Injeção de Falhas na Fase de Teste de Aplicações Distribuídas

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. *Injecting communication faults, a test engineer can observe the behavior of network based applications in response to real faults and thus validate the fault tolerance of such systems. However, fault injection alone is not sufficient to validate other important functional aspects of an application. In order to increase the test coverage, fault injection must be applied together with well known software testing techniques integrated into conventional development and test environments. To reach these requirements, a fault injector to evaluate distributed applications based on RMI and its integration into JUnit and ANT frameworks are introduced.*

Resumo. *Injeção de falhas de comunicação visa emular falhas comuns a sistemas baseados em rede para assim observar o comportamento de aplicações em resposta a diferentes condições de falha. Apesar de eficiente, a técnica não é suficiente para validar outras características funcionais importantes de uma aplicação. Para aumentar a cobertura dos testes, injeção de falhas deve ser usada de forma complementar a técnicas convencionais de teste e operar integrada a plataformas de desenvolvimento e teste de sistemas. Visando atender estes requisitos, este trabalho apresenta um injetor de falhas para a avaliação de aplicações distribuídas baseadas em RMI, e sua integração aos frameworks JUnit e ANT.*

1. Introdução

Sistemas distribuídos vêm sendo desenvolvidos usando Java, sendo a invocação remota de métodos o mecanismo preferencial para a interação entre objetos. Para a validação destes sistemas, o engenheiro de testes se defronta com o desafio de, além de avaliar as funcionalidades de aplicações em resposta a entradas especificadas e situações esperadas, incluir testes em situações de erro. Estes últimos permitem verificar a robustez, disponibilidade e confiabilidade do sistema. Entretanto, o teste de aplicações distribuídas sob situações de erro não é uma tarefa trivial, pois o comportamento do ambiente de operação de tais sistemas, incluindo os diversos nodos de execução e os *links* de comunicação, não é facilmente emulado. Atuar fisicamente sobre o ambiente de execução, desligando nodos servidores e provocando desconexão de *links*, nem sempre é uma estratégia adequada principalmente quando o ambiente de teste é o próprio ambiente de desenvolvimento e monitoramento, quando não o próprio ambiente de produção da empresa.

Ferramentas de injeção de falhas por software mostram-se um método eficiente para a validação de sistemas. A abordagem visa a emulação de falhas de *hardware*, sem

comprometer fisicamente o ambiente, e a análise do comportamento do sistema na presença destas falhas. Porém, para que injeção de falhas possa ser usada de maneira complementar às técnicas já empregadas para o teste funcional ou estrutural das aplicações, é necessário que estas ferramentas sejam integradas a plataformas que tenham este objetivo. A adequada integração de injetores de falhas a ambientes usuais de teste permite ao engenheiro de teste criar cenários mais representativos das situações esperadas no ambiente de operação de tais aplicações.

Este artigo apresenta um injetor de falhas para aplicações Java distribuídas baseadas em RMI e sua integração a ambientes de teste. Vários injetores de falhas influenciaram o desenvolvimento deste trabalho, entre elas Jaca [Martins et al. 2002], uma ferramenta de injeção de falhas baseada em reflexão computacional para aplicações Java, e FIONA [Jacques-Silva et al. 2004], que insere falhas de comunicação no protocolo UDP. Entretanto, nem as ferramentas citadas nem as demais que foram analisadas [Dawson et al. 1996, Stott et al. 2000, Chandra et al. 2004, Drebes et al. 2005] se mostraram capazes de emular diretamente os cenários de falhas de RMI. Nenhuma das ferramentas previa a integração em ambientes de teste. Neste contexto, o objetivo deste trabalho foi 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 detecção e tratamento de erros de tais aplicações.

O injetor apresentado neste artigo, denominado FIRMI, complementa o ciclo de teste de aplicações através da integração em diferentes plataformas de desenvolvimento, agregando as vantagens da técnica de injeção de falhas voltada a sistemas distribuídos para estes ambientes. A Seção 2 apresenta o conceito de injeção de falhas. A Seção 3 descreve o modelo de falhas referente a arquitetura de RMI. Os detalhes do injetor são mostrados na Seção 4. A integração de FIRMI nas plataformas de teste e desenvolvimento de aplicações é detalhada na Seção 5. Na Seção 6 é demonstrado o uso da ferramenta através de um experimento de injeção de falhas. Trabalhos relacionados são descritos na Seção 7 e considerações finais são feitas na Seção 8.

2. Injeção de Falhas

Injeção de falhas corresponde à inserção artificial de falhas em um sistema computacional real para avaliar o seu comportamento na presença de falhas. Neste processo de validação deve ser especificada uma carga de trabalho (conjunto de ações que o sistema alvo irá executar) e a carga de falhas (falhas que serão injetadas no alvo durante a execução da carga de trabalho). Uma rodada de testes usando um injetor de falhas com uma determinada carga de trabalho e uma determinada carga de falhas é chamado experimento.

Este trabalho foca a injeção de falhas de comunicação por *software* [Hsueh et al. 1997]. Nesta abordagem são emuladas falhas em componentes de *hardware*, acelerando a manifestação de erros devidos a tais falhas e assim permitindo observar o comportamento do sistema sob teste. A injeção de falhas emuladas de *hardware* por *software* é diferente da injeção de falhas de *software*, onde é comum a geração de mutantes de código [Delamaro et al. 2001b, Delamaro et al. 2001a] para verificação da cobertura dos procedimentos de teste. A injeção de falhas por *software* opera principalmente alterando o fluxo de execução para que as rotinas do injetor responsáveis pela emulação das falhas de *hardware* possam inserir erros na aplicação sob teste durante sua execução. Uma das

maneiras mais eficientes de inserir erros é por instrumentação de código.

