FIRMI: Um Injetor de Falhas para a Avaliação de Aplicações Distribuídas baseadas em RMI

Juliano C. Vacaro, Taisy S. Weber

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

{jcvacaro,taisy}@inf.ufrgs.br

Abstract. *In the test phase dependable distributed applications based on RMI have to demonstrate they work correctly even in the presence of faults in the communication subsystem. Exceptions thrown by RMI in response to communication faults must be caught and handled by the application. In order to generate such exceptions and thus testing the error-handling strategies of the application, faults are injected during the test phase and the behavior of the system under these faults is observed. This paper presents a fault injector to evaluate applications that use RMI as base for their communication subsystem. It also shows the techniques used and demonstrate through an experiment the features of the tool.*

Resumo. *Aplicações distribuídas baseadas em RMI que precisem atender a requisitos de dependabilidade, devem demonstrar na fase de testes que operam corretamente mesmo na presença de falhas no subsistema de comunicação. Exceções geradas por RMI em resposta a falhas de comunicação devem ser capturadas pela aplicação e devidamente tratadas. Para provocar a ocorrência destas exceções e assim testar as estratégias de tratamento de erros da aplicação, falhas são injetadas durante a fase de testes e o comportamento do sistema sob o efeito destas falhas é observado. Este artigo apresenta um injetor de falhas para a avaliação de aplicações que usam o protocolo RMI. O artigo descreve as técnicas empregadas e demonstra as funcionalidades do injetor através da condução de um experimento.*

1. Introdução

RMI (*Remote Method Invocation*) é um sistema de comunicação baseado em objetos distribuídos, portável e de rápida prototipação utilizado como base para o desenvolvimento de aplicações distribuídas. Uma aplicação baseada em RMI com requisitos de alta disponibilidade precisa ser construída de forma que exceções do protocolo sejam capturadas e medidas sejam tomadas para continuar o processamento recuperando o estado do sistema ou usando algum recurso redundante disponível. Tal processamento alternativo ocorre apenas esporadicamente, sob alguma condição de falha, muitas vezes difícil de reproduzir durante a fase de testes da aplicação. Procedimentos de tolerância a falhas não testados possuem grande probabilidade de apresentar erros quando são finalmente ativados, provocando então danos irreparáveis ao sistema. Assim como a funcionalidade do sistema sob condições de operação normal deve ser avaliada, também o sistema deve ser testado sob condições de falha.

Vários injetores de falhas de comunicação influenciaram o desenvolvimento deste trabalho. ORCHESTRA [Dawson et al. 1996] introduz o conceito de sondagem e injeção de falhas orientada a *scripts* para avaliar e validar características temporais e de tolerância a falhas de protocolos de comunicação. Jaca [Martins et al. 2002] é uma ferramenta de injeção de falhas baseada em reflexão computacional para a linguagem Java em desenvolvimento na Universidade Estadual de Campinas. Outros injetores desenvolvidos no Grupo de Tolerância a Falhas também foram analisados. FIONA [Jacques-Silva et al. 2004] insere falhas de comunicação no protocolo UDP (recentemente a ferramenta foi estendida para operar sobre o protocolo TCP) e ComFIRM [Drebes et al. 2005], injeta falhas no nível do núcleo (*kernel*) do sistema operacional Linux nos protocolos IPv4 e IPv6.

Entretanto, nenhuma das ferramentas analisadas se mostrou capaz de emular diretamente os cenários de falhas de RMI. Neste contexto, o objetivo deste trabalho é desenvolver uma ferramenta capaz de emular os cenários de falhas de RMI de maneira eficiente e direta, e assim possibilitar a avaliação de aplicações construídas sobre este protocolo bem como os mecanismos de tolerância a falhas de tais aplicações.

A Seção 2 apresenta o conceito de injeção de falhas. A Seção 3 descreve o funcionamento do protocolo RMI e o modelo de falhas referente a sua arquitetura. Os detalhes da implementação do injetor proposto são mostrados na Seção 4. Na Seção 5 é demonstrado o uso da ferramenta através de um experimento de injeção de falhas. Considerações finais são feitas na Seção 6.

2. Injeção de Falhas

Considerando a fase de desenvolvimento do sistema, as falhas a serem emuladas e os componentes a serem avaliados, as técnicas de injeção de falhas são categorizadas em injeção de falhas baseada em simulação e injeção de falhas baseada em protótipo [Hsueh et al. 1997]. A primeira abordagem é usada nas fases iniciais de desenvolvimento do sistema enquanto que a última é usada diretamente sobre a sua versão final ou um protótipo funcional do mesmo. A injeção de falhas baseada em protótipo ainda se divide em injeção de falhas por hardware e injeção de falhas por software.

