Um Injetor de Falhas de Comunicação construído usando Programação Orientada a Aspectos*

Karina Kohl Silveira, Taisy Silva Weber

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

{kohl,taisy}@inf.ufrgs.br

***Resumo**: Aplicações de alta dependabilidade devem estar aptas a detectar falhas e corrigir erros para que sejam evitados danos irreparáveis a vida, ao meio ambiente ou aos negócios. Uma técnica usada para garantir operação adequada de mecanismos tolerantes a falhas é a injeção de falhas. FICTA é uma ferramenta de injeção de falhas de comunicação para a validação de aplicações distribuídas baseadas em UDP e desenvolvidas em Java. FICTA utiliza Programação Orientada a Aspectos, a qual permite que comportamentos afetando diversas classes do sistema, os chamados interesses cruzados, sejam reunidos em um único módulo chamado aspecto. A abordagem baseada em AOP permitiu a construção de uma ferramenta altamente modular, reusável e que não causa intrusividade espacial nas aplicações alvo.*

***Abstract**: High dependable applications detect faults and correct errors to avoid risks to life, environment and business that depend on them. Fault injection is an efficient technique to test fault tolerance mechanisms. We develop FICTA, a fault injector to validate UDP-based distributed network applications programmed in Java. FICTA uses aspect oriented programming, that permits behavior that affects a variety of classes to be joined in a single module called aspect. The AOP-based approach allows the implementation of a high modular and reusable tool that is not intrusive in target applications.*

1 Introdução

Aplicações distribuídas estão sujeitas a falhas que afetam o sistema de troca de mensagens na rede de comunicação. Para alcançar dependabilidade, é necessária a implementação de mecanismos de detecção e remoção das falhas que afetam mensagens ou nós do sistema. Além disso, é necessário que esses mecanismos sejam validados para assegurar que satisfazem sua especificação. A ausência de testes desses mecanismos pode levar a comportamentos inesperados e errôneos na fase operacional da aplicação. Uma etapa fundamental no desenvolvimento destes sistemas é a fase de validação sob falhas, na qual é observado o comportamento da aplicação em vários cenários de falhas emuladas.

A injeção de falhas pode ser definida como a introdução intencional e controlada de falhas em uma aplicação alvo para observar seu comportamento [ARL90]. As falhas injetadas permitem avaliar se a aplicação sob teste se comporta conforme especificado na presença de falhas, além de determinar se a cobertura de falhas dos mecanismos

* Financiado pelos projetos ACERTE/CNPq (472084/2003-8) e DepGriFE/HP Brasil P&D

implementados está adequada para a confiabilidade e a disponibilidade exigidas. Para esse trabalho interessa a injeção de falhas implementada por software, onde falhas são injetadas por software corrompendo código ou dados ou alterando o fluxo de operações.

A crescente necessidade de tornar aplicações distribuídas cada vez mais confiáveis exige que seja ampliado o uso da técnica de injeção de falhas. Para isso é necessário o estudo de abordagens que levem a soluções modulares, reusáveis e que causem o mínimo de intrusividade espacial nas aplicações alvo, visto que essas características facilitam e ampliam a utilização de injeção de falhas.

Baseado nessas necessidades, esse artigo apresenta uma nova abordagem, baseada em Programação Orientada a Aspectos [KIC97] para a validação de aplicações distribuídas escritas em Java baseadas no protocolo UDP. A abordagem se baseia na instrumentação de métodos de envio e recepção de mensagens dos protocolos usados pela aplicação, simulando falhas de comunicação. O artigo apresenta conceitos de injeção de falhas por instrumentação de código e de orientação a aspectos, discute como a orientação a aspectos pode auxiliar no desenvolvimento de injetores de falhas, apresenta o protótipo de uma ferramenta construída com essa abordagem e demonstra sua utilização em um caso exemplo. Além disso, a abordagem e a ferramenta apresentadas nesse trabalho se preocupam em evitar a intrusividade espacial, comum em algumas abordagens de injeção de falhas baseadas em *software*. Finalmente, são tecidas algumas considerações sobre os resultados alcançados.

2 Redução da intrusividade espacial através de instrumentação de código

A interferência do injetor de falhas da aplicação alvo, chamada de intrusividade, pode ser observada de forma temporal e espacial [DAW96]. Na intrusividade temporal, o tempo de execução da aplicação é aumentado devido ao acréscimo das atividades do injetor, que são executadas juntamente com a aplicação alvo. A espacial surge da modificação do código da aplicação alvo. A intrusividade temporal é crítica principalmente em aplicações de tempo real, porém a intrusividade espacial é uma preocupação para todas as classes de aplicação, pois alterações no código fonte podem espalhar *bugs* e mesmo alterar o fluxo de execução da aplicação. Adicionalmente, não é raro que a totalidade ou uma parte considerável do código fonte não esteja disponível para teste.

