam a esse trabalho falhas de *hardware* emuladas e injetadas por *software*. Nesse caso, são as manifestações de falhas de *hardware* (erros) que são injetadas e não as falhas propriamente ditas.

Exemplos de falhas de *hardware* são: falhas de memória, de processador, de barramento e, no caso de sistemas distribuídos, falhas de comunicação. Estas afetam mensagens que são transmitidas através de um canal de comunicação, que podem ser omitidas, alteradas, duplicadas ou entregues com atraso.

Os mecanismos que podem ser usados para injetar falhas por *software* são classificados em três categorias [Rosenberg e Shin 1993]: (i) a injeção ativa é realizada por um processo especial executado concorrentemente com os processos que fazem parte do sistema sob teste, podendo injetar falhas em ambientes onde não haja proteção contra acesso externo; (ii) a injeção por alteração de fluxo de controle é usada para alterar o comportamento funcional do sistema sob teste e, quando ativada, executa uma seqüência alternativa de instruções, de modo que a função corrente seja executada incorretamente; (iii) a injeção por alteração de código é usada para injetar falhas em áreas do programa nas quais um processo externo não pode ter acesso. A execução da aplicação sob teste é interrompida e em seu lugar é executado um código que injeta falhas no recurso do sistema especificado.

Essa alteração pode ser estática, quando feita em tempo de compilação, ou dinâmica, quando feita em tempo de execução. Na alteração estática, as falhas são introduzidas no código do programa alvo, alterando instruções do programa. A vantagem desta abordagem é que não é necessário nenhum *software* extra durante a execução para injetar falhas, porém, o modelo de falhas que pode ser injetado é limitado a falhas de *software* e conseqüências de falhas permanentes de *hardware*. Na alteração dinâmica, código extra é necessário para injetar falhas e monitorar seus efeitos e, também, é requerido um mecanismo para disparar a injeção de falhas.

Para a alteração dinâmica pode-se usar os seguintes métodos [Hsueh *et al.* 1997]: modo depuração, baseado em *time out*, exceções/interrupções e inserção de código. Jaca é baseado em exceções/interrupções, onde as falhas são disparadas por ocasião da ocorrência de alguns eventos, tais como a execução de uma determinada instrução, chamadas a rotinas de sistema ou o acesso a uma determinada posição de memória.

Um problema a ser considerado quando se usa mecanismos dinâmicos de injeção, é o fato de eles serem intrusivos, uma vez que sua execução não é transparente para o sistema alvo, alterando, muitas vezes de maneira bastante significativa, seu desempenho. Dessa forma os resultados obtidos são afetados devido à sobrecarga causada pela inserção do código do injetor de falhas. As ferramentas de injeção, além de interferirem no sistema para introduzir os erros desejados, precisam monitorar o sistema e determinar se a falha e os mecanismos de tolerância à falhas foram ativados, coletando dados que auxiliem no diagnóstico dos erros apresentados. Uma maneira de se reduzir essa interferência é usar técnicas híbridas, empregando ferramentas de *software* e de *hardware*.

Para se aplicar injeção de falhas por *software* não é preciso de *hardware* especial, não há risco de dano aos componentes e os testes podem ser mais facilmente controlados e observados. Devido a essas vantagens, injeção de falhas por *software* tem se tornado mais popular entre os desenvolvedores de sistemas tolerantes a falhas.

2.2. Ferramentas de Injeção de Falhas

Para que se possa injetar falhas é necessário dispor de ferramentas apropriadas. Essas ferramentas executam o sistema sob teste e produzem ou simulam a presença de falhas, monitorando

o sistema para verificar qual é seu comportamento. Para que se tenha sucesso nos testes, a ferramenta deve ser eficiente tanto no momento de injetar as falhas quanto ao reportar os resultados dos testes e o comportamento do sistema ao tratar as falhas injetadas.

Alguns exemplos de ferramentas de injeção de falhas de comunicação são CSFI, ORCHESTRA e ComFIRM. CSFI (*Communication Software Fault Injection*) [Carreira *et al.* 1995b] foi uma das primeiras ferramentas desenvolvidas unicamente para injeção de falhas de comunicação e seu objetivo principal era avaliar o impacto de falhas em sistemas paralelos. A versão existente de CSFI foi implementada para um sistema *transputer* T805. Outro ambiente bastante conhecido é ORCHESTRA [Dawson *et al.* 1996], desenvolvido especificamente para teste de dependabilidade de protocolos distribuídos. O mecanismo de injeção de falhas é através da inserção de uma camada na pilha de protocolos, chamada de PFI (*Protocol Fault Injection*). A ferramenta ComFIRM (*Communication Fault Injection through OS Resources Modification*) [Barcelos *et al.* 2000] propõem-se a injetar falhas apenas de comunicação. ComFIRM se situa no núcleo do sistema operacional Linux, no nível mais baixo do tratamento de mensagens pelo subsistema de rede. O código da ferramenta é inserido diretamente código do núcleo do sistema operacional, o que diminui consideravelmente a intrusão da ferramenta no sistema sob teste.