Este trabalho se restringe a injeção de falhas por software. Nesta abordagem falhas podem ser inseridas por instrumentação do código da aplicação sob teste. Instrumentação de código é uma técnica flexível e de baixo custo, porém o nível de perturbação gerado ao ambiente de teste deve ser levado em consideração.

O Projeto DBench [Madeira et al. 2001] distingue a noção de perturbação através dos conceitos de intrusividade e interferência. Intrusividade ocorre toda a vez em que modificações são feitas no sistema alvo. Interferência consiste na influência não desejada causada à aplicação sob teste. O trabalho também cita que por definição todo método de injeção de falhas causa interferência, pois componentes são adicionados ao sistema para a condução de experimentos. Os dois conceitos são similares, a diferença é que intrusividade compreende alterações diretas na aplicação enquanto que interferência altera o comportamento da aplicação de forma indireta. Ainda, é importante diferenciar interferência em desejada (as alterações causadas são bem conhecidas) e não desejada (as alterações causadas são imprevisíveis).

3. Protocolo RMI

RMI [Sun Microsystems] tem por objetivo estender o conceito de chamada local de procedimento (LPC - *Local Procedure Call*) de maneira transparente ao contexto de objetos distribuídos. A Figura 1 mostra a arquitetura do protocolo RMI. Pode-se notar a existência de quatro módulos: cliente, servidor, registro de objetos e servidor web. Objetos são criados no servidor e cadastrados no registro de objetos RMI a fim de serem acessados remotamente. A comunicação entre cliente e servidor é feita através da invocação de métodos de tais objetos. Através do protocolo RMI, um cliente acessa o registro para obter uma referência de um objeto remoto localizado no servidor. Na comunicação por invocação de métodos, parâmetros (possivelmente objetos) são transferidos. Quando uma classe não está definida na máquina virtual corrente, RMI irá obter a classe da máquina virtual remota através do servidor web.

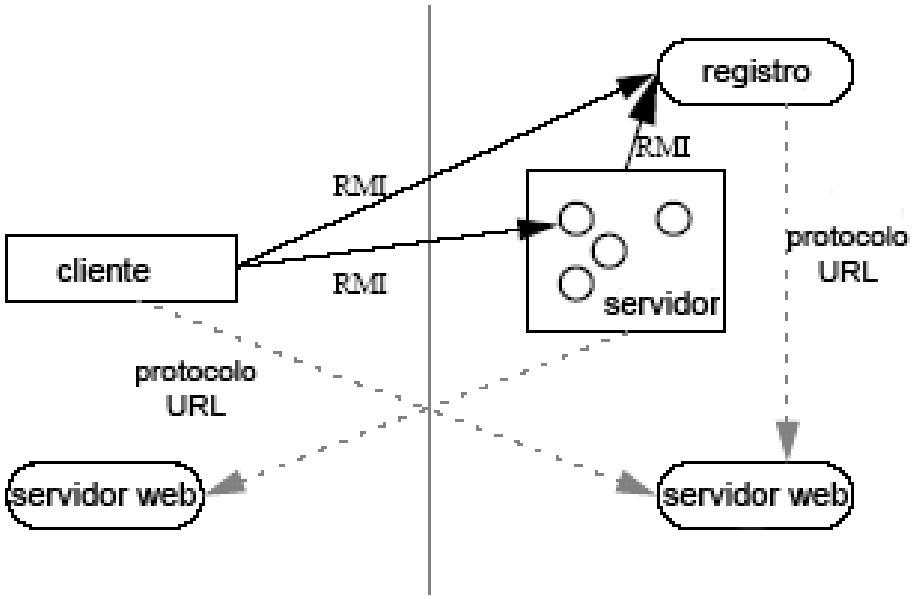

Figura 1. Arquitetura do protocolo RMI

O protocolo RMI introduz e também usa diversos recursos já existentes da linguagem Java adaptados ao contexto do mecanismo de comunicação via objetos remotos. Dentre os recursos estão o mecanismo de *Stubs* e *Skeletons*, Coletor de Lixo (*Garbage Collection*), Carga Dinâmica de Classes (*Dynamic Class Loading*), Ativação de Objetos Remotos (*Remote Object Activation*) e a opção do uso de HTTP como protocolo de transporte a fim de evitar problemas de comunicação entre domínios devido ao uso de *firewalls*. RMI usa a Serialização de Objetos como base para a implementação da maioria dos recursos mencionados e a partir da versão 1.5 de Java, usa Reflexão como nova alternativa na geração de *Stubs*.

