Uma Ferramenta para Teste de Aplicações Distribuídas em Redes sem Fio

César Francisco de Moura Couto - cesarfmc@pucmg.br
Orientador: Marco Túlio de Oliveira Valente - mtov@pucminas.br

Departamento de Ciência da Computação
Pontifícia Universidade Católica de Minas Gerais

Resumo: Este artigo descreve uma ferramenta implementada e projetada para testar aplicações distribuídas em redes sem fio. A ferramenta descrita é capaz de simular eventos típicos dessas redes, tais como desconexões e variações na taxa de transmissão de dados. Além disso, a ferramenta é capaz de simular a ocorrência destes eventos em uma rede local, de forma a facilitar o teste de aplicações distribuídas no mesmo ambiente utilizado em seu desenvolvimento. A ferramenta proposta, chamada RMITK (ou RMI *Test Toolkit*), permite o teste de aplicações distribuídas desenvolvidas em Java usando-se o pacote RMI desta linguagem. Java RMI (ou *Remote Method Invocation*) é um sistema de chamada remota de métodos usado freqüentemente por aplicações distribuídas desenvolvidas em Java.

Palavras-Chave: Computação Móvel, Java RMI, Teste de Sistemas

1 Introdução

Atualmente, existe uma demanda crescente pela construção de aplicações para ambientes de computação móvel. Esta demanda deve-se à popularização e barateamento das tecnologias de *hardware* embutido, telecomunicações sem fio e redes móveis de computadores [2, 4]. Telefones celulares vêm gradativamente incorporando capacidade de processamento. Já existem, por exemplo, diversos modelos de celulares que possuem embutida uma Máquina Virtual Java e que, portanto, são capazes de executarem programas nessa linguagem [1,2,12].

Assim, é importante que as ferramentas usadas no desenvolvimento de *software* sejam adaptadas a este novo tipo de ambiente, bem como novas ferramentas sejam propostas para lidar com características típicas do mesmo [3,6,7,10]. Duas destas características são particularmente importantes: a existência de desconexões freqüentes e a ocorrência de flutuações na largura de banda da rede. Desconexões ocorrem pela própria mobilidade dos dispositivos computacionais, que podem sair da faixa de alcance das estações base de uma rede sem fio. Além disso, desconexões voluntárias são freqüentemente realizadas para economizar custos de comunicação e bateria. Já flutuações na largura de banda ocorrem pelas características do próprio meio de comunicação sem fio, o qual é mais sujeito a erros de transmissão e a diversos tipos de interferência [3,6].

Cada vez mais, aplicações para redes sem fio utilizam-se então de técnicas para atenuar os efeitos destas duas características sobre as mesmas. Em geral, estas técnicas assumem a existência de *caches* nos clientes para armazenar informações localmente e, portanto, reduzir os efeitos dos problemas de comunicação. Assim, qualquer ambiente para teste de

aplicações distribuídas sem fio deve ser capaz de simular a ocorrência dos problemas de comunicação típicos destas redes, de forma a verificar se as mesmas atendem adequadamente aos requisitos definidos pelos usuários. Logo, estas aplicações não devem ser testadas apenas em um ambiente tradicional de rede local, visto que desconexões e variações de largura de banda não são eventos observáveis neste tipo de rede.

Daí então a importância de se desenvolver uma ferramenta que simule os problemas de comunicação de uma rede sem fio e que possa apoiar o teste de aplicações distribuídas desenvolvidas para este novo paradigma de computação. Ressalte-se que a fase de testes em geral demanda cerca de 25% por cento do tempo total de desenvolvimento de um projeto e, portanto, qualquer ferramenta que procure automatizá-la ou reduzir seus custos é de fundamental importância [5,9,11].