Outra classe de ferramentas é a das que permitem a extensão de seu modelo de falhas, tais como GOOFI, NFTAPE e FIDe. GOOFI (*Generic Object-Oriented Fault Injection*) [Aidemark *et al.* 2001] é uma ferramenta desenvolvida recentemente e é considerada genérica por não ser presa a nenhuma técnica específica de injeção de falhas. Desta maneira é construído um ambiente de fácil adaptação para injeção de falhas necessárias para um determinado sistema alvo. Esta ferramenta é altamente portável por ser desenvolvida em Java e usar um banco de dados compatível com a linguagem SQL. Sua principal diferença em relação a Jaca é que o seu desenvolvimento não foi baseado em um sistema de padrões de *software*. A ferramenta NFTAPE [Stott *et al.* 2000] destaca-se por ser uma ferramenta com múltiplos modelos de falhas, diversos modos de disparo para injeção de falhas e também diversos sistemas alvo. Para permitir essa multiplicidade, NFTAPE inova criando o conceito de Injetor “Leve” de Falhas (*LightWeight Fault Injector* – LWFI). Para desenvolver um novo injetor de falhas, apenas um novo LWFI precisa ser implementado, o que facilita sua expansibilidade para novos modelos de falhas. FIDe (*Fault Injection via Debugging*) [Gonçalves *et al.* 2001] é um injetor de falhas baseado nos recursos de depuração do sistema operacional Linux. Para injetar falhas, FIDe usa a chamada de sistema `ptrace()`, que intercepta a aplicação toda vez que uma chamada de sistema ocorre. A extensibilidade de FIDe se diferencia de outras ferramentas, como GOOFI e NFTAPE, pois sua capacidade de expansão está diretamente ligada a flexibilidade da chamada `ptrace()`. O modelo de falhas desta ferramenta é tão amplo quanto as chamadas de sistemas que são interceptadas por `ptrace()`.

Jaca é uma ferramenta de injeção de falhas extensível destinada à validação de aplicações orientadas a objeto desenvolvidas em Java. Sua implementação é baseada em reflexão computacional e sua arquitetura baseada em um sistema de padrões de *software*. Ao contrário de Jaca, a maioria das ferramentas descritas apresentam graves problemas de portabilidade, que vão desde o desenvolvimento para uma arquitetura específica (CSFI, para *transputers*) até o desenvolvimento para uma versão específica de núcleo de sistema operacional (ComFIRM, existente apenas para uma versão antiga do *kernel* do Linux). Além disso, por estas ferramentas terem sido criadas dentro de universidades, a maioria delas já tiveram seu desenvolvimento descontinuado, o que dificulta sua utilização.

2.3. Sistema de Padrões para Injeção de Falhas

No desenvolvimento de *software* é comum deparar-se com problemas que são recorrentes dentro de um determinado domínio de aplicações e não raro, resolvem-se esses problemas de forma similar a outras soluções já utilizadas por outros desenvolvedores. Essas soluções podem ser documentadas em forma de padrões que são independentes de linguagens de programação. Esses padrões auxiliam o desenvolvimento mais eficiente, robusto, portável e reutilizável [Schmidt 1995], facilitando o entendimento e instanciando soluções que têm sido consideradas eficientes para solucionar os problemas a que se propõem.

Baseado nessa constatação e na arquitetura proposta por Hsueh [Hsueh *et al.* 1997], foi desenvolvido um padrão para o desenvolvimento de novas ferramentas de injeção de falhas [Leme *et al.* 2001] que contempla: (i) ativação e monitoramento do sistema sob teste, (ii) coordenação dos experimentos e (iii) apresentação e armazenamento dos dados coletados.

Uma ferramenta de injeção de falhas pode ser decomposta em cinco subsistemas: (i) o *Controlador* que coordena os demais subsistemas, (ii) o *Injetor* que injeta as falhas no sistema sob teste, (iii) o *Monitor* que coleta as ocorrências relacionadas ao comportamento do sistema em presença das falhas injetadas, (iv) o *Ativador* que ativa o sistema e (v) a *Interface de Usuário* que permite que os usuários especifiquem os experimentos, acompanhem o progresso desses experimentos e obtenham os resultados.