A comunicação entre objetos remotos é baseada no conceito de *proxies*. Um cliente para ter acesso a objetos em um servidor deve primeiramente obter uma referência (*proxy*) para o objeto. Um *proxy* (*Stub*) de um objeto é um objeto que possui a mesma interface que o objeto original, mas que repassa as requisições de invocação de métodos via rede ao objeto "real" no servidor. Ao receber a requisição para a execução de um

método, o servidor irá obter o objeto correto para processar a requisição. O resultado da computação é então retornado ao *proxy* no cliente e posteriormente para a aplicação.

3.1. Modelo de Falhas

Em um sistema distribuído baseado em rede, onde a comunicação entre nodos é feita através de troca de mensagens, falhas são comuns. Cristian [Cristian 1991], Schneider [Schneider 1993], Birman [Birman 1997] definem modelos de falhas para descrever tal comportamento. Este trabalho utiliza a definição proposta por Birman, pois esta considera falhas de particionamento de rede, que é uma situação de falha possível dentro do contexto de RMI. Birman define falhas de colapso, parada segura (*fail-stop*), omissão de envio, omissão de recepção, rede, particionamento de rede, temporização e bizantinas.

Além de falhas de comunicação, falhas que afetam o conteúdo das mensagens antes que as mesmas sejam encaminhadas para rede, podem ocorrer. Pode-se citar falhas de *hardware*, como dispositivos de armazenamento ou até falhas humanas (maliciosas ou não), inseridas na aplicação durante a fase de desenvolvimento [Avizienis et al. 2004]).

3.2. Comportamento de RMI na Presença de Falhas

Tomando como base a Figura 1 e o modelo de falhas descrito na Seção 3.1 é possível estabelecer como as falhas manifestadas nos níveis inferiores de abstração do sistema RMI afetam os componentes de maior nível de abstração. É importante salientar que: 1) toda a comunicação RMI é sempre baseada no paradigma cliente/servidor, ou seja, uma comunicação sempre envolverá um cliente e um servidor, isto ocorrerá mesmo que um nodo possa assumir ambos os papéis, 2) todos os erros detectados no nodo servidor são sempre enviados ao cliente e 3) quando uma falha é detectada pelo sistema RMI, um erro será sinalizado para a aplicação através do uso de exceções.

Exceções em RMI podem ser categorizadas em três grupos: exceções referentes ao protocolo de transporte de dados (TCP ou HTTP), exceções geradas pelo protocolo RMI e exceções geradas por objetos remotos (definidas pelo usuário). Ambos nodos cliente e servidor estão sujeitos a falhas. A questão é identificar em que momento uma falha em um dado componente afetará o sistema. Nesta seção serão abordadas as principais situações geradas direta ou indiretamente por falhas do sistema de comunicação.

Durante o processo de invocação, um cliente com acesso a uma referência remota solicita ao servidor a execução de um determinado método. Neste processo, falhas podem ocorrer. O colapso do servidor, antes que seja efetuada a requisição, gerará uma exceção à aplicação de `java.rmi.UnknownHostException` (servidor não encontrado). Entretanto, se o nodo entrar em colapso durante a execução da requisição, o nodo cliente não saberá se o servidor ainda está processando o pedido, fazendo com que o cliente fique esperando indefinidamente ou até que o servidor seja reiniciado. O problema mencionado também ocorre no momento em que o servidor processou a requisição e deve retornar o resultado para o cliente. Caso o cliente entrar em colapso durante este processo, o servidor ficará indefinidamente esperando pelo cliente. A especificação de RMI não define a obrigatoriedade da criação de *threads* para o tratamento de cada requisição no nodo servidor, por esta razão, o servidor poderá ficar bloqueado.

Falhas de temporização são mais significativas no nodo servidor. Este tipo de falha pode ocorrer devido a sobrecarga do nodo em questão, degradando os serviços oferecidos

pelo componente.

Particionamento de rede é um cenário de falha que pode ocasionar problemas ao sistema de Coletor de Lixo Distribuído (Distributed Garbage Collection) de RMI. Quando um cliente obtém uma referência remota, é associada uma permuta (*Lease*), que define o tempo que um objeto é considerado pelo servidor como referenciado por um cliente. É responsabilidade do cliente atualizar o tempo de uso de cada objeto antes que o mesmo expire. Uma falha de particionamento pode fazer com que clientes não sejam capazes de atualizar a permuta dos objetos que referenciam. Isto faz com que o servidor considere os objetos como não mais referenciados, portanto sendo coletados. Ao retornar da falha de particionamento, clientes não serão mais capazes de acessar seus objetos remotos.

