Experimentos de Tolerância a Falhas em Java

Maria Lúcia B. Lisbôa
Instituto de Informática
llisboa @ inf.ufrgs.br

Werner Haetinger
CPGCC
wernerh @ inf.ufrgs.br

Gustavo Canto da Silva
CBCC
canto @ inf.ufrgs.br

Universidade Federal do Rio Grande do Sul
Instituto de Informática
Av. Bento Gonçalves, 9500 - Bloco IV
CEP 91501-970 Porto Alegre - RS

Resumo

No modelo de objetos, programas são estruturados a partir de componentes encapsulados e que interagem através de interfaces bem definidas. A interação dos componentes depende fortemente da estrutura adotada no programa ou sistema, bem como seu cenário de execução: seqüencial, paralelo ou distribuído. Entre as condições propícias à manifestação de uma falha, o meio-ambiente desempenha um papel importante e, portanto, deve ter a sua atuação bem delimitada. Um meio-ambiente desfavorável pode ocasionar diferenças de comportamento em duas cópias idênticas do mesmo software. Pequenas diferenças nas máquinas virtuais onde o software é executado podem ser suficientes para a manifestação de uma falha. É precisamente nas semelhanças e diferenças de diversas formas de interação de componentes e seus distintos ambientes de execução que este trabalho concentra seus experimentos, estudando a adequação da linguagem Java para a implementação de programas tolerantes a falhas.

Abstract

In the object model, programs consist of encapsulated components, which interact through well-defined interfaces. The interactions among the components depend on the current system or program architecture as well as the execution environment: sequential, parallel or distributed. Also aspects of the environment in which the system operates cannot be ignored by the system specification. An improper environment can trigger a component/interaction fault, so its important to bound the system influence over the running program. Even when executing replicas of the same base component, small differences in the system behavior can lead the program to a faulty state. This paper is actually about the similarities and differences among the several ways the components interact and their execution environment. A major goal is to test the adequacy of Java programming language and its several virtual machines to implement fault-tolerant applications.

1 Introdução

O desenvolvimento de software tolerante a falhas no modelo de orientação a objetos se apresenta como uma estratégia promissora para a construção de software com requisitos de alta confiabilidade [BUZ97], por razões inerentes ao próprio modelo. A habilidade de classes de herdar propriedades, de forma integral ou seletiva, a possibilidade de aumentar, modificar ou anular esta herança, viabiliza a programação por diferença [GOL83]. Apenas as propriedades distintas necessitam ser implementadas, mantendo intocadas as demais propriedades. Uma conseqüência imediata da programação por diferença é a manutenção da confiabilidade dos componentes já existentes, verificados e/ou testados pelo uso.

Falha de software é qualquer imperfeição presente no código executável de um programa, cuja manifestação possa implicar a diminuição de sua confiabilidade. Mesmo presente, uma falha pode passar despercebida durante todo o processo de desenvolvimento do software e mesmo por longos períodos de sua vida útil pode permanecer latente, caso não ocorram as condições propícias à sua manifestação. Entre as condições propícias à manifestação de uma falha, o meio-ambiente desempenha um papel importante e, portanto, deve ter a sua atuação bem delimitada. Um meio-ambiente desfavorável pode ocasionar diferenças de comportamento em duas cópias idênticas do mesmo software. Por exemplo, pequenas diferenças nas máquinas virtuais onde o software é executado podem ser suficientes para a manifestação de uma falha.

É precisamente nas semelhanças e diferenças de distintos ambientes de execução que este trabalho concentra seus experimentos, estudando a adequação da linguagem Java para a implementação de programas tolerantes a falhas. Para a obtenção de componentes redundantes, uma mesma funcionalidade é implementada de três formas distintas, através de classes Java, usando diversidade de métodos em uma única classe e diversidade de classes, separadamente compiladas. Estes componentes são submetidos à execução em diferentes máquinas virtuais, com o intento de observar diferenças no meio-ambiente de execução Java. Distintas formas de execução de componentes - seqüencial e concorrente, também são objeto de estudo, para fins de observação dos tempos de execução.

2 Componentes de programas

Programas são organizados a partir de componentes individuais, que implementam as diversas funcionalidades previstas na especificação do programa ou sistema. Componentes de software oferecem a possibilidade de múltiplas execuções independentes. A informação exigida para a execução de um componente pode ser decomposta em duas partes: uma parte permanente, que consiste da estrutura dos dados e da seqüência de instruções a serem executadas, e uma parte temporária, que compreende valores de dados e outras informações contextuais que variam a cada execução do componente.