O *Injetor*, o *Monitor* e o *Ativador* são os únicos subsistemas que podem interagir diretamente com a aplicação. Diversas instâncias do *Injetor* e do *Monitor* podem ser criadas para permitir a injeção e o monitoramento de diferentes partes do sistema sob teste durante um experimento. Além desses componentes, dois repositórios de dados são utilizados, um que armazena as falhas a serem injetadas (*Gerenciador de Falhas*) e outro que armazena os aspectos do sistema a serem monitorados (*Gerenciador de Monitoração*).

O *Injetor* e o *Monitor* por sua vez, são subsistemas que podem usar um padrão próprio, definindo um padrão de mais baixo nível. O *injetor* é composto de três componentes: o *gerenciador de injeção*, o *injetor lógico* e o *injetor físico*. O *gerenciador de injeção* controla o processo de injeção através da criação dos *injetores lógicos* de acordo com o tipo de falha especificada no *Gerenciador de Falhas* e ativa o *injetor lógico* apropriado quando chega o momento de injeção. Um *injetor lógico* é específico para uma determinada falha, mas não interage diretamente com o sistema sob teste. Isto será feito através do *injetor físico*, que irá efetivamente injetar as falhas especificadas através de sua interface, que contém operações que permitem essa interação. No caso da aplicação ou seu ambiente serem trocados, apenas esse componente deverá ser reescrito.

O padrão do Monitor é semelhante ao padrão acima descrito e por esse motivo não será aqui reapresentado.

3. Jaca – Ferramenta de Injeção de Falhas em Java

Como já citado na subseção 2.2, a principal motivação da ferramenta Jaca é auxiliar na validação de aplicações implementadas em Java. Jaca é uma evolução da ferramenta FIRE [Martins e Rosa 2000], que também usa de reflexão computacional para introduzir falhas no sistema alvo. A diferença entre as duas ferramentas é que, além de Jaca basear-se nos padrões de *software* descritos na seção 2.3, FIRE visa validar aplicações implementadas em C++. Jaca oferece mecanismos para injetar falhas de alto nível em sistemas orientados a objetos. Ao invés de afetar *bytes* de memória ou o conteúdo de registradores, Jaca afeta a interface pública de objetos, ou seja, atributos públicos, bem como parâmetros e valores de retorno de métodos públicos.

Para injetar falhas de alto nível, Jaca usa reflexão computacional, permitindo uma fácil adaptação, atendendo a especificação de qualquer sistema Java com um mínimo de reescrita, e não necessitando do código fonte do sistema sob teste. Isso acontece devido ao uso de uma plataforma de apoio a reflexão chamada Javassist [Chiba 2000], que permite a transformação de *bytecodes* em tempo de carga. Com isso, a instrumentação necessária para a injeção é introduzida no *bytecode*, podendo então ser aplicada mesmo que o código fonte não esteja disponível. Essa característica proporciona independência e portabilidade da ferramenta que pode rodar em qualquer plataforma que possa executar a máquina virtual padrão do Java. Jaca também monitora os sistemas para verificar se as respostas por ele produzidas estão dentro dos padrões esperados, mesmo em presença de falhas.

A figura 1 mostra a arquitetura da ferramenta Jaca. Como pode ser visto, sua estrutura de pacotes é semelhante à estrutura dos subsistemas do padrão para ferramentas de injeção de falhas. A funcionalidade de cada pacote segue a mesma funcionalidade descrita para cada um dos subsistemas equivalentes do padrão.

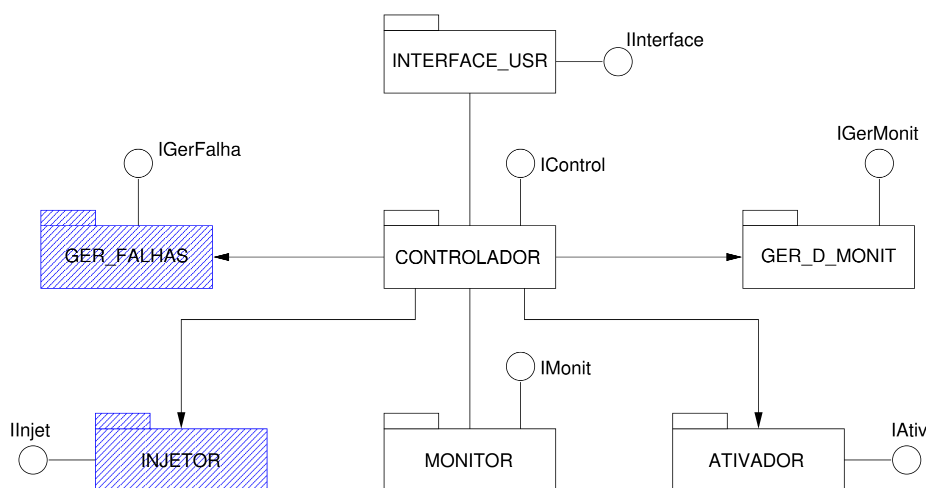

Figura 1: Arquitetura da ferramenta Jaca

3.1. Reflexão Computacional em Jaca

De acordo com Maes [Maes 1987], computação reflexiva é a atividade desempenhada por um sistema computacional quando a computação realizada é sobre a sua própria computação. A base para a construção de sistemas reflexivos é a sua divisão em dois níveis: o *nível base* e o *meta-nível*. O *nível base* é onde se encontra a implementação da aplicação. O *meta-nível* é onde as estruturas do *nível base* podem ser observadas ou ter seu comportamento modificado. Uma possível implementação destes conceitos a linguagens orientadas a objeto é o uso de *meta-objetos*.