Existem casos em que o sistema RMI não é capaz de detectar a manifestação de certas falhas. Por exemplo, a alteração dos valores de parâmetros passados durante o processo de invocação pode não ser percebida pelo sistema. Como descrito na Seção 3.1, isto pode ocorrer devido a falhas em dispositivos de armazenamento ou por falhas humanas (maliciosas ou não). Na Seção 4.3 é mostrado como o injetor, através da descrição de cenários de falhas, emula os erros previstos na especificação de RMI.

4. Injetor

O injetor FIRMI descrito neste trabalho deve ser capaz de emular os cenários de falhas de RMI. O objetivo da ferramenta é avaliar o comportamento sob falhas de aplicações que usam o protocolo como infraestrutura de comunicação bem como a cobertura de falhas dos mecanismos de tolerância a falhas empregados. Como requisitos do injetor estão facilidade de uso, portabilidade, integrabilidade e compatibilidade com diferentes implementações de RMI como a Máquina Virtual Java desenvolvida pela IBM [IBM Corporation]. Para tanto, toda a especificação da ferramenta é baseada na especificação fornecida pela Sun.

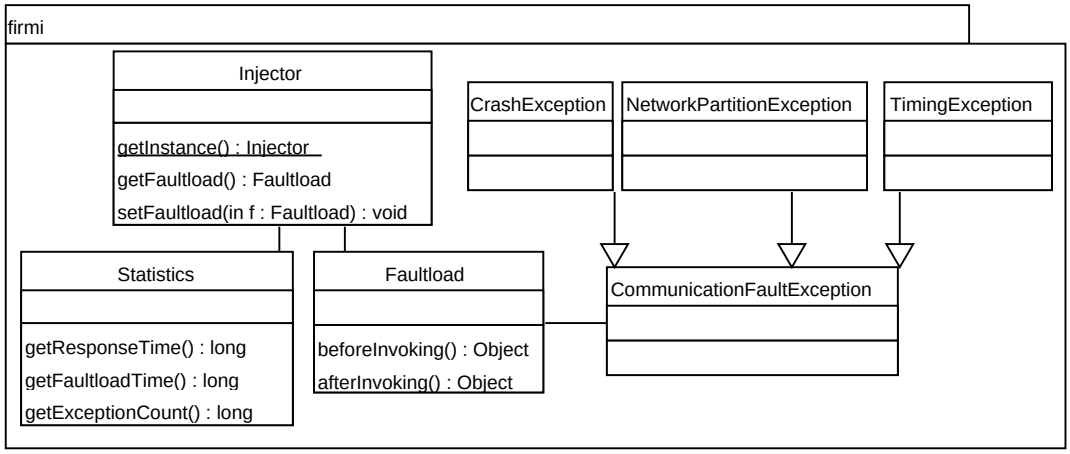

Figura 2. Diagrama de classes de FIRMI

A Figura 2 é um diagrama de classes de FIRMI. A Classe `Injector` é o ponto de acesso das funcionalidades da ferramenta tanto pelos ganchos inseridos no código de RMI quanto pelos próprios cenários de falhas. A classe `Faultload` define a estrutura de um módulo de cenário de falhas, devendo ser derivada a fim de especificar como falhas

serão inseridas no sistema sob teste. FIRMI também define classes para que o engenheiro de teste não precise ter um conhecimento prévio dos tipos de exceções do protocolo, deixando ao injetor o papel de mapear tais erros para exceções específicas de RMI. Por fim, a classe `Statistics` contém as informações coletadas durante um experimento. A classe pode ser acessada para que as informações sejam armazenadas ou até usadas pelo cenário de falhas para tomada de decisão. As próximas seções descreverão com mais detalhes cada componente de FIRMI.

4.1. Instrumentação do Protocolo

Um dos requisitos do injetor é que este deve ser compatível com diferentes implementações de RMI. Desta maneira, a inserção dos ganchos do injetor no sistema deve ser feita de maneira padronizada. Então, a instrumentação das classes de sistema de RMI foi baseada na especificação fornecida pela Sun. Também é desejado que a descrição dos cenários de falhas seja similar à API do protocolo, portanto, é necessário que o local escolhido para os ganchos seja o ponto mais abstrato em que não se perca a semântica do protocolo.