Este artigo tem como objetivo descrever uma ferramenta para teste de aplicações distribuídas em redes sem fio, chamada RMITK (ou RMI *Test Toolkit*), capaz de simular eventos típicos destas redes, tais como desconexões e variações na taxa de transmissão de dados. Além disso, a ferramenta é capaz de simular a ocorrência destes eventos em uma rede local, de forma a viabilizar o teste das aplicações no mesmo ambiente utilizado em seu desenvolvimento. A ferramenta tem como objetivo final ser empregada para teste de uma nova geração de *software* para computadores móveis que encontra-se em desenvolvimento.

A ferramenta RMITK foi desenvolvida em Java e, portanto, deverá ser utilizada para testar aplicações desenvolvidas nesta linguagem. Atualmente, é crescente o uso de Java para desenvolvimento de aplicações para dispositivos móveis, principalmente pelo fato de programas nessa linguagem serem automaticamente portáveis entre diferentes arquiteturas. Esta característica é fundamental em computação móvel, dada a variedade de fabricantes de computadores móveis atualmente no mercado.

Os eventos que a ferramenta proposta no trabalho simula são os seguintes:

- Desconexões: a idéia é que ao testar uma aplicação móvel utilizando a ferramenta, o desenvolvedor determine inicialmente a freqüência com que desconexões ocorrem no ambiente sem fio real, onde a aplicação será utilizada. Com isso, a ferramenta será capaz de simular os efeitos destas desconexões sobre a aplicação. Basicamente estes eventos podem produzir: diminuição do desempenho, término precipitado da aplicação, ativação de exceções não tratadas originalmente pela aplicação ou mesmo a ocorrência de um estado inconsistente na mesma. Ressalte-se que provavelmente nenhum destes erros seria detectado caso a aplicação fosse testada apenas em uma rede local.

- Variações na largura de banda da rede: a idéia é que o programador de uma aplicação móvel determine a banda de rede da qual a aplicação poderá fazer uso, bem como especifique como esta banda pode variar ao longo da área de cobertura. Em geral, em redes sem fio a largura de banda é bastante inferior àquela disponível em redes locais [4]. Assim, o programador poderá utilizar a ferramenta proposta para verificar os efeitos desta banda limitada sobre a aplicação. Basicamente, estes efeitos podem produzir: degradação no desempenho da aplicação e mesmo o término abrupto da mesma em qualquer estágio de sua execução.

O artigo encontra-se organizado como descrito a seguir. A Seção 2 apresenta uma visão geral de Java RMI. A Seção 3 descreve a ferramenta proposta, a qual foi denominada RMITK. A Seção 4 mostra um exemplo de uso da ferramenta. A Seção 5 descreve a implementação da ferramenta. A Seção 6 descreve alguns trabalhos relacionados. Por fim, a seção 7 conclui o artigo.

2. Desenvolvimento de Aplicações Distribuídas em Java

2.1 Java RMI: Conceitos Básicos

Java RMI (*Remote Method Invocation*) permite que objetos Java executando no mesmo computador ou em outros computadores comuniquem entre si por meio de chamadas de métodos remoto [8]. Essas chamadas de métodos são semelhantes àquelas que ocorrem entre objetos de um mesmo programa.

RMI está baseado em uma tecnologia anterior semelhante para programação procedural, chamada de *chamada de procedimentos remotos (Remote Procedure Calls,* ou RPC), desenvolvida nos anos 80. Um objetivo de RPC foi permitir aos programadores se concentrar nos requisitos funcionais de um aplicativo distribuído e, ao mesmo tempo, tornar transparente para o programador o mecanismo que permite que partes desse aplicativo se comuniquem através de uma rede. RPC encapsula todas as funções de comunicação e empacotamento dos dados, isto é, empacotamento de argumentos de função e valores de retorno para transmissão através de uma rede.