Quando Jaca foi implementada, Javassist [Chiba 2000] foi escolhido como ferramenta de reflexão devido a sua conformidade com os requisitos da Jaca. Requisitos como portabilidade, introspecção nas classes, independência de código fonte e facilidade de programação foram possíveis com o uso de Javassist. A arquitetura de Jaca não é presa a nenhum protocolo de meta-objetos especificamente. Porém, quando um é escolhido para implementação, este será um fator limitante, já que Jaca vai estar limitada pelo poder de atuação do próprio protocolo.

O uso das classes do *toolkit* de reflexão computacional em Jaca se dá basicamente em três pacotes: `INJETOR`, `MONITOR` e `ATIVADOR`. O pacote `INJETOR` é o que faz a injeção de falhas em si, através da classe `InjetorFisico`. Para injetar de falhas, o `InjetorFisico` é implementado como um meta-objeto, sendo uma subclasse da classe

`javassist.reflect.Metaobject` de Javassist. Este `InjetorFisico` é associado a cada objeto da lista de classes, fornecida por um arquivo de configuração usado para identificar quais classes devem ser monitoradas. Desta forma os objetos especificados são interceptados sempre que há a leitura ou escrita de um atributo de classe ou então quando uma chamada de método é executada. É durante esta interceptação que o processo de injeção de falhas ocorre. O tipo de falha é buscada em um arquivo, que contém a lista de falhas que serão injetadas durante o experimento. Este arquivo é interpretado pelo gerenciador de falhas, que passa ao gerenciador de injeção, através do controlador, as informações necessárias para instanciar os injetores lógicos.

Quanto ao pacote `MONITOR`, o uso do *toolkit* Javassist é similar. O objeto associado aos objetos do nível base é o meta-objeto `SensorFisico`. Este meta-objeto faz a interceptação com o objetivo único de executar o monitoramento. Os dados obtidos são repassados ao controlador, que os registra em um arquivo de *log*. O pacote `ATIVADOR` usa das classes de Javassist para tornar reflexivas as classes que foram especificadas na configuração e também para ativar o sistema alvo. O diagrama de funcionamento de Jaca pode ser vista na figura 2.

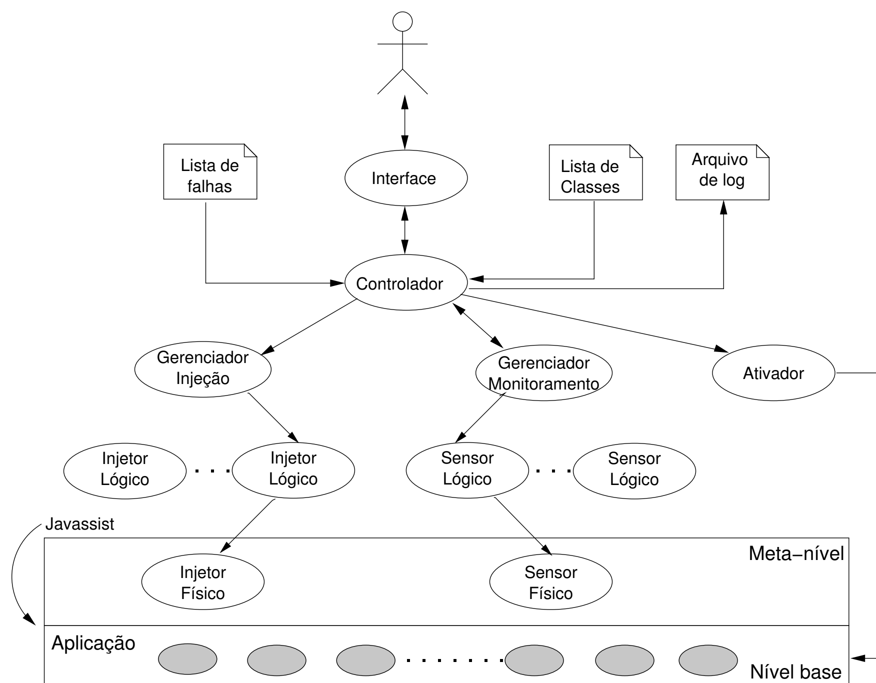

Figura 2: Diagrama de funcionamento da ferramenta Jaca

4. Extensão de Jaca para Falhas de Comunicação

A ferramenta Jaca, em sua primeira versão, injeta três tipos de falhas: falhas de atributos, falhas de parâmetro e falhas de retorno de métodos. Considerando as três classes disponíveis, a dificuldade para testar sistemas distribuídos é consideravelmente grande, uma vez que a latência para uma destas falhas se manifestar como um erro no sistema de troca de mensagens é muito alta. Para facilitar a execução de testes neste tipo de aplicações, Jaca foi estendida para modelos de falhas relativos a comunicação.