A análise da especificação de RMI mostrou que a localização ideal para a instrumentação do protocolo são as estruturas pertinentes aos Stubs e Skeletons, já que estes concentram os fluxos de envio e recepção de requisições. A especificação de RMI define as interfaces `java.rmi.server.RemoteRef` e `java.rmi.server.ServerRef`. A primeira é usada no nodo cliente para o envio de requisições enquanto que a segunda é usada no nodo servidor na recepção de requisições. Como estas estruturas tratam-se de interfaces, deve-se alterar as classes que implementam tais definições, fato que permite ao injetor operar sobre qualquer implementação de RMI que seja compatível com a especificação.

4.2. Técnicas de Instrumentação

Identificado o local para a inserção dos ganchos do injetor, é necessário decidir qual a melhor técnica para realizar a instrumentação do código. Com este objetivo, foram analisadas ferramentas desde a manipulação de *bytecodes* como Javassist [Chiba 1998], ASM [ObjectWeb Consortium], Serp [White] e BCEL [Apache Software Foundation] até ferramentas com base em orientação a aspectos como AspectJ5 [Eclipse Foundation a]. A maioria das ferramentas analisadas suportam a instrumentação estática e dinâmica de classes em Java, fato que influencia a interferência causada na aplicação alvo.

As classes a serem instrumentadas para o injetor são classes de sistema. Classes de sistema são carregadas pelo Carregador de Classes de Inicialização (*Bootstrap Class Loader*) na ativação da máquina virtual. Uma vez carregada, uma classe não pode ser alterada. Consequentemente as técnicas convencionais de instrumentação dinâmica não podem ser usadas. Desta forma, é necessário usar o suporte oferecido pela máquina virtual. A substituição de classes de sistema pode ocorrer durante a inicialização ou durante a execução da máquina virtual Java. No primeiro caso é indicado o caminho para o Carregador de Classes de Inicialização das novas classes de sistema. Assim, o Carregador usará a primeira classe encontrada como a classe de sistema sendo procurada. Para a substituição durante a execução da máquina virtual é usada a Arquitetura de Depuração Java (JPDA), um framework poderoso com funcionalidades para depuração e monitoramento de aplicações e, a partir da versão 1.5 de Java, também pode ser usado o pacote

`java.lang.instrument`, uma API totalmente baseada em Java específica para este objetivo.

A escolha para a melhor técnica de instrumentação foi baseada em três aspectos: simplicidade, flexibilidade e menor interferência causada na aplicação alvo. Assim, foi escolhida Javassist para a manipulação de *bytecodes*. Para a substituição das classes de sistema de RMI é usado o suporte oferecido pelo pacote `java.lang.instrument`. Javassist foi escolhida por sua facilidade de uso e por já ter sido testada com sucesso no injetor Jaca [Martins et al. 2002]. O pacote `java.lang.instrument` foi escolhido pois é específico para a substituição de classes e por permitir a definição de vários agentes simultaneamente, possibilitando a integração deste injetor com outros injetores como FIONA [Jacques-Silva et al. 2004] com modificações mínimas no seu sistema de instrumentação.

No entanto, é possível que, em certas situações, a interferência causada pela substituição de classes durante a execução de aplicações (pois é adicionado o tempo de alteração da classe durante o procedimento de carga da mesma dentro do ambiente de execução da máquina virtual) não seja desejada. Por esta razão o injetor também foi projetado para suportar a substituição de classes durante a inicialização da máquina virtual.

4.3. Cenários de Falhas

Um experimento de injeção de falhas inicia pela definição do conjunto de falhas, também chamado carga de falhas, que será injetado na aplicação alvo. Essa carga de falhas deve ser representativa dos tipos de falhas que a aplicação está sujeita em operação normal e se baseia no modelo de falhas para o sistema. Diferentes cargas de falhas podem ser definidas para uma mesma aplicação sob a mesma carga de trabalho. Nenhuma carga de falhas sozinha consegue reproduzir todas as situações possíveis de falhas às quais a aplicação pode estar sujeita. Uma carga de falhas especificada compõe o cenário de falhas de um teste em um experimento de injeção de falhas.

FIRMI interage com o cenário de falhas através da classe `Faultload` (Figura 2). Toda a vez em que uma requisição RMI é originada de um cliente ou recebida pelo servidor, o injetor irá invocar o módulo de tratamento do cenário de falhas para interceptar e manipular a requisição. Este comportamento é similar a noção de `Listeners`, largamente utilizado na linguagem. Para cada ação reconhecida pelo injetor, existe um método que será invocado e passará o controle ao módulo de cenário de falhas. O método `beforeInvoking` é invocado imediatamente antes que o cliente faça a requisição remota. Já o método `afterInvoking` é invocado imediatamente após o retorno do processamento do nodo servidor.