A instrumentação de código permite que módulos de programas sejam completamente reescritos em tempo de execução facilitando a introdução de novas funcionalidades. É possível a introdução ou remoção de instruções, alterando o uso de classes, variáveis e constantes. Esse trabalho usa instrumentação de código para a inserção de falhas nas aplicações alvo. Várias técnicas vêm sendo estudadas para evitar a alteração do código original da aplicação, como por exemplo, reflexão computacional e programação orientada aspectos.

A reflexão computacional oferece flexibilidade à instrumentação de código, pois permite alteração nas classes de uma aplicação independente da disponibilidade do código fonte. Porém, dependendo da ferramenta de reflexão utilizada, uma cláusula de instanciação deve ser declarada ao longo da declaração da classe base (por exemplo, a ferramenta OpenJava), introduzindo sobrecarga e custos em manutenção e alteração [ROY2003].

A instrumentação por AOP permite a interceptação e instrumentação de elementos de classes, necessitando apenas o conhecimento da API das aplicações. Além disso, preserva a

integridade da classe base, isto é, não são necessárias mudanças estruturais na base. A AOP possui elementos que identificam claramente os nomes de cada método em cada classe que está envolvida no processo de cruzamento, dessa forma o programador pode identificar facilmente a parte do código que necessita ser atualizada após a chamada de cada método [ROY2003]. A principal vantagem do uso de AOP em relação à reflexão computacional é que sua funcionalidade é plugável. Se uma funcionalidade necessita ser removida do sistema, basta que se remova o aspecto responsável pela funcionalidade. Ao contrário da ferramenta de reflexão computacional OpenJava, que cria a necessidade de varrer cada classe afetada e remover as cláusulas de instanciação

3 Injeção de falhas de comunicação

Uma ferramenta de injeção de falhas por *software* geralmente é um trecho de código que usa todos os ganchos possíveis do processador para criar um comportamento errôneo de maneira controlada [CAR98]. Para esse trabalho, interessam as falhas de comunicação que podem ocorrer em sistemas distribuídos suportados por uma infra-estrutura de rede. Estas afetam as mensagens que são transmitidas por um canal de comunicação, que podem ser omitidas, duplicadas ou entregues com atraso.

Diversas ferramentas de injeção de falhas têm sido desenvolvidas [HSU97]. A maior parte destas ferramentas é específica para um determinado sistema alvo, não permitindo reuso. Várias são experimentos acadêmicos e não estão disponíveis para avaliação ou aplicação. Poucas dessas ferramentas ([DAW96], [JAC2004a]) são injetores de falhas de comunicação. Jaca e FIONA são ferramentas bastante próximas a FICTA. Jaca [MAR2002] é uma ferramenta extensível com a finalidade de validar aplicações em Java. Jaca é baseada em reflexão computacional e sua arquitetura segue um padrão de software projetado para ferramentas de injeção de falhas. Como plataforma para reflexão computacional, Jaca utilizada o protocolo de meta objetos não padrão Javassist, que permite que os *bytecodes* sejam transformados durante o tempo de carga. Jaca teve seu modelo de falhas estendido para injetar falhas de comunicação UDP [JAC2004b]. Essa extensão necessita uma fase de pré-processamento do código da aplicação (código fonte ou *bytecode*) para a injeção de falhas. Isto é necessário para que a aplicação possa se referir às classes de comunicação que suportam a injeção de falhas. FIONA [JAC2004a] tem um comportamento semelhante à extensão de Jaca, mas utiliza JVMTI (*Java Virtual Machine Tool Interface*). JVMTI é uma nova interface de programação nativa da plataforma Java para o desenvolvimento de ferramentas de monitoramento e depuração, permitindo a instrumentação de aplicações Java. O uso de JVMTI faz que não seja necessária a alteração da máquina virtual nem da aplicação, pois é uma biblioteca dinâmica que é carregada opcionalmente pela máquina virtual quando é inicializada.

A abordagem desse trabalho diferencia-se de FIONA, Jaca e também das demais ferramentas de injeção de falhas de comunicação conhecidas por usar AOP para realizar a interceptação e instrumentação de *bytecodes* das classes que implementam os métodos de comunicação das aplicações alvo distribuídas. AOP é um paradigma de programação relativamente novo, complementar à orientação a objetos e que tem como objetivo a melhor modularização de interesses que cruzam várias classes de um sistema. A utilização de abordagem AOP permite que os aspectos sejam carregados junto com a aplicação alvo (diz-se que os aspectos e as classes são integrados em tempo de carga) e a instrumentação das primitivas de comunicação é realizada efetivamente em tempo de execução.