Os modelos escolhidos para a extensão são modelos para falhas que podem ocorrer no protocolo UDP (*User Datagram Protocol*) e no protocolo TCP (*Transmission Control Protocol*). Ambos os protocolos estão disponibilizados através da interface de programação de Java. A principal vantagem de oferecer um injetor de falhas de comunicação onde as falhas são injetadas em um

nível de abstração mais alto é a portabilidade. A maioria dos injetores de falhas de comunicação existentes esbarram no problema da portabilidade, pois geralmente são ligados diretamente ao sistema operacional.

A extensão do modelo de falhas de Jaca foi bastante facilitada pelo fato desta ferramenta ser baseada no sistema de padrões descrito na seção 2.3. O padrão de projeto *Injetor* já prevê a expansão do modelo. Com isso, a extensão da ferramenta passou por três fases: o estabelecimento dos modelos de falhas, a decisão do momento em que a falha deve ser injetada e a implementação de cada uma das falhas do modelo. As seções a seguir descrevem cada uma destas fases.

4.1. Modelos de Falhas

Falhas são fenômenos aleatórios, imprevisíveis e que podem levar um sistema a um estado errôneo. Se os erros não são tratados, o sistema pode apresentar defeitos. Um defeito ocorre sempre que um serviço não é prestado de acordo com a sua especificação. Dependendo do tipo de falha, ela recebe uma classificação. Em um contexto distribuído, as falhas que são considerados são baseados no modelo de falhas para sistemas distribuídos definido por Cristian [Cristian 1991].

Este modelo descreve falhas de omissão, temporização, resposta e colapso. Uma falha de omissão ocorre quando um servidor não responde a uma requisição. Uma falha é considerada de temporização quando uma resposta do servidor ocorre fora do intervalo de tempo especificado, podendo ser tanto uma resposta cedo demais quanto tardia demais. A última geralmente é associada a falhas de desempenho. A falha de resposta ocorre quando o servidor responde incorrentamente, tanto podendo o valor de retorno de uma requisição ser incorreto quanto uma transição de estado ocorrer incorretamente. Uma falha por colapso ocorre quando o servidor pára totalmente de responder. Este tipo de falha ainda pode ser classificada em subtipos, levando em consideração o estado do servidor após a sua recuperação. Observa-se que a causa específica de cada falha não interessa ao modelo, pois uma aplicação distribuída reage a falhas que afetam a troca de mensagens ou a queda de servidores para manter o sistema distribuído livre de defeitos.

O modelo de falhas para UDP é o mais detalhado. Os tipos de falhas consideradas são: omissão, temporização e colapso. São incluídas também falhas de ordenamento, que é um subtipo de falhas de temporização, e inserção de mensagens espúrias e duplicação de mensagens, ambas subtipos de falhas de resposta. A escolha por este modelo vem do fato de estes tipos de falhas serem bastante comuns em ambientes distribuídos. Além disto, nenhuma destas falhas é tratada pelo próprio protocolo UDP. Com isso, todas essas falhas podem ser diretamente percebidas pela aplicação, que deve estar preparada para sua detecção e correção.

O modelo TCP inclui apenas falhas de colapso, pois assume-se que é a única falha que pode chegar até a aplicação. Falhas de omissão, ordenamento, inserção de mensagens, duplicação e temporização são todas já mascaradas pelo protocolo. Como o objetivo desse injetor não é o teste dos mecanismos de tolerância a falhas do próprio protocolo, mas sim da aplicação que usa o protocolo, estas falhas não são consideradas.

4.2. Ativação da Falha

Uma abordagem comumente usada para ativar falhas de comunicação é disparar a falha no envio de uma mensagem ou na solicitação de uma requisição remota, no caso de uma chamada de método remoto. Considerando que deseja-se injetar falhas de comunicação na linguagem Java, a interceptação para realizar a injeção deve ser na interface de programação dos recursos de redes, como nos *sockets* TCP e UDP e no procurador do método remoto (*stub*).