A parte permanente ou estrutural, uma vez determinada pelo projeto do componente, permanece invariante, e as correspondentes estruturas de dados e as instruções são reutilizadas ao longo das diversas execuções. Além disso, podem ser transportadas para outros programas ou sistemas sem exigir alterações. A parte temporária ou comportamental é ditada pela interação dos componentes e determina, a cada instante, o estado da execução do componente, incluindo valores de dados, de registradores e outras informações mantidas durante o processo de execução. Portanto, a parte temporária é específica de cada interação entre componentes e se constitui no diferencial mais importante entre execuções de um mesmo componente[LIS97a]

Mais especificamente, no modelo de objetos, os componentes são representados por classes de objetos, que, sob o ponto de vista estrutural, são considerados como componentes atômicos de um programa, uma vez que classes gozam da propriedade de encapsulamento. Sob o ponto de vista de execução, os objetos (instanciados a partir das classes) respondem

pelo estado da execução. Outrossim, tão importante quanto cada componente individual de um programa é o modo como estes componentes interagem para atender os serviços esperados do programa [SHA95]. A interação dos componentes depende fortemente da estrutura adotada no programa ou sistema, bem como seu cenário de execução: seqüencial, paralelo ou distribuído. A seguir são sumarizadas as principais formas de concepção, execução e interação de componentes.

2.1 Concepção de objetos

Um objeto é uma abstração descrita em uma classe e delineada a partir de seus requisitos funcionais - o que ele deve fazer - e não funcionais - o que ele deve observar além de sua funcionalidade específica. Entre os requisitos não funcionais de um sistema incluem-se: adaptabilidade, interoperabilidade, eficiência, testabilidade, reutilização e confiabilidade. De acordo com Bushmann et al. [BUS96], a confiabilidade inclui os aspectos de robustez e tolerância a falhas.

Uma abstração de um objeto começa a ser esboçada a partir de dados, serviços, interfaces e, antecipando a sua utilização, o ambiente de execução. O projeto de tolerância a falhas deve ser feito neste momento, principalmente quando envolver projeto diversitário. A tolerância a falhas em software usando redundância de componentes exige nesta fase a implementação de métodos ou objetos diversificados, derivados ou não da mesma classe ancestral, e a implementação de técnicas de gerenciamento de redundância.

Técnicas de gerenciamento de redundância tem por objetivo controlar a execução de cada serviço solicitado a objetos replicados ou diversificados, fornecendo uma única resposta, supostamente confiável, ao cliente desse serviço. A forma de interação dos objetos envolvidos e o ambiente de execução desempenham um papel muito importante na questão de gerenciamento da redundância.

2.2 Cenários de execução: seqüencial e concorrente

Na execução seqüencial e centralizada, um componente requisita a execução de outro componente através do envio de uma mensagem, interrompendo a sua execução enquanto aguarda o término da execução do componente requisitado. A figura 1 esquematiza o fluxo de execução por requisição e serve como cenário básico para introduzir a idéia de componentes com propriedades de tolerância a falhas.

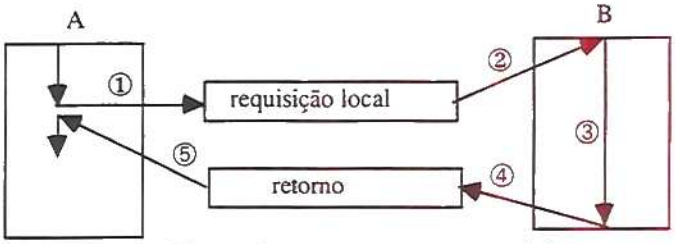

Figura 1: Fluxo de execução por requisição

Neste cenário destacam-se os seguintes pontos demarcados na figura 1 e que indicam momentos de execução e suas possibilidades de falha.

(1) O componente A requisita a execução do componente B, por meio de uma mensagem parametrizada. Falha na mensagem de requisição. Ex. Nome incorreto do método.

(2) O componente B recebe a requisição, identifica o método destinatário da mensagem e inicia a sua execução. Falha nos parâmetros de requisição. Ex. Tipo incorreto de parâmetro.

(3) O componente B executa as operações contidas no método selecionado. Falha em operação interna de B. Ex. divisão por zero.

(4) O componente B termina a sua execução e retorna resultados. Execução com erro. Ex. Retorna indicação de erro.
(5) O componente A recebe os resultados da execução de B e retoma a sua execução no ponto de interrupção. Resultados incorretos da execução de B. Ex. Valor fora dos limites.

A linguagem de implementação pode contribuir para prevenir a ocorrência das falhas mencionadas nos momentos (1) a (3), caso possua mecanismos como verificação estática e/ou dinâmica de tipos e mecanismos de tratamento de exceções. No caso de Java, estas características estão presentes. Já os tipos de falhas descritas em (4) e (5) devem ser abordados através de técnicas de programação tolerante a falhas, visto que o componente requisitado foi incapaz de fornecer o serviço solicitado. A solução consiste em solicitar o mesmo serviço a outro componente, pressupondo a existência de redundância.