4 Programação orientada a aspectos

AOP separa interesses que afetam diversos módulos de um sistema, em unidades únicas chamadas aspectos. Esses ”interesses cruzados” são chamados *crosscutting concerns*. Um *concern* é um comportamento específico, um conceito ou uma área de interesse. Como exemplo pode ser citada a manipulação de mecanismos de *logging*, autenticação, segurança, persistência, etc. Os *crosscutting concerns* tendem a afetar vários módulos de implementação de um sistema, isto é, seus códigos se espalham em vários módulos, dificultando a sua manutenção e compreensão [LAD2002].

Normalmente, as diversas implementações para AOP utilizam construções chamadas *advices* para descrever o código que deve ser inserido em localizações chamadas de *join points*. O código pode ser inserido antes ou depois dos *join points*. Além disso, um *around advice* permite que o desenvolvedor especifique código que deve substituir o código original que é executado naquele *join point*. Também existem as construções chamadas de *pointcuts,* que especificam em quais *join points* o código deve ser executado. Um aspecto, por definição, é uma unidade modular, que encapsula os conceitos anteriores e que corta a estrutura de outras unidades. Um aspecto é similar a uma classe tendo um tipo, estendendo outras classes e outros aspectos. Ele pode ser abstrato ou concreto e ter campos, métodos e tipos como membros. A AOP permite a implementação individual de comportamentos, de forma fracamente acoplada, e a combinação dessas implementações para formar o sistema final. De fato, cria um sistema usando implementações modulares, fracamente acopladas, de comportamentos cruzados.

4.1 Orientação a aspectos versus orientação a objetos

A unidade modular de AOP é o aspecto [LAD2002]. Na Programação Orientada a Objetos, OOP, a unidade natural modular é a classe, e um comportamento cruzado está espalhado em várias classes. Trabalhar com código que aponta para responsabilidades que atravessam o sistema gera problemas que resultam na falta de modularidade. A AOP complementa a OOP por facilitar um outro tipo de modularidade que expande a implementação espalhada de um comportamento dentro de uma simples unidade. Como resultado, um único aspecto pode contribuir na implementação de procedimentos, módulos ou objetos, aumentando a reusabilidade dos códigos.

O processo de criação e integração de aspectos com as classes que devem ser afetadas por esses aspectos é chamado de *weaving*. O *weaving* realiza o processo de instrumentação das classes, modificando o *bytecode* de modo a incluir interfaces e chamadas aos *advice*s. Os integradores de aspectos (*aspect weavers*) devem processar as classes componentes dos sistemas com os aspectos, compondo-os de forma apropriada para produzir a operação total desejada do sistema.

O processo de *weaving* pode ser estático ou dinâmico. Com o *weaving* estático, os aspectos são integrados resultando em uma única classe, os aspectos são entidades em tempo de compilação. Em *weaving* dinâmico, os aspectos sempre são classes separadas, que são chamadas por algum *framework* que intercepta dinamicamente o uso de algum objeto, como é realizado, por exemplo, pelo AspectWerkz [BON2004]. Nesse caso, os aspectos são entidades em tempo de execução.

4.2 Um framework para AOP

AspectWerkz é um *framework* dinâmico de programação orientada a aspectos, para Java, e foi usado como base na implementação de FICTA. Qualquer classe Java pode ser um aspecto, ou seja, pode ser definida para ter um comportamento que corta outras classes, significando que ela se torna uma unidade modular para *crosscutting concerns*. Não há necessidade de estender nenhuma classe em especial ou implementar interfaces específicas. Os aspectos podem ser definidos utilizando XML ou anotações. Para esse trabalho importam aspectos definidos em arquivos XML, pois se tornam mais apropriados para construir aspectos para injeção de falhas sem intrusividade e de fácil configuração, adição e remoção.

AspectWerkz trabalha com *weaving* dinâmico e para isso oferece mecanismos chamados de modo *offline* e modo *online*. No modo *offline* as classes sofrem uma pós-compilação antes da sua execução para que os aspectos sejam integrados nas aplicações, nesse caso são necessárias duas fases para gerar as classes. A primeira fase é uma compilação padrão utilizando o compilador padrão Java (`javac`). Na segunda fase é executado o compilador do AspectWerkz no modo *offline,* apontando para as novas classes criadas. O compilador modifica os *bytecodes* das classes incluindo os *advices* nos *join point* corretos. Esse processo assemelha-se aos mecanismos de *weaving* estático, necessitando que todas as classes sejam recompiladas para que os aspectos sejam efetivamente integrados as classes.

No modo *online*, as classes são modificadas durante sua execução de forma transparente. Para o modo *online*, AspectWerkz está preso ao mecanismo de baixo nível de carregamento de classes da JVM, permitindo interceptar todas chamadas de carregamento de classes e transformando os *bytecodes* em tempo de execução. Para isso, AspectWerkz modifica o mecanismo de carga de classes da JVM utilizando *HotSwap*, no caso de Java 1.4, ou JVMTI no caso de Java 1.5. De forma simplificada, tanto *HotSwap*, quanto JVMTI encapsulam a habilidade de substituir o código instrumentado em uma aplicação que está executando, através do mecanismo de depuração ou de processos chamados de agentes.