No caso de *sockets* UDP, a classe que permite o envio de mensagens é `java.net.DatagramSocket`, através do método `send`. Portanto, é na execução deste

método que a interceptação deve ocorrer. Para a execução de um método ser interceptada, a classe deve ser reflexiva, ou seja, todos os objetos desta classe devem ter um meta-objeto associado. Com isso, toda execução de um método deste objeto é interceptada e assim a falha pode ser injetada ou não, baseada no arquivo de configuração de falhas. Entretanto, o carregador de classes de Javassist que torna as classes reflexivas não permite a reflexão de classes de sistema.

Para contornar este problema foi criada uma classe *wrapper* para a classe `java.net.DatagramSocket`. No caso, esta classe *wrapper* é uma subclasse da classe de sistema, que reimplementa o método `send` e pode ser reflexionada pelo carregador de classes de Javassist. A classe `java.net.DatagramSocket` pode ter subclasse pois não é declarada com a palavra chave `final`. Um exemplo simplificado de classe *wrapper* pode ser visto na figura 3. O atributo boleano é usado para decidir se a mensagem deve ser enviada ou não. Este atributo é necessário pois, no momento que um determinado método é interceptado, este deve ser executado até o seu final. Se este não for executado, a pilha de execução não é corretamente desfeita e o programa não consegue prosseguir de forma correta.

```
public class DatagramSocket_ extends java.net.DatagramSocket
{
   boolean envia;

   // Métodos para definição do atributo 'envia'

   public void send(java.net.DatagramPacket packet)
                  throws java.io.IOException
   {
      if ( envia == true )
         super.send(packet);
      else
         return;
   }
}
```

Figura 3: Classe *wrapper* para a classe de sistema

O emprego de classes *wrapper* já insere uma nova condição na ferramenta Jaca, que antes não requisitava o código fonte das aplicações para executar os experimentos de injeção de falhas. Neste caso, o usuário da ferramenta poderia usar de *scripts* para fazer a alteração do programa, redefinindo o nome das classes de `java.net.DatagramSocket` para `DatagramSocket_`. Para casos onde o código fonte não está disponível ou o usuário não deseja alterar código, pode-se alterar diretamente o *bytecode* das classes. A tabela de símbolos deve ser mudada, trocando a referência da classe de sistema para a referência da classe *wrapper*. Para isso, será desenvolvido um programa que use um *toolkit* que permita fazer alteração de *bytecodes*.

O segundo problema é relativo à ativação das falhas em programas que usam *sockets* TCP. Quando se usa *sockets* UDP, a única forma de enviar mensagens é pelo método `send` da classe `java.net.DatagramSocket`. Porém, quando se trata de *sockets* TCP, há uma vasta gama de opções para se escrever dados em um *socket*, inclusive classes que fazem operações de entrada e saída. Com isso surgem duas dificuldades: (i) a implementação de uma classe *wrapper* para cada uma das possíveis classes, e (ii) a diferenciação entre escritas em um arquivo local e em um *socket*. Tais problemas ainda estão sendo investigados, e por estas razões tal modelo ainda não foi implementado.

Está prevista também a extensão de Jaca para a injeção de falhas no protocolo RMI (*Remote Method Invocation*). Para isso, as chamadas remotas devem ser interceptadas, o que não deve apresentar dificuldades adicionais, visto que estas chamadas podem ser interceptadas durante a execução de métodos da classe *stub*. As classes *stub* são geradas pelo compilador RMI (`rmic`)

e não são consideradas classes de sistema. Portanto, podem ser reflexionadas pelo carregador de classes do Javassist.

4.3. Implementação do Modelo UDP

Apesar do protocolo UDP não ser confiável, este é comumente usado como base para a construção de mecanismos de tolerância a falhas, onde a confiabilidade requisitada é implementada nas camadas superiores. Um exemplo é o sistema de comunicação de grupo JGroups [Ban 1998], usado como bloco básico para construção de aplicações de alta disponibilidade. JGroups usa UDP como padrão na base de sua pilha de protocolos, sendo as camadas superiores da pilha que implementam a segurança de funcionamento da troca de mensagens. Neste caso, um injetor de falhas de comunicação para o protocolo UDP é fundamental da validação do funcionamento correto destas camadas.

Para adicionar um novo tipo de falha em Jaca precisam ser alterados dois pacotes: `GER_FALHAS` e `INJETOR`, destacados na figura 1. No pacote `GER_FALHAS` tem-se que adaptar o gerenciador para interpretar no arquivo de lista de falhas este novo tipo de falha adicionado. No pacote `INJETOR` inicia-se o processo de extensão pela adaptação de `GerenciadorDeInjecao`. Este é responsável pela instanciação de cada um dos injetores lógicos requisitados durante o experimento. A classe `InjetorFisico` também deve ser alterada, para estar preparada para injetar a nova falha. As duas classes que devem ser adicionadas são o injetor especializado para a nova falha e também a classe que descreve e guarda os dados da falha. Alterando e adicionando estas classes, Jaca está pronta para injetar o novo tipo de falha especificada. Até o momento foram adicionadas falhas de omissão, colapso e duplicação de mensagens.