3. Comportamento de RMI na Presença de Falhas

A arquitetura de RMI [Sun Microsystems 1997] consiste basicamente de três módulos: cliente, servidor e registro de objetos. Para que objetos sejam referenciados por clientes, estes devem ser criados no nodo servidor e cadastrados no registro de objetos. 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.

Como RMI é uma abstração de alto nível de comunicação, falhas inerentes a mecanismos de mais baixo nível afetarão as aplicações baseadas neste protocolo. Assim, é importante identificar as situações de falha possíveis em sistemas distribuídos para posteriormente compreender como estas falhas podem afetar componentes de maior nível de abstração. Este trabalho utiliza um modelo de falhas de comunicação [Birman 1996], que define falhas de colapso (o componente pára sua execução e entra em colapso sem tomar ações incorretas), parada segura ou *fail-stop* (similar à falha de colapso, porém a falha é precisamente detectável pelos outros componentes do sistema), omissão de envio e recepção de mensagens (mensagens são perdidas no próprio nodo emissor ou receptor durante o processamento do sistema operacional), rede (mensagens são descartadas pela infraestrutura de comunicação), particionamento de rede (uma rede se divide em uma ou mais subredes desconectadas, onde os nodos de cada subrede não se comunicam), temporização (onde um requisito temporal do sistema é violado) e bizantinas (incluem uma grande variedade de comportamentos falhos, como o corrompimento de dados ou até um comportamento malicioso da aplicação).

Ambos nodos cliente e servidor estão sujeitos a falhas. É importante salientar que: 1) toda a comunicação RMI é sempre baseada no paradigma cliente/servidor, 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 de exceções. Estas exceções podem ser de três tipos: as referentes ao protocolo de transporte de dados (TCP ou HTTP), as geradas pelo protocolo RMI e finalmente as geradas por objetos remotos (definidas pelo usuário).

Durante o processo de invocação, um cliente com acesso a uma referência remota solicita ao servidor a execução de um método. O colapso do servidor, antes que seja efetuada a requisição, é detectado por RMI e sinalizado à aplicação através de uma exceção. 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. Particionamento de rede pode ocasionar problemas ao sistema de coletor de lixo distribuído 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.

Falhas de temporização são mais significativas no nodo servidor. Este tipo de falha pode ocorrer devido à sobrecarga do nodo em questão, degradando os serviços oferecidos pelo componente. Ainda, falhas bizantinas, como a alteração dos valores de parâmetros passados durante o processo de invocação podem não ser percebidas pelo sistema RMI, podendo ocasionar o colapso de aplicações. Na Seção 4.3 é mostrado como o injetor, através da descrição de cenários de falhas, emula as situações vistas nesta seção.

4. Injetor

FIRMI, *Fault Injector for RMI*, é capaz de emular os cenários de falhas de comunicação para aplicações baseadas em RMI. O objetivo da ferramenta é permitir avaliar o comportamento sob falhas de aplicações que usam o protocolo RMI bem como a cobertura de falhas dos mecanismos de tolerância a falhas por elas empregados. Para que o injetor seja usado de maneira plena na fase de teste de sistemas, FIRMI foi projetado para ter uma arquitetura simples e modular, possibilitando a integração em diferentes ambientes de desenvolvimento.

RMI é largamente usado em aplicações distribuídas, sendo alvo de otimizações e adaptações a diferentes necessidades. Cheng-Wei Chen [Chen et al. 2004] estende RMI para operar sobre ambientes como Bluetooth, GPRS e WLAN. Chang [Chang and Aharnad 2004] estende RMI para comunicação P2P, agregando escalabilidade e tolerância a falhas ao *framework*. Para que o injetor possa ser usado na avaliação de sistemas que usam diferentes implementações de RMI, como as citadas, o projeto de FIRMI segue a especificação de RMI [Sun Microsystems 1997], ficando transparente ao injetor os detalhes de uma determinada implementação.

4.1. Instrumentação do Protocolo

Além da compatibilidade com diferentes implementações de RMI, também é desejado que a descrição dos cenários de falhas seja similar à API do protocolo, portanto, é necessário que o local escolhido para a inserção dos ganchos de instrumentação seja o ponto mais abstrato em que não se perca a semântica do protocolo. A análise da especificação 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 define as interfaces `java.rmi.server.RemoteRef` e `java.rmi.server.ServerRef`, sendo a primeira usada no nodo cliente para o envio e a segunda no nodo servidor na recepção de requisições. Como são interfaces, deve-se alterar as classes que implementam tais definições, fato que permite ao injetor operar sobre qualquer implementação de RMI compatível com a especificação.

As classes a serem instrumentadas para o injetor são classes de sistema. Classes de sistema são carregadas pelo Carregador de Classes de Inicialização na ativação da máquina virtual. Uma vez carregada, uma classe não pode ser alterada. Consequentemente as técnicas convencionais de instrumentação não podem ser usadas. Desta forma, é necessário usar o suporte oferecido pela máquina virtual. FIRMI usa o pacote

`java.lang.instrument`, 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. O pacote `java.lang.instrument` tem acesso às classes carregadas pelo Carregador de Classes de Inicialização conforme estas são referenciadas na aplicação, podendo operar sobre o conteúdo das mesmas. Apesar de flexível, a interferência causada pelo agente de instrumentação durante a execução de uma aplicação pode não ser desejada. Então, FIRMI também foi projetado para suportar a substituição de classes durante a inicialização da máquina virtual. Aqui, é indicado o caminho para o Carregador de Classes de Inicialização das novas classes de sistema (opção `-Xbootclasspath`). Assim, o Carregador usará a primeira classe encontrada como a classe de sistema sendo procurada.