O *framework* AspectWerkz foi escolhido pelo seu mecanismo de *weaving* dinâmico *online.* Para uma abordagem de injeção de falhas, que visa evitar intrusividade, essa característica é muito importante. Os sistemas sob teste nem sempre possuem seu código disponível para ser alterado e recompilado. Além disso, a recompilação de todas as classes do sistema alvo pode ser um processo oneroso, ou seja, o *weaving* estático não seria uma boa solução. O *framework* AspectWerkz permite que os aspectos alterem os *bytecodes* das classes alvo em tempo de carga ou de execução, atingindo todas as classes que se deseja instrumentar, sem necessidade de serem compiladas especificamente com a ferramenta do *framework.*

5 Uso de AOP em injeção de falhas

AOP é uma excelente alternativa para impedir a alteração do código fonte das aplicações a serem testadas, evitando assim intrusividade espacial. A AOP oferece um mecanismo para a injeção de falhas de alto nível em sistemas orientados a objetos, possibilitando a instrumentação de métodos, classes e variáveis de um sistema, em tempo de compilação, carga ou execução, sem a necessidade de alterar a estrutura do sistema, tornando indiferente a disponibilidade do código fonte para o teste. Dessa forma, é necessário apenas o mínimo de informação sobre a aplicação, como classes, métodos e nomes de atributos. Com o uso

de AOP a instrumentação é encapsulada em um aspecto e é introduzida em nível de *bytecodes*. O aspecto identifica as chamadas aos métodos que devem ser modificados e instrumenta todo o código ou parte dele em tempo de carga e execução. Além disso, o injetor de falhas pode ser facilmente inserido e removido da aplicação a ser validada, apenas pela remoção dos aspectos relativos à injeção de falhas (simplesmente sem carregar o aspecto no momento que a aplicação é disparada).

5.1 Recursos de AOP para injeção de falhas

A injeção de falhas possui um comportamento que abrange os diversos módulos da aplicação alvo, afetando métodos que são executados em diversas classes em diversos pontos da aplicação. Desta forma, a injeção de falhas pode ser encapsulada sob a forma de aspectos. Os métodos da aplicação que devem sofrer a injeção de falhas quando executados, se encaixam nos conceitos de *join point* e *pointcuts*, os quais sinalizam que aqueles determinados métodos poderão ter comportamento alterado e ou adicionado. Ao serem atingidos esses métodos (*join points*), eles tem seus comportamento alterado pela lógica de injeção de falhas que se encontra nos *advices* do aspecto. A abordagem básica deste trabalho é a implementação de aspectos que definem *pointcuts* em termos das primitivas de comunicação das aplicações e protocolos, e implementam *advices* que substituem os códigos originais por códigos instrumentados.

De acordo com Carreras [CAR98], uma ferramenta de injeção de falhas por software geralmente é um trecho de código que usa todos os ganchos possíveis do processador para criar um comportamento errôneo de maneira controlada. Dessa forma, mostra-se que o comportamento conjunto dos *join points* com os *pointcuts* é compatível com os ganchos definidos por Carreras. Além disso, os *advices* possibilitam a inserção do comportamento errôneo de forma controlada.

Para a injeção de falhas em aplicações distribuídas de rede, é necessário instrumentar as primitivas de comunicação. A AOP é uma alternativa viável e eficiente, pois torna possível a construção de aspectos que interceptam, em um único ponto, os métodos que são chamados de qualquer lugar da aplicação. Além disso, as classes da aplicação não precisam ser alteradas de forma alguma, nem ao mesmo para a inserção de cláusulas de instanciação ou inserção de bibliotecas adicionais. A união das classes da aplicação com os aspectos é realizada pelo *weaver* da ferramenta de programação orientada a aspectos que está sendo utilizada.

5.2 Vantagens da AOP para injeção de falhas

O encapsulamento da lógica de injeção de falhas em um único módulo que afeta as diversas classes do sistema, separando o código funcional do código de injeção, facilita a inserção e remoção dos aspectos na aplicação alvo e auxilia na criação rápida de diversos cenários de falhas. Além disso, caso o sistema deva ser executado sem o injetor de falhas (por já ter sido validado e colocado em produção, por exemplo), basta que se remova o aspecto responsável por essa funcionalidade, conservando intacto o programa original