Para criar um módulo de cenário de falhas correspondendo a carga de falhas desejada para um experimento o engenheiro de teste estende a classe `Faultload` de FIRMI e redefine os métodos `beforeInvoking` e `afterInvoking`. Ao interceptar uma requisição, o injetor poderá: 1) não fazer nada, permitindo a execução da requisição sem falhas; 2) alterar os parâmetros referentes ao método sendo invocado (neste caso é emulada a situação de falhas de dispositivos de armazenamento ou humanas conforme mencionado na Seção 3.1); ou finalmente 3) retornar uma exceção emulando uma situação de erro para a aplicação, reproduzindo falhas de comunicação ou até erros do próprio sistema RMI, de acordo com o tipo de exceção escolhido. Na Seção 3.2 foi definida

a relação entre as falhas do sistema de comunicação e como estas influenciam os componentes de maior nível de abstração. Para que o engenheiro de teste não precise ter um conhecimento prévio dos tipos de exceções de RMI, FIRMI define exceções como `CrashException`, `NetworkPartitionException` e `TimingException` que serão internamente mapeadas para exceções específicas de RMI. Essa característica, no entanto, não restringe a possibilidade do uso de exceções específicas do protocolo diretamente pelo módulo de cenário de falhas.

O uso de classes Java para a descrição de cenários de falhas facilita o processo de integração do injetor em várias plataformas de desenvolvimento, permitindo estender *frameworks* já existentes com o conceito de injeção de falhas no auxílio à avaliação de sistemas. Além disso, possibilita que a seleção e a manipulação de mensagens sejam influenciadas por informações estáticas, informações internas ao próprio módulo de cenário de falhas e informações extraídas diretamente da aplicação sob teste. Estas características possibilitam a criação de cenários de falhas representativos, pois o comportamento da injeção de falhas pode ser modificado dinamicamente durante a execução da aplicação sob teste.

4.4. Monitoramento

A arquitetura cliente/servidor com comunicação baseada em objetos distribuídos facilita a tarefa de monitoramento e geração de medidas estatísticas da execução de aplicações. A classe `Statistics` é a responsável pelo armazenamento de tais medidas. As informações desta classe podem ser acessadas durante a execução de um experimento pelos próprios cenários de falhas, permitindo ao engenheiro de teste tomar decisões baseado nestas medidas.

Durante um experimento, falhas são inseridas no ambiente de execução da aplicação sob teste, o que pode levar esta a apresentar colapso ou simplesmente entrar em *loop* descartando os dados coletados pelo injetor. Para evitar este problema, é registrado junto a JVM um pedido de notificação do término da execução da máquina virtual (inclusive por motivo de falha) através do método `Runtime.addShutdownHook`. Desta maneira, o injetor poderá salvar todas as medidas coletadas no experimento.

Entre as medidas coletadas por FIRMI, pode-se citar o tempo de resposta de requisições RMI (medido no nodo cliente), o tempo gasto para o processamento de requisições no nodo servidor e a quantidade de acessos de clientes para um dado objeto remoto. Estas informações permitem identificar possíveis degradações de desempenho na ocorrência de falhas e a identificação de gargalos no sistema, pois serão localizados quais os objetos mais acessados e quais métodos demandam maior tempo de execução em um servidor. Além de medidas estatísticas, FIRMI também gera *logs* da execução de RMI e das ações tomadas pelos módulos de cenários de falhas. Combinando as medidas coletadas pelo injetor, os *logs* das ações tomadas pelos módulos de cenários de falhas e as informações geradas pelas próprias aplicações sob teste é possível estabelecer relações entre as falhas inseridas pelo injetor e o comportamento de tais aplicações.

Outra medida importante prevista é o tempo gasto pela execução dos cenários de falhas. A interferência do injetor corresponde ao tempo gasto pelo mecanismo de instrumentação (estático ou dinâmico), o tempo gasto pelo código do injetor e o tempo gasto pela execução dos módulos dos cenários de falhas. O mecanismo de instrumentação

é configurável, permitindo a escolha do método estático para a redução da interferência ou o método dinâmico para flexibilizar o uso da ferramenta. O tempo gasto pelo código do injetor é relativamente baixo e previsível, pois este é composto basicamente pelos ganchos nas classes de sistema de RMI. O tempo mais significativo e não previsível está na execução dos cenários de falhas, já que é dependente da aplicação alvo e do que se deseja validar. Para que o engenheiro de teste tenha controle da interferência da ferramenta, é medido o tempo gasto na execução dos cenários de falhas, permitindo a modificação dos mesmos a fim de otimizar sua funcionalidade.