Note que tanto o uso do pacote `java.lang.instrument` quanto o uso da opção `-Xbootclasspath` apenas provêem o acesso a uma classe de sistema, possibilitando sua substituição. Porém FIRMI, além disso, necessita de uma API que permita manipular o *bytecode* das classes de RMI para inserir os ganchos nos fluxos de envio e recepção de requisições. Foi escolhido Javassist [Chiba 1998] por sua facilidade de uso e por já ter sido testada com sucesso no injetor Jaca [Martins et al. 2002].

4.2. Arquitetura

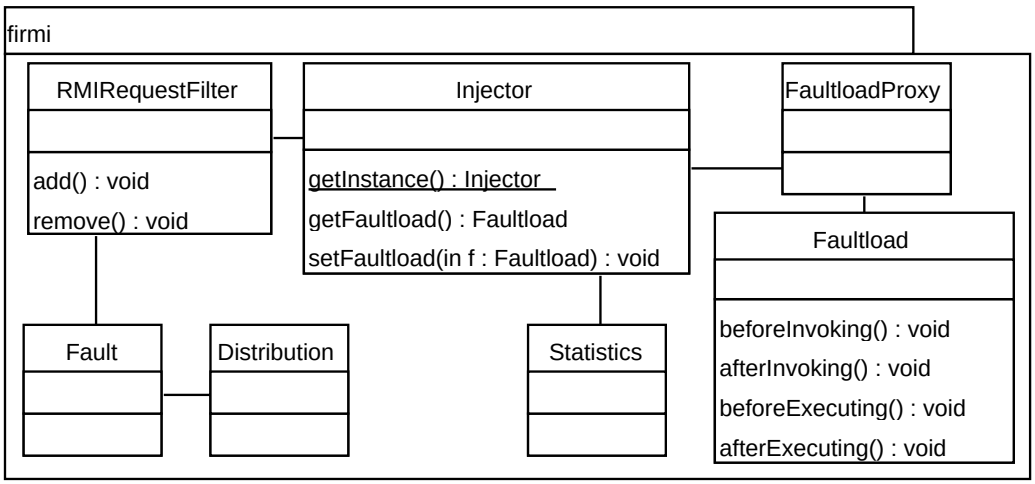

Figura 1. Diagrama de classes de FIRMI

A Figura 1 é um diagrama de classes de FIRMI. As principais classes do injetor são: `Injector`, `Faultload`, `Statistics` e `RMIRequestFilter`. A classe `Injector` controla as funcionalidades da ferramenta. Quando uma requisição RMI é interceptada pelos ganchos do mecanismo de instrumentação, o módulo irá acionar a classe de coleta de dados, registrar a requisição em log, invocar o `Faultload` para decidir a ação a ser tomada sobre a mensagem e finalmente injetar as falhas definidas pelo `Faultload` através da classe `RMIRequestFilter`. Para garantir que apenas uma instância de `Injector` seja criada, foi usado o padrão de projeto *Singleton* [Gamma et al. 1994]. Desta maneira, os demais componentes do sistema terão uma única visão do estado atual do injetor.

`Faultload` é a classe que representa um cenário de falhas. 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. Este comportamento é similar a noção de *listeners*, largamente utilizado na linguagem. A principal função desta classe é decidir, baseado nas informações do fluxo RMI, quando falhas devem ser ativadas ou interrompidas. FIRMI também define um conjunto de classes para permitir a emulação de erros causados por falhas de comunicação (Seção 3). `RMIRequestFilter` atua como um filtro de requisições RMI, injetando as falhas definidas por um `Faultload`. Para ativar ou interromper a ação de uma falha, deve-se adicionar/remover a mesma do filtro de requisições. Em FIRMI, cada falha é representada por uma classe (`CrashFault`, `LinkCrashFault`, `TimingFault` ou `NetworkPartitionFault`). Assim, quando uma requisição RMI for submetida ao filtro, todas as falhas ativas serão aplicadas sobre a mensagem.

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. Durante um experimento, as atividades realizadas por *faultloads* são monitoradas através da classe `FaultloadProxy`. A classe segue o padrão de projeto *Proxy* [Gamma et al. 1994]. Desta maneira, quando uma requisição RMI é percebida por FIRMI, o *proxy* coleta informações da execução do cenário de falhas ativo e atualiza as medidas contidas na classe `Statistics`.

4.3. Cenários de Falhas

A descrição de cenários de falhas é feita através da criação de classes Java. Como uma carga de falhas é dependente da aplicação sob teste e da carga de trabalho definida, todo módulo de cenário de falhas deve estender a classe `Faultload` para que o injetor possa interagir com o módulo de maneira padronizada. A classe `Faultload` é um *listener* de requisições RMI, possuindo métodos que indicam o fluxo de mensagens do protocolo. Os métodos `beforeInvoking` e `afterInvoking` são invocados respectivamente antes e após uma requisição no nodo cliente. `beforeExecuting` e `afterExecuting` são similares aos métodos anteriores, porém são invocados no nodo servidor na execução de uma requisição. Todos os métodos da classe `Faultload` possuem, praticamente, o mesmo conjunto de argumentos: a referência remota usada, o método sendo invocado, os valores dos parâmetros deste método e, após a execução de uma requisição, o valor de retorno do processamento.

Uma das vantagens da abordagem adotada para a descrição de cenários de falhas é a injeção de falhas baseada tanto no fluxo de mensagens quanto no conteúdo das mesmas, permitindo a validação de aplicações com precisão, já que falhas podem ser ativadas em pontos específicos no fluxo de execução de tais aplicações. É importante salientar que *faultloads* são responsáveis pela ativação de falhas e não pela sua injeção. FIRMI possui uma API para a injeção de falhas composta da classe `RMIRequestFilter` e por classes que modelam o comportamento exibido por RMI na presença de falhas de comunicação. *Faultloads*, após decidir pela ativação de uma falha, devem primeiramente criar um objeto do tipo desejado (`CrashFault`, `LinkCrashFault`, `TimingFault` ou `NetworkPartitionFault`) e então registrar o mesmo junto a classe `RMIRequestFilter`, a qual irá realizar o processo de injeção de falhas, gerenciando os aspectos de cada falha de acordo com o modelo adotado pela ferramenta.