Essa funcionalidade plugável (facilidade de inserção e remoção) oferecida pela AOP traz outro benefício importante na construção de injetores de falhas. Sistemas de missão crítica, que exigem alta disponibilidade, devem ser fortemente validados antes de serem colocados em produção, pois podem até mesmo envolver risco a pessoas (por exemplo, sistemas de controle aéreo). Porém, além desses, existem sistemas distribuídos sem um alto

grau de risco, mas que nem por isso podem deixar de ser validados. Muitas vezes por falta de tempo no desenvolvimento, ou na falta de ferramentas flexíveis que possam ser facilmente usadas em várias aplicações, a fase de validação dos mecanismos de tolerância a falhas fica relegada a um segundo momento, muitas vezes já na fase de manutenção. Um injetor de falhas, altamente reusável e facilmente (des)plugável é de valor inquestionável, pois amplia o uso da injeção de falhas como mecanismo de validação, fazendo que sistemas sejam mais facilmente validados e ofereçam maior segurança de uso.

6 A ferramenta FICTA

FICTA (*Fault Injection Communication Tool based on Aspects*) tem como principal objetivo a injeção de falhas de comunicação em aplicações Java distribuídas de rede, usando a Programação Orientada a Aspectos. Como resultado de um experimento de teste aplicando FICTA, a aplicação alvo Java, que está sendo validada e que tenha sido construída usando técnicas de tolerância a falhas, terá sua cobertura de falhas determinada para um determinado cenário de teste. Caso a cobertura não seja adequada, o desenvolvedor da aplicação pode refinar os mecanismos construídos. Cenários de falhas podem ser redefinidos e os experimentos podem ser repetidos tantas vezes quanto necessário para o teste dos mecanismos de tolerância a falhas da aplicação alvo.

6.1 Aplicações alvo para o injetor de falhas FICTA

O protocolo UDP (*User Datagram Protocol*) é um protocolo que envia pacotes de dados independentes, chamados datagramas, de um computador a outro, porém sem garantias sobre a chegada desses pacotes. O protocolo UDP é não confiável e não orientado a conexão, no entanto é comumente usado como base para a implementação de protocolos que implementam a confiabilidade necessária através de camadas superiores adicionais. Um exemplo é o *middleware* de comunicação de comunicação de grupo JGroups [BAN98] que usa UDP por padrão na base de sua pilha de protocolos.

FICTA implementa injeção de falhas de comunicação através da instrumentação da classe `java.net.DatagramSocket` da API padrão da linguagem Java. É essa classe que permite o envio e recebimento de pacotes através do protocolo UDP. A instrumentação de uma classe base da linguagem, garante a reusabilidade para toda e qualquer aplicação, protocolo ou *middleware,* escrito em Java e baseada no protocolo UDP. Portanto, qualquer aplicação Java construída sobre essa classe pode ser validada utilizando a ferramenta FICTA. Caso a ferramenta tivesse sido construída para instrumentar uma classe específica da aplicação alvo, não seria genérica o suficiente para garantir reusabilidade. No momento que se instrumenta uma classe da aplicação, restringe-se a ferramenta apenas aquela aplicação. Por aplicação entende-se desde sistemas desenvolvidos diretamente com a API, protocolos, *middlewares* e *toolkits* que se utilizam dessa classe para a construção de outros sistemas.

A classe Java `java.net.DatagramPacket` representa um datagrama UDP. Datagramas são utilizados para a entrega de pacotes em sistemas sem conexão e normalmente incluem o endereço destino e informação de porta. A classe `java.net.DatagramSocket` é um *socket* utilizado para o envio de datagramas em uma rede pelo protocolo UDP. Um datagrama é enviado por um `java.net.DatagramSocket` pela chamada do método `send(DatagramPacket)`. O método `receive(DatagramPacket)` é utilizado para realizar a recepção de um

datagrama. A classe `java.net.MulticastSocket` também pode ser utilizada para envio e recebimento de datagramas para um grupo *multicast*. Essa classe é subclasse de `java.net.DatagramSocket` e adiciona as funcionalidades para *multicasting*. Com FICTA também é possível validar protocolos que utilizam *multicast* UDP, já que a classe `java.net.MulticastSocket` é uma classe filha de `java.net.DatagramSocket` e não reimplementa os métodos *send* e *receive*. Ou seja, a injeção de falhas acontece transparentemente, através do mesmo mecanismo.

6.2 Modelo de falhas e ativação

FICTA considera as falhas de colapso de nós e omissão de mensagens. Omissão de mensagens pode representar tanto falhas no receptor ou emissor, assim como perdas de pacotes na rede de comunicação. Essas falhas não são tratadas pelo protocolo UDP e devem, portanto, ser consideradas no desenvolvimento de protocolos de mais alto nível usadas pela aplicação, ou na própria aplicação alvo.