Sendo uma extensão de RPC, Java RMI permite comunicação distribuída de um objeto Java com outro. Uma vez que um método (ou serviço) de um objeto Java é registrado em um *Servidor de Nomes* como sendo remotamente acessível, um cliente pode pesquisar esse serviço e receber uma referência que permita utilizar o mesmo (isto é, chamar seus métodos). A sintaxe usada em chamadas de métodos remotos é idêntica àquela de uma chamada para um método de outro objeto no mesmo programa. Como ocorre em RPC, o empacotamento dos dados é tratado pelo RMI. O programador não precisa se preocupar com a transmissão dos dados sobre a rede. RMI também não exige que o programador domine qualquer linguagem particular para definição de interfaces, porque todo o código de rede é gerado diretamente a partir das classe existentes no programa.

2.2 Java RMI: Arquitetura

A arquitetura de Java RMI é dividida em três camadas:

- A camada de *stub*/*skeleton* oferece as interfaces que os objetos da aplicação usam para interagir entre si.

- A camada de referência remota é responsável por criar e gerenciar referências para objetos remotos;

- A camada de transporte implementa o protocolo que especifica o formato de solicitações enviadas aos objetos remotos pela rede.

A Figura 1 ilustra a organização dessas três camadas em uma aplicação RMI:

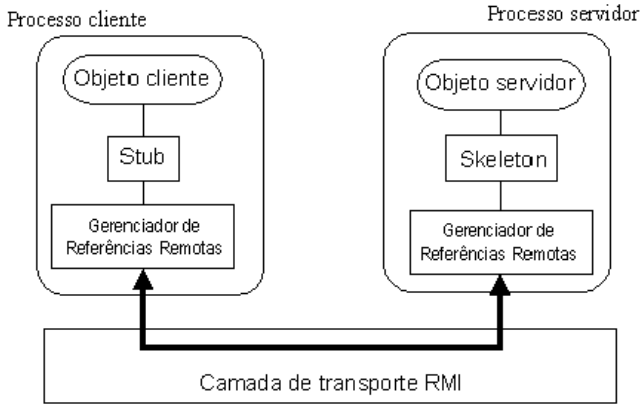

Figura 1 – Arquitetura de camadas de RMI.

2.3 Java RMI: Exemplo de Uso

No desenvolvimento de uma aplicação cliente-servidor usando Java RMI, é essencial que seja definida a interface de serviços que serão oferecidos pelo objeto servidor. A especificação de uma interface remota é equivalente à definição de qualquer interface em Java, a não ser pelos seguintes detalhes:

- A interface deverá, direta ou indiretamente, estender a interface Remote.

- Todo método da interface deverá declarar que a exceção RemoteException (ou uma de suas subclasses) pode ser gerada na execução do método.

O exemplo de definição de interface abaixo relaciona os métodos que serão invocados remotamente usando Java RMI. Este exemplo descreve uma aplicação de Chat onde clientes interagem com o servidor conectando, enviando mensagens e desconectando. O servidor interage com um cliente enviando mensagens de outros clientes.

```
public interface ServidorChat extends Remote {
  public String conectar (Cliente c) throws RemoteException;
  public void enviar (String mensagem) throws  RemoteException;
  public void desconectar(Cliente c) throws RemoteException;
}
```

Já o código mostrado a seguir descreve a interface de um cliente da aplicação de *Chat*, a qual contém o método *exibir* cuja função é mostrar a mensagem propagada pelo servidor da aplicação.

```
public interface Cliente extends Remote {
  public void Exibir(String mensagem) throws RemoteException;
}
```

Em seguida, classe *ServidorImpl* que estende *UnicastRemoteObject* implementa os métodos remotos *conectar*, *desconectar* e *enviar* definidos na interface servidor citada acima.

```
public class ServidorImpl extends UnicastRemoteObject
                          implements Servidor {
```

O método remoto *conectar* possui um vetor dinâmico onde são armazenadas referências para os objetos clientes conectados à aplicação.

```
public String conectar (Cliente c)  throws   RemoteException {
  conectados.addElement(o);
  return("Usuario Conectado");
}
```

O método remoto *desconectar* elimina uma referência remota armazenada no vetor de clientes conectados.

```
public void desconectar (Cliente c) throws RemoteException {
  conectados.remove(c);
}
```