Também é possível associar distribuições de probabilidade à ativação de falhas (classe `Distribution`), permitindo a especificação de cenários de falhas representativos, já que distribuições de probabilidade são usadas para modelar comportamentos observados na realidade.

Na Seção 3 foi definida a relação entre as falhas do sistema de comunicação e como estas influenciam os componentes de maior nível de abstração. FIRMI, através de sua API, deixa transparente tais detalhes do protocolo RMI, possibilitando a engenheiros de teste reproduzir situações inerentes a sistemas distribuídos sem a necessidade de dominar os detalhes específicos de RMI. Além disso, a arquitetura do injetor e as características inerentes a linguagem Java permitem que falhas sejam ativadas com base em 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.

5. Integração com outras plataformas de desenvolvimento

Paradigmas de desenvolvimento focadas unicamente na programação de sistemas (*Extreme Programming* [Beck and Andres 2004]) tem se tornado populares. O desenvolvimento baseado em testes (*Test-Driven Development*) é um dos pontos atacados por estas metodologias. A abordagem consiste no processo contínuo de desenvolvimento e teste, caracterizando uma metodologia bastante dinâmica. Consequentemente, são necessárias ferramentas de apoio ao desenvolvimento de aplicações capazes de agilizar este processo.

O modelo adotado em FIRMI 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. O uso de linguagens de *script* como BSF (Seção 5.1), de *frameworks* para teste de aplicações como JUnit (Seção 5.2) e o uso de ambientes de desenvolvimento como ANT (Seção 5.3), aceleram o processo de desenvolvimento de aplicações baseadas em Java. Como o objetivo de FIRMI também é a validação de sistemas, o processo de integração do injetor permite que as vantagens associadas à técnica de injeção de falhas sejam incorporadas de maneira natural às plataformas citadas.

5.1. Integração com Linguagens de *Script*

Scripts são frequentemente usados para estender funcionalidades de aplicações devido a sua facilidade na descrição de algoritmos. A integração de FIRMI com linguagens de *script* visa o mesmo objetivo, estender os mecanismos para a descrição dos módulos de cenários de falhas e assim agilizar o processo de avaliação de aplicações.

BSF [Apache Software Foundation 2002] é uma abstração para o uso de linguagens de *script*. A plataforma permite que *scripts* sejam invocados a partir de aplicações Java e que aplicações Java executem *scripts*. É importante salientar que BSF é um framework composto por uma série de classes abstratas que devem ser implementadas pelas reais linguagens. BeanShell [Niemeyer and Knudsen 2000], Jython [Pedroni and Rappin 2002] são linguagens bem conhecidas e compatíveis com este padrão. `javax.script` é um pacote nativo de Java que implementa funcionalidades similares ao *framework* BSF.

A especificação JSR-223 [Java Community Process 2005] define os requisitos para a padronização do suporte ao uso de *scripts* a partir da linguagem Java (o pacote será incluído na versão 1.6 da Linguagem).

Os *frameworks* mencionados nesta seção possibilitam a descrição de cenários de falhas em várias linguagens de *script* de maneira transparente, diminuindo a curva de aprendizado para o uso do injetor de falhas.

5.2. Integração com JUnit

A integração de FIRMI com o *framework* JUnit [Gamma and Beck 2001] introduz a noção de injeção de falhas na fase de teste de aplicações distribuídas, estendendo a definição de casos de teste. A integração da ferramenta permite reproduzir situações inerentes a sistemas distribuídos como colapso de nodos, particionamento de rede e falhas em canais de comunicação. Assim, agilizando o processo de avaliação de aplicações e de seus mecanismos de tolerância a falhas. Além disso, as medidas estatísticas coletadas pelo injetor juntamente com as medidas obtidas pelo próprio ambiente de teste auxiliam a caracterização das aplicações na presença de falhas.

Em JUnit, um caso de teste é modelado através da classe `TestCase`. Casos de teste podem ser agrupados através da classe `TestSuite`. Para conduzir testes em JUnit deve-se criar uma classe que estenda `TestCase`, criar um método para cada teste que se deseja executar e prefixar o nome destes métodos com a palavra `test`. Em cada método deve-se validar o resultado das operações para indicar ao *framework* se o teste foi bem sucedido ou não. Finalmente, os testes devem ser agrupados em um `TestSuite` e executados através de um *runner*. Assim, para a integração do injetor, foi criado o pacote `firmi.junit.framework` e estendidas as classes `TestCase` e `TestSuite`. A Figura 2 mostra a estrutura das novas classes desenvolvidas. A extensão possibilita definir cenários de falhas para um caso de teste específico, para um conjunto de casos de teste e também permite combinar as duas maneiras anteriores. Toda a interação com FIRMI é feita pelas classes estendidas, permitindo ao engenheiro de teste focar unicamente na especificação dos *faultloads* a serem utilizados.