Uma abordagem comumente usada para a ativação de falhas de comunicação é disparar a falha durante o envio ou recepção de uma mensagem [DAW96]. FICTA utiliza como ativador uma quantidade determinada de mensagens enviadas e recebidas no processo da aplicação distribuída que está sendo executado com o injetor de falhas. Após um determinado número de mensagens enviadas ou recebidas é disparada a falha. Esse número de mensagens é determinado pelo usuário que está realizando o experimento de teste ou pode ser gerado aleatoriamente pelo injetor. O injetor identifica se deve gerar os valores do experimento se receber um parâmetro de entrada chamado "`RANDOM`". Caso o usuário deseje especificar o percentual de mensagens que deve ser utilizado para que as falhas sejam disparadas, o parâmetro "`DETERMINISTIC`" deve ser utilizado. Caso os parâmetros "`RANDOM`" ou "`DETERMINISTIC`" não sejam especificados pelo usuário, é assumido como padrão o parâmetro "`RANDOM`" e gerado um valor pelo próprio protótipo.

6.3 Arquitetura da ferramenta FICTA

A ferramenta é composta por dois componentes: um aspecto e um arquivo XML. O aspecto contém os *advices* que possuem os códigos instrumentados com a lógica de injeção de falhas. O arquivo XML define os *pointcuts* e quais *advices* devem ser executados.

Cada tipo de falha considerada no modelo de falhas é implementado no aspecto e para realizar a injeção são realizadas interceptação e instrumentação de primitivas de comunicação do protocolo UDP. As classes que implementam os aspectos são carregadas da mesma forma que classes Java normais. O contêiner AOP fica responsável por diferenciar classes e aspectos e identificar em que momento os aspectos responsáveis pela instrumentação dos *bytecodes* devem executar no lugar do original. No caso do *framework* utilizado, AspectWerkz, essa função é realizada através da alteração do mecanismo de carga de classes do sistema para que seja possível essa identificação, conforme pode ser observado na Figura 1.

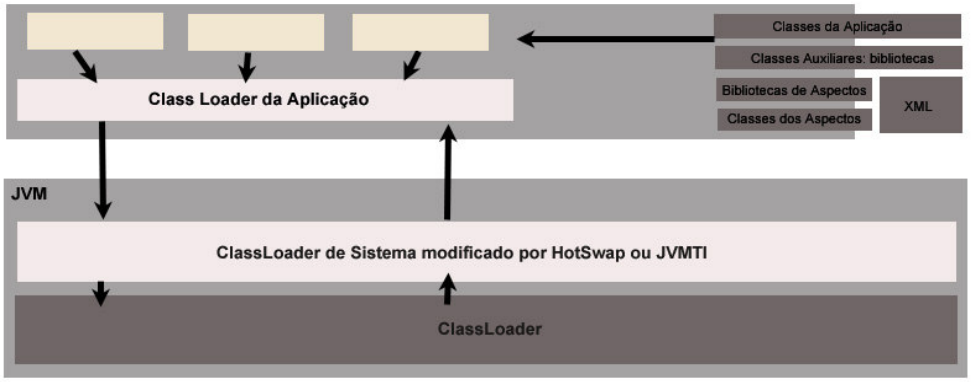

Figura 1 – Arquitetura de FICTA – ClassLoader de sistema modificado

Na integração de classes e aspectos, não é realizada uma instrumentação total da classe `java.net.DatagramSocket`, mas sim, são inseridos ganchos para a sinalização dos métodos que podem vir a ser instrumentados. Não é o *advice* com a lógica de injeção de falhas que é integrado na classe, mas sim o campo `JoinPoint`, definindo o *join point* especifico e uma chamada de método ao método `proceed`() da classe `JoinPoint`. Quando a classe alvo é instanciada, no caso a classe `java.net.DatagramSocket`, a instância do `JoinPoint` se registra no sistema e são inseridos os ganchos para ligar os *advices* aos *join points* que os definem.

6.4 Operação da ferramenta

A classe original `DatagramSocket` executa normalmente até que a aplicação alvo realize uma chamada a algum método que deva ser instrumentado. O contêiner AOP (*framework* AspectWerkz) identifica o *join point* (através do arquivo de definição em XML), recolhe informações sobre o mesmo, determinando quais os advices que estão ligados a ele e então passa o fluxo para o aspecto que implementa o advice responsável pela instrumentação do método. O *advice* verifica as condições que devem ser respeitadas para a injeção da falha através do método chamado `trigger()`. Esse método é privado do aspecto e realiza as verificações necessárias para definir se a falha deve ser injetada naquele momento ou não. Os cálculos são realizados a partir do número de mensagens que já foram enviadas ou recebidas pelo processo e também é levado em consideração se devem ser injetadas falhas de omissão ou colapso. Esses parâmetros são definidos em um arquivo de configuração. Após a execução do advice (método instrumentado), o fluxo é devolvido para o método original através do método proceed() da classe JoinPoint.