A execução concorrente de componentes estende o cenário seqüencial para vários fluxos de execução: vários componentes de um mesmo programa podem ser simultaneamente executados (ou dar esta impressão ao cliente). Ou ainda, o mesmo componente pode ser submetido a mais de um fluxo de execução, atendendo a diferentes requisições. Em Java, a programação concorrente pode ser implementada através de *threads* de execução. Em alguns ambientes de execução Java, esta forma de implementação possui atualmente algumas importantes limitações, tais como: (a) dificuldade de obtenção de paralelismo real, visto que todos os *threads* são executados em um mesmo processador; (b) ao final da execução de um *thread*, o objeto é destruído, necessitando ser criada uma nova instância para posterior execução[LIN96].

3 Estudo de caso

Tendo por objetivo principal observar diferenças na forma de interação dos componentes e no meio-ambiente de execução Java, uma mesma funcionalidade é implementada de três formas distintas, por diversidade de métodos e diversidade de classes. A partir destas classes, foram estruturados três programas, com distintas arquiteturas: o primeiro utiliza uma única classe, com diversidade de métodos, o segundo reúne as três classes, separadamente compiladas e o terceiro utiliza as mesmas classes, com *threads* de execução. Estes programas foram submetidos à compilação e execução em diferentes máquinas virtuais, com o intento de medir os respectivos tempos de execução.

Para a implementação da redundância foram selecionados três métodos numéricos de interpolação polinomial. Esta escolha deve-se à facilidade de implementação dos algoritmos de interpolação e a grande probabilidade de divergência de resultados, por problemas inerentes aos próprios métodos bem como erros gerados no processo de cálculo. Tais características são úteis para testes de algoritmos de votação e de aceitação de resultados, usados pelas técnicas de programação em n-versões e blocos de recuperação.

Experimento 1: Execução seqüencial

a) O primeiro programa, executado como um 'applet', utiliza diversidade de métodos: cada algoritmo de interpolação é implementado como um dos métodos de uma mesma classe. Os três métodos usam idênticos argumentos:

```
r1= this.lagrange(vet,10,aux.doubleValue());
r2= this.linear(vet,10,aux.doubleValue());
r3= this.newton(vet,10,aux.doubleValue());
```

b) No segundo programa, os objetos diversificados são implementados pelas classes Linear, Lagrange e Newton. A classe principal instancia e executa os objetos diversificados:

```
double r1= newt.method(vet,n,aux.doubleValue());
double r2= lag.method(vet,n,aux.doubleValue());
double r3= lin.method(vet,n,aux.doubleValue());}
```

Experimento 2: Execução concorrente

Para este exemplo, o programa (c) foi estruturado com as mesmas três classes do programa (b), porém os objetos são executados através de *threads*. A classe principal continua responsável pela instanciação dos objetos, execução dos métodos e coleta dos resultados.

```
newt= new newton(this,aux.doubleValue(),vet,n);
lag= new lagrange(this,aux.doubleValue(),vet,n);
lin= new linear(this,aux.doubleValue(),vet,n);
lag.start();
newt.start();
lin.start();
```

3.1 Resultados dos experimentos

Inicialmente, os programas (a) e (b) foram submetidos à execução seqüencial em diferentes ambientes, usando máquinas semelhantes em configuração e foram coletados os tempos de tempos de execução. Os experimentos para medição de tempo foram realizados com 50 execuções de cada método ou objeto, para potencializar eventuais divergências, bem como minimizar o efeito do tempo de carga do programa, observado na primeira execução.

O efeito do tempo de carga não pode ser desprezado, como pode ser visto na Tabela 1, que registra o tempo de uma execução e o tempo médio de 50 execuções do mesmo método ou objeto. Os resultados registrados na tabela 1 forma obtidos através de uma média de 10 ativações, para minimizar a influência do momento de execução.

Tabela 1 - Tempo médio de execução

Ambiente	Média - 1 execução	Média - 50 execuções
PC/JDK 1.1.	331 ms	13 ms
PC/ Visual J++	204 ms	3.4 ms
PC/Borland Suite/Java	210 ms	10 ms
PC/ Visual Café	29 ms	4.4 ms

A seguir, foram repetidos os testes com execução paralela, utilizando *threads*. Em Java*, a execução concorrente é obtida através de herança da classe *Thread*. Um *thread* de execução deve ser explicitamente ativado para um determinado objeto e, quando encerra a execução, normal ou excepcional, o objeto é destruído. Portanto, para a obtenção de diversos *threads* de execução, deve ser criado um novo objeto e feita uma nova ativação. Os resultados obtidos encontram-se esquematizados na tabela 2.