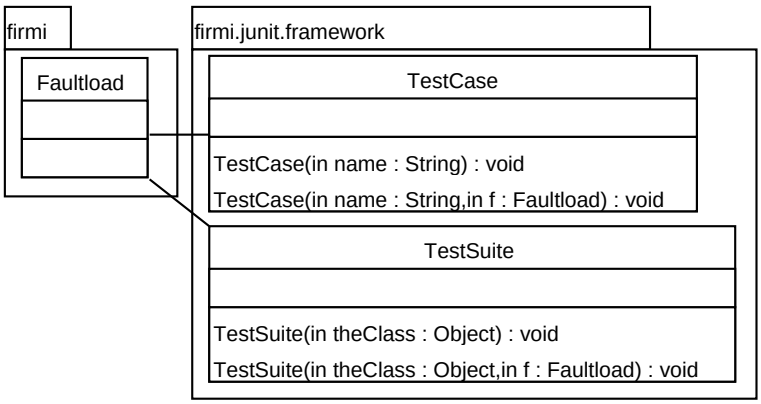

Figura 2. Extensão desenvolvida para a plataforma JUnit

Na classe `TestCase` foram alterados os construtores para aceitar um objeto `Faultload` como parâmetro. A modificação permite que seja definido um cenário de

falhas específico para este caso de teste, o que diminui a complexidade do módulo e permite a criação de cenários de falhas mais simples e claros. Também é possível definir um cenário de falhas para um conjunto de casos de teste. As modificações realizadas na classe `TestSuite` são similares as modificações da classe `TestCase`, sendo alterados apenas os construtores do módulo. Esta opção é útil quando deseja-se injetar um determinado perfil de falhas a um conjunto de testes relacionados. Ainda é possível combinar a definição de cenários de falhas para um único ou um conjunto de casos de teste. Deste modo, cenários de falhas podem ser associados a conjuntos de casos de teste, e para situações específicas, pode-se definir um cenário de falhas para um em especial.

```
1  public class MyTestCase extends firmi.junit.framework.TestCase {
2    public MyTestCase(String name)              {super(name);}
3    public MyTestCase(String name, Faultload f) {super(name, f);}
4
5    public void test1() throws Exception {...}
6    public void test2() throws Exception {...}
7    public void test3() throws Exception {...}
8
9    public static Test suite() {
10     TestSuite suite = new TestSuite();
11     suite.addTest(new MyTestCase("test1"));
12     suite.addTest(new MyTestCase("test2", new MyFaultload1()));
13     suite.addTest(new MyTestCase("test3", new MyFaultload2()));
14     return suite;
15   }
16 }
```

Figura 3. Caso de teste em JUnit com as classes estendidas por FIRMI

A Figura 3 mostra a estrutura para a definição de casos de teste usando as classes estendidas por FIRMI. A classe `MyTestCase` (linha 1) é a classe que representa um caso de teste. A classe estende `firmi.junit.framework.TestCase` (linha 1) para suportar a especificação de *faultloads* individuais para cada teste. `test1` (linha 5), `test2` (linha 6) e `test3` (linha 7) são métodos que representam testes a serem executados. `MyFaultload1` (linha 12) e `MyFaultload2` (linha 13) são módulos de cenários de falhas. O código mostra que a extensão de FIRMI possibilita a criação de casos de teste de forma convencional (linha 11) e também permite associar módulos de cenários de falhas a casos de teste específicos (linhas 12 e 13), oferecendo ao engenheiro de teste um maior controle em relação ao momento em que falhas deverão ser inseridas no sistema sob teste.

5.3. Integração com ANT

Para que FIRMI seja configurado a fim de injetar falhas em uma aplicação alvo é necessário fornecer informações adicionais ao ambiente Java. Entre estas informações estão a alteração do `bootclasspath` para conter os pacotes do injetor e as classes instrumentadas. Também é necessário indicar o módulo de cenário de falhas que deverá ser usado. ANT [Apache Software Foundation 2000] é um ambiente de execução similar a um Makefile em UNIX, porém destinado a aplicações Java. A integração da ferramenta a este ambiente de desenvolvimento facilita o processo de uso do injetor, ocultando detalhes de configuração para o engenheiro de teste.

O *script* ANT é um arquivo no formato XML organizado em projetos (*project*), alvos (*targets*) e tarefas (*tasks*). O projeto define as informações da aplicação sendo de-

senvolvida como nome, descrição e diretório base para execução de comandos. Alvos são a maneira com que um desenvolvedor organiza seu *script*. Pode-se criar um alvo para compilar uma aplicação, para executá-la ou verificar dependências externas. Cada alvo definido pelo usuário executa suas ações através de tarefas. Tarefas são estruturas pré-definidas por ANT. Existem tarefas para compilação (`javac`), execução (`java`) até tarefas para usar programas externos como FTP, SCP, CVS (*Control Version System*) e para o próprio ambiente JUnit.

Cada tarefa em ANT é associada a uma classe Java. Como é desejado integrar o injetor à execução de aplicações e ao ambiente JUnit, as classes associadas a estas tarefas foram estendidas para suportar as informações adicionais à execução de FIRMI. No caso, `java` é a tarefa responsável pela execução de uma aplicação e `junit` a responsável pela execução de casos de teste, sendo as duas classes os módulos estendidos pela ferramenta. A Figura 4 mostra o formato da tarefa `firmi-java`. `firmi-java` suporta os mesmos atributos da tarefa `java` como `classname`, `classpath` etc, com adição das informações para associar um módulo de cenário de falhas. O atributo `faultloadclasspath` indica a localização do cenário de falhas, pois este não precisa estar inserido na mesma estrutura da aplicação sob teste. O atributo `faultload` define o nome da classe Java que representa o cenário de falhas. A tarefa também possibilita controlar a instrumentação das classes de sistema de RMI responsáveis pelos fluxos de envio e recepção de requisições, o que é feito através dos atributos `remoterefimpl` e `serverrefimpl` (Seção 4.1). A integração com ANT elimina a necessidade de informar os pacotes do injetor para a JVM, pois são localizados e configurados automaticamente pela extensão desenvolvida.

```
1 <firmi-java
2   faultloadclasspath="classpath para o faultload"
3   faultload="classe Java do faultload"
4   remoterefimpl="implementação da interface RMI RemoteRef"
5   serverrefimpl="implementação da interface RMI ServerRef"
6   ...>
7     <!-- demais opções da tarefa java -->
8 </firmi-java>
```