As falhas que devem ser ativadas durante a execução da aplicação, que compõem o cenário de falhas ou *faultload*, são especificadas em um arquivo texto. Esse arquivo é carregado juntamente com as classes e aspectos e as configurações são armazenadas em variáveis membros do aspecto. A Figura 2 ilustra um exemplo de como o arquivo pode ser configurado.

```
#parameters
perc=50
type=OMISSION
expType=DETERMINISTIC
```

Figura 2 – Exemplo de arquivo de configuração de FICTA

7 Aplicando FICTA em alvos reais

É importante ressaltar que esse artigo não visa uma validação de algum sistema específico, mas sim, o uso de algumas aplicações alvo como auxílio na prova de conceitos da abordagem apresentada.

Para demonstração da funcionalidade da ferramenta, foi escolhida uma aplicação distribuída cujo comportamento sob falhas fosse de fácil visualização. A mesma aplicação foi usada para demonstrar o uso da ferramenta FIONA [JAC2004a]. A aplicação alvo consiste de um quadro de anotações distribuído, onde cada vista do quadro reside em um diferente nó do sistema. Cada usuário pode fazer anotações na sua vista, essas anotações são distribuídas para todos os demais nós ponto a ponto. Se a aplicação não usa nenhum recurso de detecção e retransmissão de mensagens, uma mensagem omitida na origem representa um ponto em claro em todas as demais vistas, uma mensagem omitida no destino representa um ponto em claro apenas em uma vista. Um nó em colapso provoca a exclusão de participante no sistema, ou seja provoca uma alteração na visão de grupo.

Esse quadro de anotações distribuído usa o *middleware* de comunicação de grupo, JGroups. O JGroups permite que seja definida uma pilha de protocolos para que se atinja maior ou menor confiabilidade, de acordo com as necessidades do sistema que está sendo construído. Para a demonstração de FICTA, o quadro de anotações foi executado, definindo apenas o protocolo UDP na base de sua pilha de protocolos. No JGroups, o protocolo UDP é implementado como um invólucro para a classe `java.net.DatagramSocket` da API padrão Java. Dessa forma, FICTA atinge seus objetivos, que são injetar falhas instrumentando primitivas de comunicação.

Figura 3 – Experimento de injeção de falhas sem retransmissão

Para o primeiro experimento, definiu-se na pilha apenas o protocolo UDP, ou seja, nenhum mecanismo de tolerância a falhas. FICTA foi configurada para uma taxa de omissão de mensagens de 50%. A Figura 3 ilustra o protótipo agindo como esperado. A janela da esquerda corresponde ao processo onde as falhas estão sendo injetadas e onde o desenho está sendo construído. A omissão de mensagens é realizada no método `send` de `java.net.DatagramSocket`, impossibilitando o envio da mensagem para a rede, resultando na omissão da mensagem. A janela da direita possui um desenho incompleto, uma vez que as mensagens não estão sendo recebidas.

O mesmo teste foi realizado inserindo na pilha de execução do JGroups os protocolos que implementam os mecanismos de retransmissão e detecção de falhas. Quando um pacote UDP é perdido, é detectado pelos protocolos responsáveis, e então o pacote é retransmitido.

Os resultados desse teste mostraram que mesmo com falhas sendo injetadas, os desenhos em todas as janelas são idênticos. Esse comportamento é esperado, pois as falhas estão sendo tratadas. Com o auxílio desta aplicação alvo, foi facilmente visualizado que FICTA efetivamente injeta falhas de omissão e colapso. O quadro de anotações distribuído é um excelente caso de demonstração para ferramentas de injeção de falhas de comunicação.

8 Considerações Finais

Esse artigo apresentou uma nova abordagem para a construção de ferramentas de injeção de falhas baseada em programação orientada a aspectos. Também foi apresentada a ferramenta FICTA, um injetor de falhas, que utiliza a abordagem apresentada, e que tem como objetivo a validação de aplicações Java distribuídas construídas sobre o protocolo UDP e que implementam mecanismos de tolerância a falhas. O desenvolvimento da ferramenta mostrou que a abordagem é factível e eficiente e resulta em uma solução modular, reusável e sem intrusividade espacial nas aplicações alvo.

A opção pelo uso de aspectos surgiu ao se identificar a possibilidade de realizar a instrumentação do código da aplicação a ser validada sem a necessidade de alteração direta do código fonte, evitando a intrusividade do injetor de falhas no código do sistema alvo. Além disso, o uso de AOP trouxe ganhos em características como flexibilidade, modularidade e facilidade de manutenção e extensão. Quanto à flexibilidade, aspectos facilitam a introdução de funcionalidades de forma não pervasiva, permitindo que a arquitetura desenvolva-se sem intrusividade sobre o código. Quanto a modularidade, aspectos permitem o projeto de componentes modularizados de forma a evitar as desvantagens de espalhamento e emaranhamento de código. Finalmente quanto à facilidade de manutenção e extensão o código de injeção de falhas é mais fácil de manter e posteriormente estender para outros modelos de falhas, pois está localizado em um único lugar ao invés de espalhado pelo código.