O método remoto enviar descrito abaixo obtém uma referência para objetos dos clientes através do vetor de conectados e envia as mensagens para os mesmos.

```
public void enviar (String mensagem) throws RemoteException {
  do {
   Cliente c =(Cliente) conectados.elementAt(i);
   c.exibir(mensagem);
  } while (i<conectados.size());
}
```

O programa principal do servidor da aplicação de *chat* é responsável por registrar o objeto remoto do servidor no serviço de nomes de Java RMI, por meio do método *Naming.rebind ("Servidor",s)*, onde "Servidor" é o nome do objeto servidor e "s" é uma referência para o mesmo.

```
public static void main (String[] args) {
   try {
     Servidor s= new ServidorImpl();
     Naming.rebind("Servidor",s);
   } catch (Exception e) {
      System.out.println ("Erro no servidor: " + e.getMessage());
}
```

Descreve-se a seguir a implementação da aplicação cliente do sistema de *chat*. A principal classe do cliente é denominada *ClienteImpl*. Essa classe implementa o método "exibir" definido na interface Cliente.

```
public class ClienteImpl extends UnicastRemoteObject
                         implements Cliente {
```

O método exibir mostra as mensagens recebidas do servidor na tela do cliente.

```
public void exibe(String m) throws RemoteException {
  try{
    interface.append(m+"\n");
  } catch(Exception e ){
    System.out.println("Erro"+ e);
  }
}
```

O programa principal do cliente é responsável por acessar o Serviço de Nomes de Java RMI e obter uma referência remota para o objeto servidor. Dessa forma, o cliente poderá se conectar, enviar mensagens e se desconectar.

```
public static void main (String[] args) {
  Servidor s= (Servidor) Naming.lookup("Servidor");
  c.conectar(); // chamada remota
  ......
  c.enviar(mensagem);
  ......
  c.desconectar();
    ......
}
```

A execução da aplicação cliente-servidor em RMI requer, além da execução da aplicação cliente e da execução da aplicação servidora, a execução do Servidor de Nomes de RMI. O aplicativo *rmiregistry* faz parte da distribuição básica de Java e é encarregado de executar o Servidor de Nomes. Tipicamente, esse aplicativo é executado como um processo em *background* que fica aguardando solicitações em uma porta, que pode ser especificada como argumento na linha de comando. Se nenhum argumento for especificado, a porta 1099 é usada como padrão.

Em uma arquitetura de objetos distribuídos, nem sempre a comunicação no estilo cliente-servidor é suficiente para atender aos requisitos de todas as aplicações. É usual que um servidor RMI funcione algumas vezes como cliente, invertendo os papéis com o cliente original. O mecanismo para atingir esse objetivo é chamado de *callback*. Esta técnica é tipicamente utilizada quando a aplicação cliente requer um retorno do servidor mas não pode permanecer bloqueada aguardando uma resposta. Através dessa técnica, o servidor obtém uma referência para o cliente de forma que possa invocar remotamente um método do mesmo. Assim, quando a execução do serviço solicitado é concluída, o servidor pode

notificar o cliente através da invocação do método disponibilizado pelo mesmo para uso remoto.

3 A Ferramenta RMITK

A ferramenta RMITK descrita neste trabalho foi projetada para testar aplicações distribuídas em redes sem fio usando o *middleware* Java RMI. Para tanto, a ferramenta simula eventos característicos de um ambiente sem fio, como desconexões e variações na taxa de transmissão de dados. Alem disso a ferramenta RMITK é capaz de simular a ocorrência destes eventos em uma rede local tradicional, de formar a facilitar o teste de aplicações distribuídas no mesmo ambiente utilizado em seu desenvolvimento. A ferramenta é implementada por meio de um pacote Java, cuja interface de programação é descrita a seguir.