No caso de falhas de omissão para o protocolo UDP foram criadas duas classes: `FalhaUdpOmissao` e `InjetorFalhaUdpOmissao`. A primeira é a classe que armazena os dados da falha e a segunda é o injetor lógico. Os atributos armazenados na classe `FalhaUdpOmissao` incluem o endereço IP de origem e de destino, a porta da conexão na origem e no destino, e também a taxa em que ocorrerá uma falha, caso ela seja configurada como intermitente. Se o usuário desejar configurar uma falha deste tipo é necessário adicionar no arquivo de lista de falhas a seguinte linha:

```
FalhaUdpOmissao:<padrao_de_repeticao>:<início>:<taxa_de_falha>:
<host_origem>:<porta_origem>:<host_destino>:<porta_destino>:
```

O padrão de repetição determina se a falha é transiente (`t`), permanente (`p`) ou intermitente (`i`). O início do processo de injeção também é configurável e é determinado a partir de qual mensagem (n-ésima mensagem) as falhas começarão a ser injetadas. O *socket* no qual a falha será injetada é identificado pelo par nome do *host* e porta na origem com nome do *host* e porta no destino. A configuração da origem é opcional e é interessante ser usada quando um *host* possui mais de uma interface de rede. Como um *socket* UDP não tem conexão, os endereços de destino são verificados diretamente no datagrama que está sendo enviado.

A implementação do novo modelo de falhas mostrou que o uso de uma ferramenta de arquitetura extensível é viável. A extensão de uma ferramenta pré-existente tem um custo de desenvolvimento bem menor que o projeto e a criação de uma nova ferramenta. No caso de Jaca, a extensão também foi facilitada por ser uma ferramenta baseada em um padrão de *software* bem documentado que já previa a expansão do modelo de falhas. Em Jaca, é necessário, pelo menos, a criação de duas classes para cada tipo de falha incorporada. A adaptação de outras classes é simples, como a classe de gerenciamento de falhas, que exige apenas a alteração do interpretador do arquivo de falhas.

4.4. Análise de Intrusão

Um aspecto importante de um injetor de falhas é a intrusão temporal causada pelo mesmo. Para fazer esta medição, foi montado o cenário de um cliente e um servidor, onde apenas o cliente

envia mensagens com o protocolo UDP. Os experimentos foram executados em dois computadores AMD Athlon XP 2000+ com o sistema operacional Linux e 512MB de memória RAM. A máquina virtual usada foi a disponibilizada pela Sun, em sua versão 1.4.1. Os tempos foram medidos com o utilitário `time(1)`, que imprime o tempo de CPU em modo usuário e modo sistema usado por um determinado processo. As falhas foram injetadas no cliente de forma intermitente, seguindo o modelo de omissão. A taxa de falhas escolhida foi de 20%. Foram medidos os tempos com o cliente executando com e sem o injetor de falhas. No primeiro experimento, o cliente envia ao servidor 1 milhão de mensagens, com tamanho de 512 *bytes*. No segundo, são enviadas 10 milhões. Cada experimento foi repetido cinco vezes. A tabela 1 apresenta os tempos médios obtidos em segundos.

	Experimento 1		Experimento 2	
	modo usuário	modo sistema	modo usuário	modo sistema
sem injetor	1,911	2,795	17,239	26,086
com injetor	2,260	2,462	17,580	25,341

Tabela 1: Resultados de testes de intrusão temporal em segundos

Como pode ser visto na tabela, a intrusão apresentada por Jaca estendida com falhas de omissão é baixa. Os testes realizados também indicam que a intrusão não é crescente a medida que aumenta o número de mensagens, e que o custo de instanciação de Jaca tende a diminuir quando o tempo de execução é maior. Outro dado obtido é a diferença entre o tempo de execução em modo usuário e em modo sistema. Quando o programa executa com falhas sendo injetadas, o tempo de usuário é maior, pois há o tempo de gerenciamento do injetor e o de processamento das falhas, onde o programa é interceptado a cada envio de mensagem. No entanto, o tempo de sistema é menor, pois quando uma falha de omissão ocorre não há efetivamente o envio da mensagem, evitando a troca para modo sistema e o custo do envio em si.

5. Conclusão

O artigo apresentou a extensão do injetor de falhas Jaca para modelos de falhas de comunicação. Jaca é baseada em um sistema de padrões que já previa a expansão dos modelos de falhas, o que facilitou a adição de novas falhas. Em sua primeira versão, Jaca injetava falhas de atributos, parâmetros e retorno de métodos. Com esta extensão, Jaca também pode ser usada para conduzir experimentos de testes em aplicações distribuídas, uma vez que pode atuar diretamente na interface de programação de *sockets* Java.