Figura 4. Extensão da tarefa java

Para a extensão de JUnit foi criada a tarefa `firmi-junit` (Figura 5). Seguindo a idéia usada em `firmi-java`, são suportados todos os atributos da tarefa `junit` com a adição dos atributos para a integração do injetor. Entretanto, como descrito na Seção 5.2, a associação de cenários de falhas com casos de teste é feita no próprio módulo de teste, não sendo necessário informar o caminho e o nome da classe para o cenário de falhas. Ainda é possível controlar a instrumentação das classes de sistema de RMI através dos atributos `remoterefimpl` e `serverrefimpl`.

```
1 <firmi-junit
2   remoterefimpl="implementação da interface RMI RemoteRef"
3   serverrefimpl="implementação da interface RMI ServerRef"
4   ...>
5     <!-- demais opções da tarefa junit -->
6 </firmi-junit>
```

Figura 5. Extensão da tarefa junit

A integração de FIRMI no ambiente ANT faz com que o uso do injetor ocorra sem maiores modificações ao ambiente já existente, facilitando ainda mais o uso da ferramenta por desenvolvedores e engenheiros de teste.

6. Experimentos de Injeção de Falhas

Para demonstrar uma campanha de injeção de falhas com FIRMI, é usado o ambiente de computação em grade OurGrid [OurGrid 2005]. Não é objetivo deste trabalho validar todos os aspectos do ambiente, mas sim mostrar como o injetor proposto pode ser usado na validação de aplicações distribuídas baseadas em RMI. OurGrid é baseado no conceito de *Bag-of-Tasks*, aplicações paralelas onde as tarefas são independentes, ou seja, tarefas não se comunicam entre si para completar sua computação. OurGrid é formado por três componentes: `MyGrid`, `Peer` e Agente. `MyGrid` é o componente central do ambiente, sendo responsável pela coordenação e escalonamento de *jobs* [1]. *Peers* são responsáveis pela organização dos nodos de um dado domínio administrativo, atuando como um provedor de nodos disponíveis para `MyGrid`. Agentes são os componentes que executam as tarefas escalonadas pelo `MyGrid`.

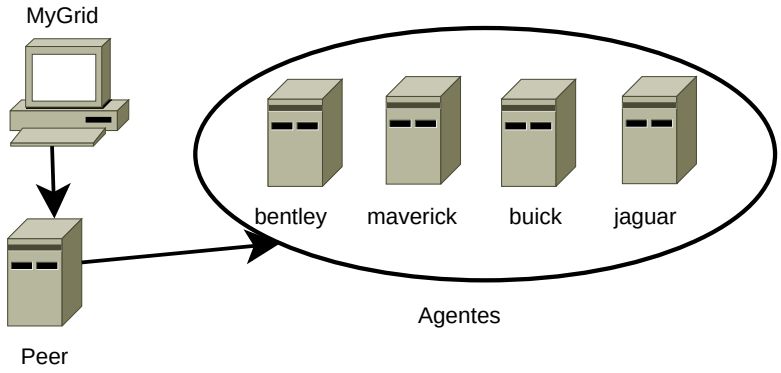

Figura 6. Organização do sistema OurGrid usado nos experimentos

A Figura 6 mostra a topologia criada para a condução dos experimentos. Como o objetivo é validar as funcionalidades do ambiente OurGrid, e não das aplicações executadas pelo mesmo, foi usada uma aplicação sintética da própria distribuição do sistema que realiza o fatorial de um número. Para a condução dos experimentos, FIRMI foi adicionado nos ambientes de execução dos agentes e também no `Peer` através da alteração dos *scripts* de inicialização de tais componentes para incluir as opções necessárias ao injetor.

6.1. Localização de Agents

Os nodos registrados no `Peer` são periodicamente monitorados a fim de manter uma lista atualizada de agentes disponíveis para a execução de tarefas. Uma análise dos logs gerados pelo injetor mostrou que este monitoramento é feito através da própria API de RMI, o que possibilita o uso de FIRMI para validar esta funcionalidade. Este caso de teste tem o objetivo de verificar se o `Peer` é capaz de detectar o colapso de agentes e consequentemente removê-los da lista de nodos disponíveis.

[1] Job é uma aplicação em OurGrid composta por tarefas independentes

O nodo `buick` foi escolhido para executar o injetor e emular o colapso desejado. Para especificar a carga de falhas é necessário decidir o momento em que a falha deve ser ativada. A análise dos logs mostrou que o método `UserAgentServer.ping` é usado pelo mecanismo de monitoramento de OurGrid. Assim, a carga de falhas para este experimento consiste na emulação do colapso do nodo `buick` após a terceira invocação de `ping`, o que permite o nodo ser primeiramente detectado pelo `Peer` para então sofrer o colapso. Para criar o módulo do cenário de falhas (Figura 7), foi criada a classe `BuickFaultload` (linha 1), que estende a classe `Faultload` da API de FIRMI, e definido o método `beforeExecuting` (linha 4), a fim de receber notificações da execução de métodos remotos. A falha de colapso (linha 2) é definida no momento da criação do *faultload* e ativada com base no fluxo de execução de RMI (linha 8).

```
1  public class BuickFaultload extends Faultload {
2    private CrashFault c = new CrashFault("c1");
3    ...
4    public void beforeExecuting(Remote obj,
5        Method method, Object[] params) {
6      times += 1;
7      if (times == 3 && method.getName().equals("ping"))
8        RMIRequestFilter.add(c);
9    }
10 }
```