3.1 Descrição do Pacote RMITK

A ferramenta RMITK foi implementada como um pacote Java de nome *RMITK* . Este pacote é constituído por duas classes: *RMITK* e *RMITKproxy,* as quais são descritas a seguir :

- Classe *RMITK*: Esta classe possuis dois métodos, conforme mostrado abaixo.

```
public class RMITK{
  public static Object lookup(String name);
  public static void application_is_ready();
}
```

O método *lookup* tem como objetivo criar um objeto que estabelece uma conexão do cliente com o servidor e criar um *proxy* para esse objeto (descrito na seção 4). O método *application_is_ready* é um método que dá a opção ao programador que utiliza a ferramenta RMITK de testar a aplicação em modo real, ou seja, sem nenhuma desconexão ou variação de largura de banda.

- Classe *RMITKproxy*: Esta classe é constituída de métodos que criam um objeto interceptador para chamadas remotas de métodos. Esse objeto é responsável por simular eventos típicos de redes sem fio.

3.2 Utilização da Ferramenta RMITK

RMITK foi construída usando a tecnologia Java RMI. Logo, a mesma só poderá ser testada em aplicações distribuídas que utilizam Java RMI. Para o programador usar a ferramenta ele deverá seguir os seguintes passos:

- Adicionar o pacote *RMITK* no arquivo de criação da aplicação.

- Ao invés de usar o método tradicional *Naming.lookup("Servidor"),* deverá inserir em seu programa a seguinte linha de código *RMITK.lookup('Servidor')*. Assim tanto a chamada de métodos do cliente para o servidor e do servidor para o cliente serão interceptada e poderão simular os eventos típicos de uma rede sem fio .

4 Implementação

4.1 Dynamic Proxy Class

A partir do JDK 1.3, Java fornece suporte a objetos *proxy*, os quais são instâncias de classes *proxy* dinâmicas. Estas classes possuem duas propriedades principais: (i) seu código é criado em tempo de execução pela API de reflexividade de Java; (ii) elas implementam uma lista de interfaces especificadas em tempo da criação. Instâncias de classes *proxy* dinâmicas têm ainda um objeto manipulador de invocação (*invocation handler*), o qual é definido pelo cliente que solicitou a criação das mesmas.Toda a invocação de método sobre uma instância de uma classe *proxy* dinâmica é enviada automaticamente para o método `invoke` do manipulador desta instância, passando os seguintes parâmetros: a instância da classe *proxy* sobre a qual a invocação foi realizada; um objeto da classe `java.lang.reflect.Method`, o qual reifica o método que está sendo chamado; e um vetor do tipo `Object` que contém os argumentos deste método.

As classes dinâmicas *proxy* são úteis por exemplo para interceptar métodos que são invocados remotamente. Toda chamada remota feita por um objeto *proxy* passará pelo método `invoke` e poderá ser introduzido algum tipo de evento. Mostra-se na Figura 2 o padrão de projeto Proxy utilizando classes dinâmicas.

4.2 Uso de Dynamic Proxy Class na Implementação da Ferramenta RMITK

As classes *proxy* de Java tiveram um papel fundamental na implementação da ferramenta RMITK. Através delas, foi possível interceptar chamadas remotas e então executar um código para simular eventos característicos de redes sem fio.