FICTA pode ser aplicada em três cenários distintos para validar aplicações distribuídas com características de alta dependabilidade baseadas no protocolo de rede UDP. No primeiro cenário, FICTA é uma ferramenta de teste para o desenvolvedor da aplicação distribuída. O desenvolvedor testa sua aplicação para diferentes cenários de falhas, refinando-a caso a cobertura de falhas não seja satisfatória, sem a necessidade de alterar seu código e sem a preocupação de posterior exclusão de código extra de teste. Para esse cenário, FICTA está pronta para uso imediato. Em um segundo cenário o integrador de sistemas ou o responsável pela aquisição de software precisa avaliar se uma determinada aplicação de rede ou *middleware* para sistemas distribuídos, semelhante ao JGroups por exemplo, apresenta a cobertura de falhas desejada. O fornecedor do *software* não tem qualquer interesse em abrir seu código na fase de avaliação. Neste cenário, FICTA apresenta a vantagem de não necessitar do código fonte para instrumentar a aplicação e pode também ser imediatamente aplicada no estado em que se encontra. Em um terceiro cenário deseja-se medir a confiabilidade, disponibilidade, latência de erro, queda de desempenho sob falhas ou qualquer outra métrica de dependabilidade. Neste último cenário, FICTA pode compor um ambiente mais completo onde monitores, coletores e analisadores de dados estariam disponíveis para cálculo e análise das métricas.

FICTA é semelhante quanto a funcionalidade à extensão de Jaca e à FIONA. Entretanto, sua construção é completamente diferente, seguindo uma abordagem pioneira nesta área. Todas as três ferramentas prometem nenhuma intrusividade espacial. Um

interessante trabalho futuro é comparar as três ferramentas quanto a sua intrusividade temporal em sistemas alvo que apresentem restrições de tempo-real.

9 Referências

[ARL90] ARLAT, J. Et al. *Fault Injection for dependability Validation: a methodology and some applications*. IEEE Transactions on Software Engineering, v. SE-16, n.2, Feb.1990.

[BAN98] BAN, B. *JavaGroups – Group Communication Patterns in Java*. Department of Computer Science, Cornell University, July 1998.

[BON2004] BONÉR, J. VASSEUR, A. *AspectWerkz for Dynamic Aspect-Oriented Programming*. In Proc. Of AOSD 2004.

[CAR98] CARREIRA, J., LEITE, F.O., WEBER, T.S. *Building a Fault Injector to Validate Fault Tolerant Communication Protocols*. In Proc. of International Conference on Parallel Computing Systems. Ensenada, Mexico. Aug. 1998.

[DAW96] DAWSON, S. JAHANIAN, F., MITTON, T. *ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementation*. In Procs. of IPDS'96. Sept. 1996.

[HSU97] HSUEH, M., TSAI, T., IYER, R. *Fault Injection Techniques and Tools*. IEEE Computer, v. 30, issue 4, April1997.

[JAC2004a] JACQUES, G., DREBES, R.J., GERCHMANN, J., WEBER, T.S. *FIONA*: *A Fault Injector for Dependability Evaluation of java-Based Network Applications*. In Proc. of 3rd IEEE NCA. Massachusetts, USA. 2004.

[JAC2004b] JACQUES, G., MORAES, R., WEBER, T., MARTINS, E. *Validando sistemas distribuídos desenvolvidos em Java utilizando injeção de falhas de comunicação por software*. V Workshop de Testes e Tolerância a Falhas. Maio 2004.

[KIC2001] KICKZALES, G. Et Al. An Overview of AspecJ. In Proc. 15th European Conference Object-Oriented Programming. Budapest, Hungary, June 2001.

[KIC97] KICKZALES, G. *Aspect Oriented Programming*. Proceedings of the European Conference on Object-Oriented Programming. June 1997

[LAD2002] LADDAD, R. *I Want my AOP!, Part 1 - Separate software concerns with aspect-oriented programming*. In Java World, http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect_p.html. 2002. Accessed in Feb. 2005.

[MAR2002] MARTINS, E., RUBIRA, C., LEME, N. *Jaca: A Reflective Fault Injection Tool Based on Patterns*. In Proc. of the 2002 International Conference on Dependable Systems and Networks. 2002.

[ROY2003] ROYCHOUDHURY, S., GRAY, J., WU, H., ZHANG, J., LIN, Y. *A Comparative Analysis of Meta-programming and Aspect-Orientation*. In Proc. Of the 41st Annual ACM SE Conference, Savannah, GA, March 2003.