Figura 7. Faultload para emular falha de colapso no nodo buick

O resultado do teste mostrou que o `Peer` foi capaz de detectar o colapso de `buick`. A validação foi feita visualmente através do comando `peer status` e também através dos próprios logs gerados por OurGrid (Figura 8). Este caso de teste ilustra a capacidade da ferramenta de injetar falhas em pontos específicos da execução de aplicações, dificilmente reproduzidas de forma manual sem a modificação direta do código fonte.

```
OurGrid Peer 3.2.1 is Up and Running
Local GuMs:
  bentley
  buick
  jaguar
  maverick
...
```

(a) antes do colapso de buick

```
OurGrid Peer 3.2.1 is Up and Running
Local GuMs:
  bentley
  jaguar
  maverick

...
```

(b) após o colapso de buick

Figura 8. Saída do comando peer status

6.2. Escalonamento de Tarefas

OurGrid implementa um mecanismo de tolerância a falhas que evita a perda de um *job* quando uma tarefa não é concluída por motivo de falha. Neste caso o escalonador irá executar a tarefa não concluída em outro nodo para então finalizar o *job* com sucesso. Este experimento irá exercitar esta funcionalidade com a definição de um *job* composto por 1000 tarefas, ou seja, todos os nodos deverão ser usados.

Neste experimento serão emuladas falhas de temporização (nodo `jaguar`) e colapso de *link* (nodo `maverick`) segundo uma distribuição de probabilidade uniforme

com taxa de erro de 5%, reproduzindo uma situação de congestionamento de rede onde pacotes são atrasados ou até perdidos, bem como falhas de componentes da infraestrutura de comunicação. A Figura 9(a) é o módulo de cenário de falhas usado para emular as falhas de temporização. Na linha 5 é definida a falha com atraso de 1 segundo por requisição, enquanto que na linha 7 é feita a ativação da mesma. O *faultload* para o nodo `maverick` (Figura 9(b)) é similar ao módulo anterior, mudando apenas o tipo de falha especificado. Note que para indicar o padrão de repetição baseado em uma distribuição de probabilidade, basta criar a classe correspondente e indicá-la na definição da falha (linha 6). Como o fluxo de requisições RMI não é relevante para a ativação das falhas, não é necessário definir nenhum método para a notificação de eventos.

```
1 public class JaguarFaultload
2     extends Faultload {
3   private TimingFault t;
4   public JaguarFaultload() {
5     t = new TimingFault(
6       "t1", 1000);
7     RMIRequestFilter.add(t);
8   }
9 }
```

(a) nodo jaguar

```
1 public class MaverickFaultload
2     extends Faultload {
3   private LinkCrashFault lc;
4   public MaverickFaultload() {
5     lc = new LinkCrashFault("lc1",
6       new UniformDistribution(0.05));
7     RMIRequestFilter.add(lc);
8   }
9 }
```

(b) nodo maverick

Figura 9. Faultloads usados neste experimento

nodos	tarefas		
	escalonadas	concluídas	falhas
bentley	212	212	0
buick	255	255	0
jaguar	255	255	0
maverick	278	278	0

(a) sem injeção de falhas

nodos	tarefas		
	escalonadas	concluídas	falhas
bentley	425	425	0
buick	522	522	0
jaguar	45	45	0
maverick	15	8	7

(b) com injeção de falhas

Figura 10. Resultado da execução do job com 1000 tarefas

Os resultados do experimento (Figura 10) mostram que os nodos `jaguar` e `maverick`, por apresentar falhas de temporização e colapso de *link*, causaram uma grande degradação no desempenho do sistema. O nodo `jaguar` foi capaz de executar apenas 45 tarefas, ou seja, 4.5% do total de tarefas da aplicação. No caso do nodo `maverick`, apenas 8 tarefas das 15 escalonadas foram processadas com sucesso (0.8%). Isso ocorre devido ao fato de que falhas de colapso de *link*, em comparação às falhas de temporização injetadas, causam maior indisponibilidade de nodos. Outro fator importante a ser considerado é que o tempo de execução de cada tarefa é muito baixo, tornando períodos de indisponibilidade mais críticos, já que muitas tarefas serão escalonadas para outros nodos. Apesar da queda de desempenho do sistema devido a falhas, nesta campanha de teste os mecanismos de tolerância a falhas do sistema alvo estão operando de acordo com a especificação. O experimento mostrou o comportamento sob falhas do sistema alvo e sua total cobertura das falhas injetadas.

7. Trabalhos Relacionados

Para diminuir a latência e permitir um maior controle sobre a manifestação da falha injetada, no lugar de introduzir falhas de memória e CPU várias ferramentas injetam diretamente falhas de comunicação. Mas nenhuma destas ferramentas apresenta a facilidade

de definir modelos de falhas adequados para validar especificamente aplicações baseadas em RMI.

DOCTOR [Han et al. 1995] injeta falhas de memória, CPU ou de comunicação em ambientes distribuídos de tempo real, que não são ambientes usuais para aplicações baseadas em RMI. ORCHESTRA [Dawson et al. 1996] foi usado para o teste da implementação de protocolos, especialmente TCP. A ferramenta é inserida na pilha de protocolos do sistema operacional, abaixo da camada onde reside o protocolo sob testes, e atua sobre cada mensagem que flui por esta pilha. Esta ferramenta inspirou ComFirm [Drebes et al. 2005], um módulo de kernel Linux que atua na camada de IP. ORCHESTRA e ComFirm poderiam ser usadas para avaliar aplicações RMI, porque tanto TCP como http, que suportam RMI, estão construídos sobre IP. Entretanto, por interceptar todas as mensagens de um nodo, a seleção de mensagens de mais alto nível específicas de um dado processo exige um esforço muito grande na construção da carga de falhas apropriada. Para evitar este custo na avaliação de aplicações baseadas em UDP, FIONA [Gerchman et al. 2005] foi desenvolvido com modelo de falhas apropriado a este protocolo. Os injetores de FIONA atuam instrumentando classes de comunicação de sistema. A menor intrusividade temporal foi alcançada com a versão usando JVMTI [Jacques-Silva et al. 2004]. Uma ferramenta semelhante foi desenvolvida em paralelo na IBM [Farchi et al. 2004], também baseada na instrumentação da comunicação UDP. Como UDP não é base de RMI, nenhum dos injetores para UDP é apropriado para validar aplicações RMI.