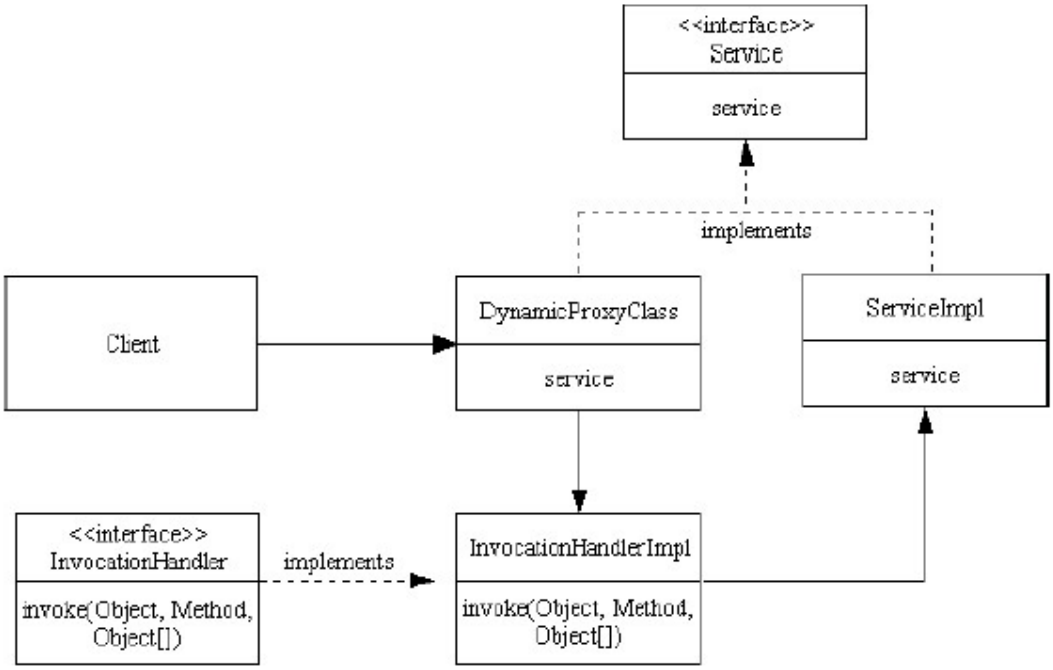

Figura 2 – Padrão de projeto *proxy* utilizando classes dinâmicas.

Java RMI propõe que para se obter uma referência para um objeto remoto é necessário primeiramente que este objeto esteja registrado em um Servidor de Nomes. Em um cliente, é feita uma chamada ao método *lookup*, o qual retorna uma referência para o objeto remoto. Esse objeto é representado na aplicação cliente pelo seu respectivo *stub*. O *stub* é encarregado de enviar chamadas remotas para o *skeleton* correspondente, que as repassa para o objeto remoto.

A solução adotada na ferramenta RMITK baseia-se em Java RMI, entretanto, ao invés da chamada ao método *RMITK.lookup* retornar uma referência para o *stub* do objeto remoto, ela retorna um uma instância de um *proxy* dinâmico, o qual por sua vez referencia o *stub*. Assim, quando uma chamada remota for realizada, a mesma passará primeiro pelo *proxy* e posteriormente chegará ao *stub,* seguindo então o fluxo normal de execução.

A Figura 3 ilustra o uso de classes *proxy* na implementação da ferramenta de RMITK.

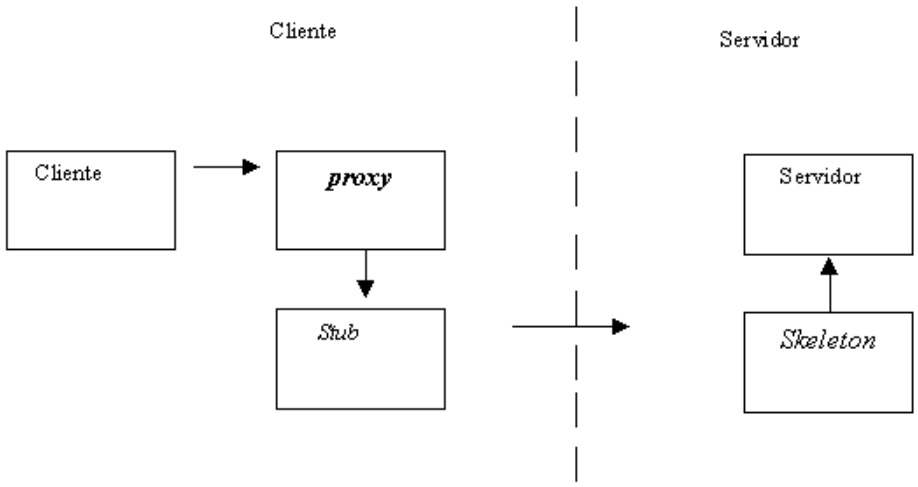

Figura 3 – Uso das classes *proxy* na implementação da ferramenta.