Jaca baseia sua implementação em reflexão computacional, que apresenta várias vantagens para implementação de injetores de falhas por *software*. A principal é a divisão clara entre a aplicação e os mecanismos de injeção de falhas e monitoramento. A escolha do protocolo de reflexão Javassist também permitiu a manutenção da portabilidade de Jaca, pois não é necessário o uso de uma máquina virtual modificada.

Trabalhos futuros incluem a incorporação de um modelo de falhas para RMI e o uso de Jaca extendido para validar experimentalmente o sistema de comunicação de grupo JGroups [Ban 1998], que vem sendo bastante usado como bloco básico de aplicações distribuídas de alta disponibilidade.

Referências

Aidemark, J.; Vinter, J.; Folkesson, P.; Karlsson, J. *GOOFI: Generic Object-Oriented Fault Injection Tool*. In Proceedings of DSN'2001. Gotemburgo, Suécia. Julho 2001.

Ban, B. *JavaGroups - Group Communication Patterns in Java*. Department of Computer Science, Cornell University. Julho 1998.

Barcelos, P. P. A.; Leite, F. O.; Weber, T. S. *Building a Fault Injector to Validate Fault Tolerant Communication Protocols*. In Proceedings of International Conference on Parallel Computing Systems. Ensenada, México. Agosto 1999.

Carreira, J.; Madeira, H.; Silva, J. G. *Assessing the Effects of Communication Faults on Parallel Applications*. In Proceedings of IPDS'95. Erlangen, Alemanha. Abril 1995.

Carreira, J.; Silva, J. G. *Why do Some (weird) People Inject Faults?* ACM SIGSOFT, Software Engineering Notes, Volume 23, Número 1, pp. 42-43. Janeiro 1998.

Chiba, S. *Load-time Structural Reflection in Java*. In Proceedings of ECOOP 2000 - Object-Oriented Programming. LNCS 1850, Springer Verlag, pp. 313-336. 2000.

Cristian, F. *Understanding Fault-Tolerant Distributed Systems*. Communications of the ACM, Volume 34, Número 2, pp. 56-78. Fevereiro 1991.

Dawson, S.; Jahanian, F.; Mitton, T. *ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementations*. In Proceedings of IPDS'96. Urbana-Champaign, Estados Unidos. Setembro 1996.

Gonçalves, L. C. R.; Rodegheri, P. R.; Manfredini, R. A.; Weber, T. S. *Testing Fault Tolerance Mechanisms in DBMS Through Fault Injection*. In Proceedings of IEEE Latin-American Test Workshop. Cancun, México. Fevereiro 2001.

Hsueh, M.-C.; Tsai, T.; Iyer, R. *Fault Injection Techniques and Tools*. IEEE Computer, Volume 30, Número 4, pp. 75-82. Abril 1997.

Iyer, R. K. *Experimental Evaluation*. In Proceedings of FTCS-25. Pasadena, Estados Unidos. Junho 1995.

Leme, N. G. M; Martins, E.; Rubira, C. M. F. *A Software Fault Injection Pattern System*. In Proceedings of PLoP 2001. Monticello, Estados Unidos. Setembro 2001.

Maes, P. *Concepts and Experiments in Computational Reflection*. In Proceedings of OOPSLA 1987. Orlando, Estados Unidos. Outubro 1987.

Martins, E.; Rosa, A. C. A. *A Fault Injection Approach Based on Reflective Programming*. In Proceedings of DSN'2000. Nova York, Estados Unidos. Junho 2000.

Martins, E.; Rubira, C. M. F.; Leme, N. G. M. *Jaca: A Reflective Fault Injection Tool Based on Patterns*. In Proceedings of DSN'2002. Washington, Estados Unidos. Junho 2002.

Rosenberg, H. A.; Shin, K. G. *Software Fault Injection and its Application in Distributed Systems*. In Proceedings of FTCS-23. Toulouse, França. Junho 1993.

Schmidt, D. C. *Using Design Patterns to Develop Reusable Object-Oriented Communication Software*. Communications of the ACM, v. 38(10), pp. 65-74. Outubro 1995.

Stott, D. T.; Floering, B.; Burke, D.; Kalbarczyk, Z.; Iyer, R. K. *NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors*. In Proceedings of IPDS'2000. Chicago, Estados Unidos. Março 2000.

Voas, J. *Software Fault Injection: Growing 'Safer' Systems*. In Proceedings of IEEE Aerospace Conference 1997, v. 2, pp. 551-561. Aspen, Estados Unidos. Fevereiro 1997.