Além de injetores, são encontrados ambientes de injeção de falhas para aplicações distribuídas. NFTAPE [Stott et al. 2000] usa injetores leves, um componente de software compacto e substituível. Entre esses injetores estão previstos componentes para falhas de comunicação, mas relatos de sua utilização não foram localizados. Loki [Chandra et al. 2004] apresenta uma arquitetura distribuída para testar aplicações em cenários com falhas em múltiplos nodos. Loki mantém um estado global entre os nodos do experimento. Gatilhos definidos neste estado global permitem ativar falhas nos processos, mas os injetores em si, assim como a carga de falhas, são deixados a cargo do engenheiro de teste. FIONA também apresenta componentes de controle e monitoramento distribuídos para permitir ativação de falhas em múltiplos nodos mas, ao contrário de Loki, não se baseia em um estado global mas apenas na troca de mensagens entre nodos.

A dificuldade de lidar com modelos de falhas de protocolos de baixo nível de abstração, a inadequação de ferramentas que instrumentam UDP para validar aplicações RMI, e a não disponibilidade de injetores para RMI foram motivações para a construção de FIRMI. Os ambientes providos por NFTAPE, Loki e FIONA não comportam integração com ambientes de teste, outra motivação do projeto de FIRMI, e portanto não estão sendo considerados no momento.

8. Conclusão

Aplicações com fortes requisitos de confiabilidade e disponibilidade devem ser submetidas a procedimentos de teste para validar suas funcionalidades tanto em condições normais de operação quanto em situações de falha. Injeção de falhas, em particular a injeção de falhas de comunicação por *software*, é uma técnica eficiente na validação de tais aplicações. No entanto, esta não é suficiente para exercitar todas as características de um dado sistema, o que não elimina a necessidade de outras técnicas já existentes de teste

de aplicações. Este trabalho apresentou a ferramenta FIRMI, um injetor de falhas para a avaliação de aplicações distribuídas baseadas em RMI que possui como uma de suas principais características a integração em diferentes ambientes de teste e desenvolvimento de aplicações, agregando as vantagens da técnica de injeção de falhas voltada a sistemas distribuídos para estes ambientes.

A descrição de cenários de falhas baseada em linguagens de *script* e a integração de FIRMI em JUnit e ANT permitem a engenheiros de teste usar a ferramenta de modo complementar nas fases de teste de aplicações e assim aumentar a cobertura dos testes empregados na validação de sistemas. Os experimentos conduzidos ilustram a capacidade da ferramenta de injetar falhas em pontos específicos da execução de uma aplicação, e também permite a definição de cenários de falhas mais genéricos que não dependam das características de uma aplicação em particular. Estes fatores fazem de FIRMI uma alternativa viável na validação de sistemas baseados em RMI.

Referências

Apache Software Foundation (2000). ANT. `http://ant.apache.org/`.

Apache Software Foundation (2002). BSF. `http://jakarta.apache.org/bsf`.

Beck, K. and Andres, C. (2004). *Extreme Programming Explained : Embrace Change*. Addison-Wesley Professional, Workingham, 2 edition.

Birman, K. P. (1996). *Building Secure and Reliable Network Applications*. Prentice Hall, 1 edition.

Chandra, R., Lefever, R., Joshi, K., Cukier, M., and Sanders, W. (2004). A global-state-triggered fault injector for distributed system evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605.

Chang, T. and Aharnad, M. (2004). GT-P2PRMI: improving middleware performance using peer-to-peer service replication. In *Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'04)*, pages 172–177, Suzhou, China. IEEE Computer Society Press.

Chen, C.-W., Chen, C.-K., Chen, J.-C., Ko, C.-T., Lee, J.-K., Lin, H.-W., and Wu, W.-J. (2004). Efficient support of java rmi over heterogeneous wireless networks. *IEEE International Conference on Communications*, 3(1):1391–1395.

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.

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.

Delamaro, M. E., Maldonado, J. C., and Mathur, A. P. (2001a). Interface mutation: An approach for integration testing. *IEEE Trans. Software Engineering*, 27(3):228–247.

Delamaro, M. E., Pezzè, M., Vincenzi, A. M. R., and Maldonado, J. C. (2001b). Mutant operator for testing concurrent java programs. In *Anais do 15th Simpósio Brasileiro de Engenharia de Software (SBES'2001)*, pages 386–391, Rio de Janeiro/RJ. SBC.

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

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

Gamma, E. and Beck, K. (2001). Junit. `http://www.junit.org`.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1st edition.

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

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

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

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

Java Community Process (2005). JSR-000223 scripting for the javatm platform. `http://jcp.org/aboutJava/communityprocess/pr/jsr223/`.

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.

Niemeyer, P. and Knudsen, J. (2000). *Learning Java*. O'Reilly, 1 edition.

OurGrid (2005). Ourgrid online manual. `http://www.ourgrid.org/`.

Pedroni, S. and Rappin, N. (2002). *Jython Essentials*. O'Reilly, 1 edition.

Stott, D. T., Floering, B., Kalbarczyk, Z., and Iyer, R. K. (2000). NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS 2000)*, pages 91–100, Chicago, Illinois, USA. IEEE Computer Society Press.

Sun Microsystems (1997). Java RMI specification. `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`.