5 Exemplo de Utilização da Ferramenta : Sistema de Chat

Para testar a ferramenta RMITK, usou-se o sistema de *chat* descrito na Seção 2 A Figura 4 ilustra a interface gráfica de um cliente desse sistema.

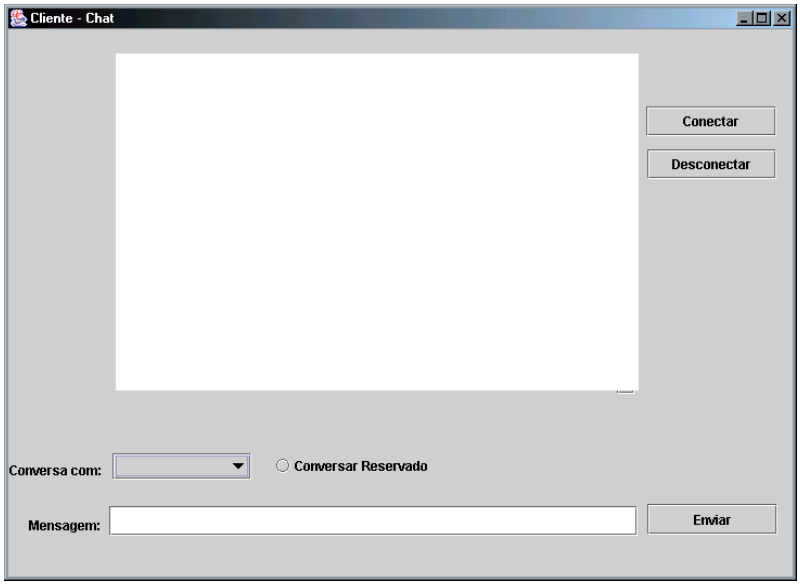

Figura 4 – Interface de um cliente do sistema *Chat*

Quando o usuário-testador conecta-se ao sistema, duas caixas de diálogo são abertas para que o mesmo informe a freqüência de desconexão e a largura de banda, conforme mostrado na Figura 5.

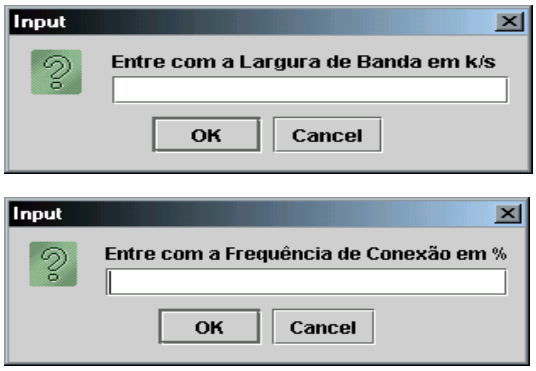

Figura 5 – Caixas de diálogo

Após o usuário informar os parâmetros para simular o uso da aplicação em um ambiente de rede móvel, cria-se automaticamente o *proxy* (interceptador) para um objeto remoto localizado no servidor da aplicação. Na forma de *callback* o servidor envia mensagens para os cliente através do método exibe(). A medida que o sistema entre em funcionamento mensagens são enviadas entre os usuários conectados e os eventos são simulados para as invocações remotas.

O interceptador inserido assume que o atraso é fixo para todas as chamadas de métodos remotos e que a freqüência de desconexões segue uma distribuição uniforme. Foi também implementada uma segunda versão deste interceptador que leva em consideração o tamanho do vetor de argumentos do método chamado para computar o atraso.

6 Trabalhos Relacionados

Injeção de falhas em *software* é uma técnica muito utilizada para se instrumentar, observar e testar diversos tipos de sistemas. Assim, RMITK pode ser considerada como uma ferramenta que se utiliza de técnicas de injeção de *software* para testar aplicações distribuídas em redes móveis.