5. Experimentos

Para demonstrar as funcionalidades do injetor foi escolhida uma aplicação alvo que realiza operações aritméticas elementares. A aplicação, baseada em RMI, consiste de um objeto remoto `RemoteCalculator` e um cliente que apenas invoca os métodos do servidor e imprime os resultados. O programa foi executado com sucesso em ambas implementações da máquina virtual Java disponibilizadas pela Sun e IBM.

```
public interface RemoteCalculator extends Remote {
  public int plus(int a, int b) throws RemoteException;
  public int minus(int a, int b) throws RemoteException;
}
```

Figura 3. Interface remota `RemoteCalculator`

```
public static void main(String[] args) throws Exception {
  Registry r = LocateRegistry.getRegistry(args[0]);
  rc = (RemoteCalculator) r.lookup("rc");
  System.out.println("3 + 2 = " + rc.plus(3, 2));
  System.out.println("3 + 2 = " + rc.plus(3, 2));
  System.out.println("3 - 2 = " + rc.minus(3, 2));
}
```

Figura 4. Método `main` do nodo cliente

As Figuras 3 e 4 mostram respectivamente a interface remota da classe `RemoteCalculator` e o código do nodo cliente que invoca o objeto remoto. A interface `RemoteCalculator` é bem simples possuindo os métodos `plus` e `minus`, que realizam a adição e subtração de dois números inteiros. Já o código no nodo cliente invoca os métodos da interface remota e exibe os resultados. A Figura 5 mostra a execução da aplicação descrita.

```
3 + 2 = 5
3 + 2 = 5
3 - 2 = 1
```

Figura 5. Resultado do experimento sem injeção de falhas

Após a fase de testes convencionais onde é avaliado se a aplicação alvo realiza as funções especificadas sobre um dado conjunto de entradas fornecendo o resultado correto,

```
public class FaultloadImpl extends Faultload {
  private int times = 0;

  public Object beforeInvoking(Remote obj, Method method,
      Object[] params, long opnum) throws RemoteException {
    times += 1;
    if (method.getName().equals("plus") && times <= 1) {
      params[0] = new Integer(6);
      params[1] = new Integer(4);
    } else if (method.getName().equals("minus")) {
      throw new CrashException("host not found");
    }
    return null;
  }

  public Object afterInvoking(Remote obj, Method method,
      Object[] params, long opnum, Object result)
      throws RemoteException {
    if (method.getName().equals("plus") && times > 1)
      result = new Integer(1);
    return result;
  }
}
```

Figura 6. Cenário de falhas para a condução do experimento

o próximo passo é realizar experimentos para observar o comportamento da aplicação alvo na presença de falhas. Para a aplicação alvo usada como exemplo foi escolhida uma carga de falhas que compreende a alteração dos operandos das funções aritméticas, por falha bizantina ou de hardware, e também a emulação do colapso do nodo servidor, essa última uma falha de comunicação comum em ambientes cliente/servidor.

A Figura 6 mostra a implementação do cenário de falhas usado no experimento. No método `beforeInvoking` são alterados os parâmetros da operação `plus` antes que a requisição seja enviada para o servidor. Caso seja uma operação de subtração é emulado o colapso do nodo servidor através da geração da exceção `CrashException` de FIRMI. Já o método `afterInvoking` altera o valor de retorno da operação de adição para demonstrar que mesmo inserindo falhas apenas no nodo cliente é possível ter o controle das etapas envolvidas no processo de invocação remota. Neste experimento, o *faultload* injeta falhas no sistema baseado no método sendo invocado, porém esta não é a única maneira de escolha. Como descrito na Seção 4.3, a seleção e manipulação de mensagens é influenciada por informações estáticas, informações internas ao próprio módulo de cenário de falhas e informações extraídas diretamente da aplicação sob teste.

Para a condução do experimento o injetor foi configurado para instrumentar as classes de sistema de maneira estática com a descrição do cenário de falhas através de classes Java e inserindo falhas a partir do nodo cliente. O resultado da execução do programa com injeção de falhas pode ser visto na Figura 7.