Tabela 2 - Execução seqüencial e concorrente

Ambiente	Programa (a) - execução seqüencial	Programa (b) execução seqüencial	Programa (c) Execução concorrente
PC/JDK 1.1.	3.7 ms	3.2 ms	32 ms
PC/ Visual J++	3.4 ms	2.2 ms	3.4 ms
PC/Borland Suite/Java	10 ms	5.5 ms	21 ms
PC/ Visual Café	4.4 ms	2.2 ms	4.4 ms

A linguagem Java se caracteriza pela universalidade do código gerado pelos diferentes tradutores e que pode ser executado em qualquer ambiente, sem necessidade de alteração e mesmo de recompilação. Durante os testes o mesmo código fonte foi usado, porém algumas vezes foi recompilado nos diferentes ambientes, para observar se o tamanho do código gerado apresentava diferenças. Constatou-se que o tamanho do código gerado nos diversos ambientes

* Ambiente JDK1.1

não apresentava diferenças significativas, principalmente devido ao pequeno número de classes que compõem os programas do experimento.

Quanto à forma de execução, seqüencial ou concorrente, os experimentos para medição de tempo mostraram divergências significativas nos diferentes ambientes. Os tempos obtidos mostraram que alguns ambientes praticamente não apresentaram diferenças nos tempos de execução seqüencial e concorrente, a exemplo de J++ e Visual Café, enquanto outros apresentaram o tempo de execução concorrente superior ao tempo de execução seqüencial. Hipóteses para justificar estas diferenças são: (a) o tempo de criação e destruição dos objetos submetidos a *threads* de execução, e, (b) tipo de suporte para a execução dos *threads*, oferecido pela máquina virtual ou sistema.

4 Conclusões e trabalhos futuros

Os experimentos sobre a adequação da linguagem Java para fins de tolerância a falhas têm mostrado resultados encorajadores em relação à portabilidade da linguagem e à existência de mecanismos que possibilitam a implementação segura e transparente de requisitos não-funcionais. Em trabalhos anteriores [LIS97b] foram realizados testes com carga dinâmica de classes e com reflexão computacional [HAE98].

Neste trabalho, os experimentos de medição de tempos de execução foram feitos com informações obtidas a partir do relógio do sistema, cuja precisão pode não ser suficiente para capturar diferenças de tempo em programas pequenos, com os do presente estudo de caso.

Na continuidade, serão feitos experimentos com programas maiores e com maior número de interações e execuções e novos experimentos envolvendo componentes:

(a) com execução distribuída usando sockets e RMI - Remote Method Invocation. O objetivo dos testes, além da medição de tempos de execução, é a avaliação de chamadas remotas parametrizadas, com diferentes tipos de dados.

(b) com componentes auto-protegidos. O mecanismo mais importante de auto-proteção é o mecanismo de *tratamento de exceções*. Através dele, a possibilidade de ocorrência de vários tipos de falhas pode ser antecipada, erros podem ser detectados durante o processo de execução e providências locais (internas ao objeto) podem ser tomadas com vistas à continuidade do serviço, ou mesmo, quando não é possível resolver internamente, o objeto pode sinalizar ao cliente de seus serviços a ocorrência de uma situação excepcional que o impede de fornecer o serviço solicitado.

Bibliografia

[BUS96] BUSCHMANN, F. et al. A System of Patterns: pattern-oriented software architecture. John Wiley & Sons, England, 1996.

[BUZ97] BUZATO, L. E.; RUBIRA, C.M.F.; LISBOA, M. L. A reflective Object-oriented architecture for developing fault-tolerant software. Jornal Of the Brazilian Computer Society, Vol. 4, No. 2; November 1997, p. 39-48.

[GOL83] GOLDBERG, A. The influence of an object-oriented language on the programming environments. In: ACM COMPUTER SCIENCE CONFERENCE, 1983, Orlando, Florida, USA. **Proceedings**...New York: ACM, 1983. p. 35-54.

[HAE98] HAETINGER, W.; LISBÔA, M.L. Substituição dinâmica de classes com validação de objetos. Trabalho submetido ao I Workshop de Tolerância a Falhas, Porto Alegre, maio de 1998.

[LIN96] LINDEN. P. v. Just Java. Sunsoft Press, Mountain View, CA, USA, 1996.

[LIS97a] LISBOA, M. L. Arquiteturas de meta-nível. Tutorial. Simpósio Brasileiro de Engenharia de Software, Fortaleza, CE, 1997.

[LIS97b] LISBÔA, M.L.; HAETINGER, W. Troca Dinâmica de Componentes: problemas e soluções no modelo OO. Argentine Simposium on Object-Orientation, Buenos Aires, anais pp.67-75, setembro, 1997.

[SHA95] SHAW, M; GARLAN, D. Formulations and Formalisms in Software Architecture. Lecture Notes in Computer Science, n. 1000, Berlin: Springer, 1995.