Em [15], descreve-se um sistema, chamado Jaca, para injeção de falhas em programas Java. Em vez de ser baseado em reflexividade, como em RMITK, o sistema Jaca utiliza reengenharia de *bytecode* para injetar falhas em programas Java.

No caso de aplicações distribuídas baseadas em CORBA, existe o conceito de interceptadores portáveis, os quais permitem introduzir código no fluxo de uma chamada remota de métodos [14]. Este código pode ser introduzido tanto na parte cliente como na parte servidora de uma aplicação distribuída. No entanto, a implementação de interceptadores CORBA disponível no ambiente J2EE possui importantes limitações. Não é possível, por exemplo, ter acesso aos argumentos de uma chamada remota. Assim, dificulta-se a implementação dos atrasos em chamadas remotas, tal como proposto neste trabalho.

7 Conclusões

Basicamente, aplicações para computação móvel ainda são testadas de modo bastante empírico, o que pode acarretar a liberação para o usuário final de programas que não atendam a seus requisitos. Através da ferramenta RMITK, aplicações para computação móvel poderão ser testadas de modo automático, contribuindo assim para liberação de aplicações que atendam melhor aos requisitos dos usuários.

A ferramenta RMITK oferece uma interface de programação bastante simples. A interceptação proposta é dinâmica, já que parte do seu código é gerado em tempo de execução. Além disso, a interceptação é transparente, já que se localiza entre a camada de aplicação e a camada original de comunicação provida por Java RMI, sem, no entanto, requerer modificações em ambas. Não são requeridas também modificações na JVM original de Java.

A principal desvantagem da ferramenta é que ela não tem a capacidade de executar em aplicações desenvolvidas em outras linguagens, isto é, o servidor e o cliente terão que ser construídos em Java para que a ferramenta possa funcionar.

Referências

[1] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 3nd edition, 2000.

[2] Maurício Grego. *Internet de Bolso*. Info Exame, 16(183):44-54, May 2001.

[3] George H. Forman and John Zahorjan. *The Challenges of Mobile Computing*. IEEE Computer, 27(6), April 1994.

[4] Geraldo Robson e Antônio Alfredo Loureiro. *Introdução á Computação Móvel*. Décima Primeira Escola de Computação, 1998.

[5] Gruia-Catalin Roman and Gian Pietro Picco and Amy L. Murphy. *Software Engineering for Mobility: A Roadmap*, The Future of Software Engineering, 241--258, ACM Press, 2000.

[6] J. J. Kistler and M. Satyanarayanan. *Disconnected Operation in the Coda File System*, ACM Transactions on Computer Systems, 10(1):3-25, February 1992.

[7] M. Satyanarayanan. *Fundamental Challenges in Mobile Computing*. ACM Symposium on Principles of Distributed Computing, May 1996.

[8] Sun Microsystems. *Java Remote Method Invocation Specification*, October 1998.

[9] Wilson de Pádua Filho. *Engenharia de Software. Fundamentos, Métodos e Padrões*. LTC, 2001.

[10] Gruia-Catalin Roman and Gian Pietro Picco. *Workshop on Software Engineering for Mobility: Summary*, co-located with ACM/IEEE International Conference on Software Engineering, Toronto, CA, May 2001.

[11] Roger Pressman. *Software Engineering : A Practitioner's Approach, 5th edition, McGraw-Hill, 2000.*

[12] NTT Inc *DoComoNet Home Page*. http://www.nttdocomo.com/release/index.html.

[13] Gregor Kiczales. The Art of the Metaobject Protocol. MIT Press, 1991.

[14] Object Management Group. *CORBA/Portable Interceptor Specification*, 2000.

[15] Eliane Martins, Cecilia M.F. Rubira e Nelson G.M.Leme *JACA: A reflective fault injection tool based on patterns*. International Performance and Dependability Symposium (IPDS), June, 2002