Apesar da exceção gerada ter sido do tipo `CrashException`, o injetor realizou o mapeamento da mesma para uma exceção específica do protocolo RMI

```
3 + 2 = 10
3 + 2 = 1
Exception in thread "main" java.rmi.UnknownHostException:
  host not found
...
```

Figura 7. Resultado do experimento com injeção de falhas

(`UnknownHostException`). O experimento permite ilustrar as funcionalidades da ferramenta FIRMI e a forma de conduzir uma validação por injeção de falhas. Os resultados da Figura 7 mostram que a aplicação não possui nenhum mecanismo de tolerância a falhas. Quando sujeita a falhas de comunicação, a aplicação apresenta defeito, ou seja, não fornece os resultados desejados. A condução do experimento também mostra que a presença do injetor não foi percebida pela aplicação, ou seja, o colapso do nodo servidor, para a aplicação, ocorreu na realidade. Com base neste experimento, procurou-se mostrar que FIRMI emula situações de falhas de RMI, viabilizando sua utilização na avaliação de aplicações distribuídas que usam este protocolo.

6. Conclusão

RMI é uma abstração de comunicação de alto nível que facilita o desenvolvimento de aplicações distribuídas, diminuindo o risco de falhas de desenvolvimento por ter uma API similar à chamada local de procedimento. Por esta razão torna-se uma alternativa para a construção de tais aplicações. Porém, falhas inerentes ao processo de comunicação ou falhas que afetem os nodos participantes do sistema são inevitáveis. Aplicações que usam RMI devem estar preparadas para tratar estas falhas evitando que o sistema apresente um comportamento não esperado e assim garantir requisitos de dependabilidade.

Com o objetivo de resolver esta questão, este trabalho apresenta um injetor de falhas capaz de emular de forma direta os cenários de falhas de RMI. A ferramenta pode ser usada nas fases de teste de sistemas completando o ciclo de desenvolvimento de aplicações e assegurando o correto funcionamento das mesmas na presença de falhas. O injetor foi especificado para possibilitar ao engenheiro de teste liberdade e flexibilidade na descrição de cenários de falhas e que a interferência causada nas aplicações sob teste seja conhecida e controlada. Apesar de simples, o experimento conduzido demonstra a capacidade da ferramenta na emulação de falhas, possibilitando a construção de situações próximas as esperadas na realidade. As próximas etapas deste projeto são a condução de experimentos com a plataforma de computação em Grid, Ourgrid [Andrade et al. 2003]. A característica de integração da ferramenta também será explorada com a extensão de ambientes de desenvolvimento e teste de aplicações como Eclipse [Eclipse Foundation b] e JUnit [Gamma and Beck].

Referências

Andrade, N., Cirne, W., Brasileiro, F., and Roisenberg, P. (2003). OurGrid: An approach to easily assemble grids with equitable resource sharing. In *9th Workshop on Job Scheduling Strategies for Parallel Processing*.

Apache Software Foundation. BCEL - byte code engineering library. http://jakarta.apache.org/bcel.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

Birman, K. P. (1997). *Building Secure and Reliable Network Applications*. Prentice Hall, 1 edition.

Chiba, S. (1998). Javassist - a reflection-based programming wizard for java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada.

Cristian, F. (1991). Understanding fault-tolerand distributed systems. *Communications of the ACM*, 34(2):56–78.

Dawson, S., Jahanian, F., Mitton, T., and Tung, T.-L. (1996). Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *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.

Eclipse Foundation. aspectj project. http://www.eclipse.org/aspectj.

Eclipse Foundation. eclipse project. http://www.eclipse.org/.

Gamma, E. and Beck, K. Junit. http://www.junit.org.

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

IBM Corporation. IBM java virtual machine. http://www.ibm.com/developerworks/java.

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. In *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (NCA'04)*, pages 303–308, Washington, DC, USA. IEEE Computer Society.

Madeira, H., Kanoun, K., Arlat, J., Crouzet, Y., Johansson, A., and Lindström, R. (2001). Preliminary dependability benchmark framework. Technical report, LAAS-CNRS, Toulouse, France.

Martins, E., Rubira, C. M. F., and Leme, N. G. M. (2002). Jaca: A reflective fault injection tool based on patterns. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pages 483–487, Washington, DC. IEEE Computer Society Press.

ObjectWeb Consortium. ASM. http://asm.objectweb.org.

Schneider, F. B. (1993). What good are models and what models are good? In Mullender, S., editor, *Distributed Systems*, pages 17–26. Addison-Wesley, Workingham, 2nd edition.

Sun Microsystems. Java remote method invocation (java RMI) specification. http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html.

White, A. Serp. http://serp.sourceforge.net.